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

TATIANA GADELHA SERRA DOS SANTOS

Reusing Values in a Dynamic Conditional Execution Architecture

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

Prof. Dr. Sergio Bampi
Advisor

Prof. Dr. Philippe Olivier Alexandre Navaux
Coadvisor

Porto Alegre, November 2004

CIP – CATALOGING-IN-PUBLICATION

Santos, Tatiana Gadelha Serra dos

Reusing Values in a Dynamic Conditional Execution Architecture / Tatiana Gadelha Serra dos Santos. – Porto Alegre: Programa de Pós-Graduação em Computação, 2004.

115 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2004. Advisor: Sergio Bampi; Coadvisor: Philippe Olivier Alexandre Navaux.

1. Superscalar Architecture. 2. Instruction Reuse. 3. Trace Reuse. 4. Multipath Execution. 5. Dynamic Conditional Execution. I. Bampi, Sergio. II. Navaux, Philippe Olivier Alexandre. III. Title.

To my mom and dad, the real heroes of this battle
Para a mãe e para o pai, os verdadeiros heróis dessa batalha

ACKNOWLEDGMENTS

This is certainly the best part on writing this Dissertation. The part that I have to look back and remember all good (and bad) times and people which crossed my way in the last 5 years. What can I say initially? Well... I already have tears on my eyes and I didn't even started!

It's pretty difficult to acknowledge all the people who contributed in this achievement. Hopefully, I won't forget you... But if for some reason your name is not here, please be sure that I didn't mean it. You're certainly in the bottom of my heart!

Let's start from the beginning, then. Mom and Dad. I think they are the real heroes here. They were always there for me. I owe them everything that I am and everything that I became. I thought about them all the times when I looked to my computer and said "OK, this is it! I'm giving up". Of course, I never gave up. And the simple fact of remembering how they taught me to be patient and persistent changed my mind right away. I really want them to be as proud to be my parents as I am to be their daughter. Priceless. My brother Yves, my little doctor, also deserves a big bite on this account. Even not talking to him as often as I want to, I know that there is a big, big love between us. I know that I can count on him and that he is there, ready to protect his little sister. It's really good to have him and to be sure about all this.

After my own family, but as important as they are, come Rafael and his family. He helped in every way that I can possibly remember. Even laughing at me when I was about to through my machine in the floor. He was and he is a great friend. And I'm very proud to say that I wouldn't have made it without his support. His family was also always present in my life in the past years, helping and supporting me in whatever they could. I'm really thankful for all of them.

My advisors Professor Sergio Bampi and Professor Philippe Navaux. Extraordinary people. They were always ready to listen, to help and, of course, to put me back on track when necessary. Today I can say that both of them are not just advisors, but they are friends, friends that I'll never forget. I'm not sure if I ever had the chance to tell how I admire them, but I'd like to take the time now. And I'd say: "When I grow up I want to be just like you!".

And the friends... I don't even know where to begin with. They are all awesome. They are all in my heart. And they all have to do with this work (even the ones who hate computers!).

First, my friends from UFRGS who were here in the bad mood days: Roberta (and Fabiano, Diogo and Pedro), Mônica, Rafael Bohrer, Mozart (and Chris), Ju, Patty, Nico, Pilla and Pizzol. Thank you, guys! Not only for your insights in this work, but specially to help me feel at home. Thank you for introduce me in the

coffee lovers life, thank you for all the lunches we had together, thank you for all sunny Sundays at the "Brique" and, above of all, thank you for your friendship. I will never forget you. I also won't forget my new friends and housemates, Márcia and Marta Pasin. They are really cool!

I'd like to remember my friends from UNISC as well: Rita, Rejane, Dani, Andrea and all others that made that environment such a nice place to work. And my students! All of them... They have a special place in my heart. In fact, sometimes I feel that they are my Professors. I sincerely hope that they have enjoyed our classes as much as I did.

And also the friends from the time I spent in USA and, mainly the Brazilian troupe: Nedja (and Andy, Ian and Lara), Dani (and Ewerton, Marcela and Leo), Andréia (and Joni and Luca), Maira (and Maurício and Enzo), Sérgio and Ricardo. Thanks for being my family during those years. Thank you for all parties, barbecues, trips and all the fun we had. You guys are terrific! Finally, all the old friends from Natal, especially Sadinha (and Rodrigo).

At last, I also would like to acknowledge CNPq for the scholarship, the Informatics Institute for the great support and, specially, the people from the Labtec and Corisco clusters for all their patience and time for running all my hundreds of simulations.

AGRADECIMENTOS

Essa é, sem dúvida, a melhor parte em escrever essa Tese. A parte em que eu tenho que olhar pra trás e relembrar de todos os bons (e maus) momentos e pessoas que atravessaram meu caminho nos últimos 5 anos. O que eu posso dizer inicialmente? Bem... eu já estou com os olhos mareados e nem ao menos comecei!

É bem difícil agradecer a todos que contribuiram com essa conquista. E, sinceramente, eu espero que eu não tenha esquecido você... mas se, por alguma razão, seu nome não está aqui, tenha certeza que não foi intencional. Você certamente está no fundo do meu coração!

Vamos, então, começar bem do princípio. O pai e a mãe. Eu realmente acho que eles são os verdadeiros heróis. Eles sempre estiveram aqui por mim. E eu devo pra eles tudo que sou e que me tornei. Todas as vezes que eu olhei para o computador e disse "OK, é o fim! Vou desistir", foi neles que pensei. É claro que nunca desisti. E o simples fato de lembrar como eles me ensinaram a ser paciente e persistente mudava meu pensamento. Eu realmente quero que eles tenham tanto orgulho em ser meus pais como eu tenho orgulho de ser filha deles. E isso não tem preço. Meu irmão Yves, meu Doutorzinho, também merece um grande crédito. Mesmo não conversando com ele com a freqüência que eu gostaria, eu sei que existe um grande amor entre a gente. Eu sei que posso contar com ele e ele sempre estará ali, pronto para proteger a irmãzinha. É realmente bom tê-lo e ter certeza de tudo que estou dizendo aqui.

Depois da minha própria família, mas tão importante quanto ela, vem o Rafael e a família dele. Ele me ajudou de todas as maneiras que eu posso possivelmente tentar lembrar. Mesmo nas vezes que ele ria de mim quando eu estava quase jogando minha máquina no chão. Ele foi e ele é um grande amigo. E eu tenho muito orgulho em dizer que eu não teria terminado sem o apoio dele. A família dele também sempre esteve presente na minha vida nos últimos anos, me ajudando e apoiando em tudo que eles podiam.

Meus orientadores Professores Sergio Bampi e Philippe Navaux. Pessoas extraordinárias. Eles estavam sempre prontos para ouvir, ajudar e, claro, me colocar de volta no caminho certo quando necessário. Hoje eu posso dizer que os dois não são apenas orientadores, mas amigos, amigos que eu nunca vou esquecer. Eu não tenho certeza se eu já tive chance de dizer o quanto eu os admiro, mas eu gostaria de fazê-lo agora. E diria: "Quando crescer, quero ser igualzinha a vocês!".

E os amigos... Eu nem mesmo sei por onde começar. Todos eles são fenomenais. Todos eles estão no meu coração. E todos eles tem sua parcela nesse trabalho (inclusive aqueles que odeiam computação!).

Primeiro, os amigos da UFRGS, que estavam aqui nos dias de mau humor:

Roberta (e Fabiano, Diogo e Pedro), Mônica, Rafael Bohrer, Mozart (e Chris), Ju, Patty, Nico, Pilla e Pizzol. Obrigada, galera! Não só pelas idéias nesse trabalho, mas especialmente por ajudarem a me sentir em casa. Obrigada por me introduzirem no mundo dos amantes do café, obrigada por todos os nossos almoços, obrigada por todos os domingos ensolarados no brique e, acima de tudo, obrigada pela amizade de vocês. Eu nunca vou esquecê-los. Eu também nunca vou esquecer minhas novas amigas que dividem o ap, a Márcia e a Marta Pasin. Elas são muito legais!

Eu gostaria de lembrar também dos meus amigos da UNISC: Rita, Rejane, Dani, Andrea e todos os outros que fizeram daquele ambiente um lugar tão legal de trabalhar. E os meus alunos! Todos eles... Eles têm um lugar especial no meu coração. De fato, às vezes sinto que eles são os meus professores. Eu sinceramente desejo que eles tenham curtido nossas aulas da mesma forma que eu curti.

Também aos amigos feitos enquanto morava nos EUA e, principalmente, a tropa brasileira: Nedja (e Andy, Ian e Lara), Dani (e Ewerton, Marcela e Leo), Andréia (e Joni e Luca), Maira (e Maurício e Enzo), Sérgio e Ricardo. Obrigada por serem minha família durantes aqueles anos. Obrigada por todas as festas, churrascos, passeios e toda a diversão que sempre fizemos. Vocês são realmente maravilhosos! Finalmente, todos os meus velhos amigos de Natal, especialmente a Sadinha (e o Rodrigo).

Por último, eu gostaria de agradecer ao CNPq pela bolsa de doutorado, ao Instituto de Informática pelo grande suporte e, especialmente, às pessoas dos clusters Labtec e Corisco por toda paciência e tempo para rodar as minha centenas de simulações.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
BHB	Block History Buffer
BTB	Branch Target Buffer
CIDI	Control Indenpendent Data Independent
CPU	Central Processing Unit
D–cache	Data Cache
DCE	Dynamic Contional Execution
DRAM	Dynamic Random Access Memory
DTM	Dynamic Trace Memoization
FP	Floating Point
FU	Functional Unit
I–cache	Instruction Cache
IC	Input Context
ILP	Instruction Level Parallelism
Int	Integer
IPC	Instructions Per Cycle
L1	cache Level 1
L2	cache Level 2
LRB	Load Reuse Buffer
LRU	Least Recently Used
OC	Output Context
OS	Operating System
PC	Program Counter
RAM	Random Access Memory
RAW	Read After Write
RB	Reuse Buffer

ROB	ReOrder Buffer
RISC	Reduced Instruction Set Computer
RTM	Reuse Trace Memory
SMT	Simultaneous MultiThread
Sn	reuse Scheme (name)
Sv	reuse Scheme (value)
Sv+d	reuse Scheme (value and dependence)
SRAM	Static Random Access Memory
TLB	Translation Lookaside Buffer
TB	Trace Buffer
WAR	Write After Read
WAW	Write After Write

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The Dynamic Conditional Execution (DCE) is 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 consequence, a smaller number of predictions is performed, and therefore, a lower number branches is mispredicted.

Nevertheless, as other multipath solutions, DCE requires a more complex control engine. In a DCE architecture, one may observe that several replicas of the same instruction are dispatched to the functional units, blocking resources that might be used by other instructions. Those replicas are produced after the join point of the paths and are required to guarantee the correct semantic among data dependent instructions. Moreover, DCE continues producing replicas until the branch that generated the paths is resolved. Thus, a whole section of code may be replicated, harming performance. A natural alternative to this problem is the attempt to reuse those replicated sections, namely the replicated traces.

The goal of this work is to analyze and evaluate the effectiveness of value reuse in DCE architecture. As it will be presented, the principle of reuse, in different granularities, can reduce effectively the replica problem and lead to performance improvements.

Keywords: Superscalar Architecture, Instruction Reuse, Trace Reuse, Multipath Execution, Dynamic Conditional Execution.

Reusando Valores em uma Arquitetura com Execução Condicional Dinâmica

RESUMO

A Execução Condicional Dinâmica (DCE) é uma alternativa para redução dos custos relacionados a desvios previstos incorretamente. A idéia básica é buscar todos os fluxos produzidos por um desvio que obedecem algumas restrições relativas à complexidade e tamanho. Como conseqüência, um número menor de previsões é executado, e assim, um número mais baixo de desvios é incorretamente previsto.

Contudo, tal como outras soluções multi-fluxo, o DCE requer uma estrutura de controle mais complexa. Na arquitetura DCE, é observado que várias réplicas da mesma instrução são despachadas para as unidades funcionais, bloqueando recursos que poderiam ser utilizados por outras instruções. Essas réplicas são geradas após o ponto de convergência dos diversos fluxos em execução e são necessárias para garantir a semântica correta entre instruções dependentes de dados. Além disso, o DCE continua produzindo réplicas até que o desvio que gerou os fluxos seja resolvido. Assim, uma seção completa do código pode ser replicado, reduzindo o desempenho. Uma alternativa natural para esse problema é reusar essas seções (ou traços) que são replicadas.

O objetivo desse trabalho é analisar e avaliar a efetividade do reuso de valores na arquitetura DCE. Como será apresentado, o princípio do reuso, em diferentes granularidades, pode reduzir efetivamente o problema das réplicas e levar a aumentos de desempenho.

Palavras-chave: Arquitetura Superescalar, Reuso de Instrução, Reuso de Traços, Execução de Multi-fluxos, Execução Condicional Dinâmica.

1 INTRODUCTION

On the latest years, the increasing demand for performance makes computational system design more complex and sophisticated. Robust and heavy softwares require fast processors with the capability to execute hundreds of millions of operations per second. For general purpose applications, where it is difficult to specialize hardware architecture and components, the challenge is even larger and it is not an ordinary task.

In order to supply the market with these high performance requirements, hardware designers have been working over dozens of innovations every year. During the last generations, however, superscalar microprocessors are dominating the general purpose microprocessors architectures. The execution stage of those machines has the potential to execute several instructions per cycle due to the many functional units available.

The number of functional units in the execution stage may vary according to the microprocessor design, while 8 or more functional units are commonly found in commercial processors (KESSLER, 1999; HOREL; LAUTHERBACH, 1999; INTEL, 2001). Even with this potential, the effective number of instructions executed per cycle (IPC) is low. Typically, state–of–the–art microprocessors do not achieve, in average, an effective IPC equal or larger than 2 (HENNESSY; PATTERSON, 2003).

There are three major problems related to this performance bottleneck in superscalar architectures (JOHNSON, 1991). Data dependencies, resource conflicts and control dependencies are the greatest limiters found and the effort to effectively deal with them is the most difficult challenge faced by microprocessor designers.

Data dependencies limit the parallelism extracted from the executed code because, if one or more instructions depend on previous instructions result, it is not possible to execute them in parallel. This may cause stalls in the pipeline until those instructions are executed. However, data dependencies are efficiently treated dynamically by the hardware through mechanisms such as register renaming (TOMASULO, 1967) and scoreboard (THORNTON, 1964).

Resource conflicts can be observed when two or more instructions try to use the same resource at the same time. Thus, one or more instructions may be pending until the resource is freed. In general, designers may replicate the hardware resources that most often are demanded and this normally presents an efficient result. The balance of the architecture has to be carefully reviewed and cost vs. benefit ratio and trade-offs analyzed, as more area and power are spent to have more parallel functional units. However, this is not the main bottleneck in contemporary superscalar architectures.

On the other hand, control dependencies produce a large penalty due to mispre-

dicted branches and represent one of the most significant barriers to achieve higher IPC.

There are several alternatives to reduce the problem caused by branches, but no definitive solution has been found yet. Branch prediction is the oldest and more widely used technique. Contemporary mechanisms provide very accurate predictions but indeed do not predict all branches correctly and the occurrence of a small number of mispredictions, on the order of 3% to 7% of all predicted branches, can decrease significantly the performance especially in very deep pipelines.

A misprediction is detected only after the complete execution of a branch. Thus, all instructions fetched, decoded, dispatched and executed after the mispredicted branch are squashed. This also includes control independent instructions, because the misprediction recovery mechanism usually flushes all younger instructions and restarts the fetch from the correct target.

Besides branch prediction, other mechanisms have been the target of studies, investigations and developments. Many architectures use sophisticated techniques to extract Instruction Level Parallelism (ILP) and increase performance (POSTIFF et al., 1999). Typically, this parallelism may be extracted and explored using one or more execution paths, depending on the approach adopted by the architecture.

The Dynamic Conditional Execution (DCE) is a wide-issue superscalar processor that exploits the locality of conditional branches to apply dynamic predication and multipath speculative execution. The result is a highly aggressive architecture that can reduce misprediction penalties. The general principle behind DCE is to fetch both paths of some conditional branches based on a semi-static selection mechanism. Both paths of selected branches are executed concurrently and the wrong paths are squashed selectively according to the branch result (SANTOS, 2003).

The major benefit of pursuing such approach is to reduce the number of predicted branches. As a consequence, the number of mispredictions is also reduced without requiring a special instruction set.

The problem of dynamically predicating complex branches is the introduction of a large overhead that hides the potential benefit of applying the technique. The overhead is introduced by fetching instructions from multiple paths, which will be later squashed. Moreover, DCE approach also creates multiple replicas of the same logical instruction in order to guarantee correct data dependency. This issue is detailed later on this PhD Dissertation.

Despite the overhead introduced, previous studies have shown that predicating complex branches produces a two-fold misprediction reduction with relation to predicating simple branches only (SANTOS, 2003). The speed-up obtained by predicating complex branches, however, only pays off for very wide issue machines due to the overhead.

One alternative to this problem can be found in the principle of reusing instructions. Instruction reuse, in different granularities, was a widely explored idea in previous works (GONZALEZ; TUBELLA; MOLINA, 1999; SODANI, 2000; COSTA; FRANÇA; CHAVES FILHO, 2000; WU; CHEN; FANG, 2001; COSTA, 2001; PILLA et al., 2002), but none of them were developed specifically for an architecture with dynamic predication. Most of those studies performed in conventional superscalar architectures pointed out that a large number of instructions can be reused. In DCE, this number is potentially greater due to the replicas occurrence. Thus, reducing the overhead can make resources available to issue other useful instructions.

The main goal of this work is to analyze the limits and potentials of instruction and trace reuse mechanisms to reduce overhead and ultimately lead to performance improvements in DCE architecture.

Chapter 2 addresses the impact of conditional branches on performance, while Chapter 3 depicts the DCE – Dynamic Conditional Execution architecture as an alternative to this problem. Chapter 4 presents the DCE main limitations, as a motivation for this work. Chapter 5 addresses previous works in instructions and trace reuse, while Chapter 6 presents the mechanisms to reuse values in DCE. Chapter 7 presents the simulation environment, and Chapter 8 shows the results achieved. Finally, Chapter 9 shows the conclusions, remarks and future work of this research.

2 CONDITIONAL BRANCHES AS PERFORMANCE LIMITER

The main function of the fetch stage is to feed the pipeline with new instructions. Apparently, this is a trivial task, but some restrictions apply to this stage, producing stalls and reducing the global performance of the architecture.

Figure 2.1 presents a scheme of the instruction fetch stage. The stage, after fetching an I-cache line, identifies whether there are branches among those instructions or not. Typically, this task is performed by analyzing the instruction opcode or by checking the branch prediction tables. If a branch is detected, the fetch stage forward that instruction to the predictor, which returns the direction to be followed by the instructions flow as well as the next probable address. When the prediction outcome is that the branch is taken, the fetch stage flushes all instructions after the branch and redirects the fetch to the correct PC. In case of not-taken prediction, the fetch stage keeps on accessing instructions sequentially.

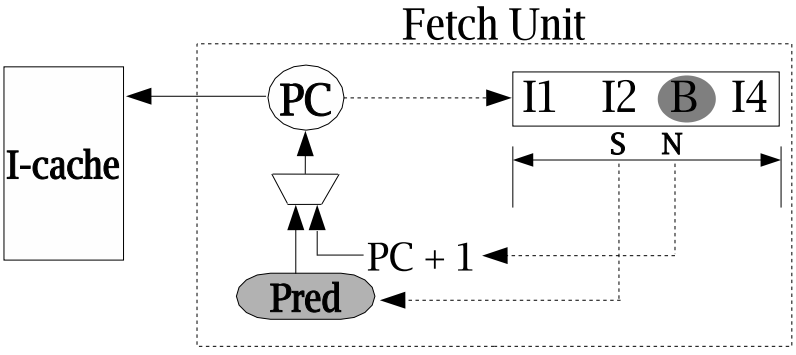

Figure 2.1: Fetch stage

Basically, the fetch stage stalls due to four reasons: I-cache misses, branch instructions occurrence, mispredictions and instruction buffer full. Each one of these implies an additional latency or a penalty to the pipeline.

Cache miss is an old and a difficult problem to be solved. The memory hierarchy is the most common alternative to reduce this bottleneck, especially because the gap between memory and processor frequencies is widening. Cache memories have achieved some success and are essential in current high performance architecture designs. But even with 128 Kbytes or more in first level on-chip caches, misses are still a concern. Each miss has a latency that varies, typically, from 2 to 80 cycles, depending on which hierarchical level the hit occurs. Hits on L2 caches cause the order of 10–15 cycles penalty, if the L2 cache is off chip. If there is a miss in the L2

cache, a latency of 80 cycles or more may be associated to the main memory access. During all those cycles, the fetch stage is stalled and none of the pipeline stages get new instructions.

In order to reduce misses and to minimize the number of cycles wasted by the fetch stage, the lower levels of the memory hierarchy are getting larger. L1 caches of up to 128 Kbytes are already available in commercial microprocessors (KESSLER, 1999), against the usual 8 Kbytes to 16 Kbytes used in just a few years back. The increase in the L1 cache size, however, is not a good solution for the next generation of high performance processors. The cost of the large capacity increases not just the die area, but mainly the L1 access delay and the additional logic for SRAM implementation as well as an efficient addressing scheme to this large on chip memory. Keeping L1 cache access delay within 1 to 2 pipeline cycles severally limits the L1 size.

After using L1 caches with large size in previous generations, most microprocessors families are integrating one more hierarchy level on chip. The idea is to keep L1 capacity small and to improve performance through the reduction of L2 access time by means of L2 integration. Pentium 4, for example, has D-cache and I-cache with 16 Kbytes each, with L2 on chip (INTEL, 2001). The intention is to make the L2 deliver data and instructions at the same frequency of the processor, reducing the 12 cycles typical latency to a few CPU cycles only.

Even with all this effort, new mechanisms are still necessary to decrease cache miss rates and to hide the access latencies. Data and instruction prefetches appear as a low cost and good results alternative (SANTOS, 2000).

The branch occurrence may also stall the fetch stage. As mentioned before, the fetch stage detects branches as soon as the line is fetched from the I-cache. When a branch is encountered and predicted as taken, the fetch stage stalls in order to redirect the fetch. Even if the prediction is correct, the fetch width is harmed.

Figure 2.2 presents an example of how the instruction width is harmed due to branch occurrence.

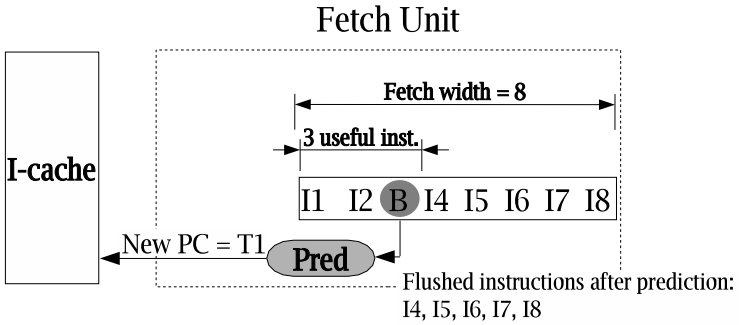

Figure 2.2: Fetch width with branch occurrence

It is possible to observe that the fetch width, in the example showed in Figure 2.2, is reduced to only three instructions, in case of a taken branch is fetched. The remaining pipeline stages are equally affected and will perform for one cycle with only three instructions. Thus, even with a correct prediction, the performance of the architecture is limited by a single occurrence of a branch instruction.

It is also observed that the situation may be worse if the target is located at the beginning of a cache line. This problem is known as cache alignment.

The cache port allows only one line to be accessed each time. Normally, an entire line is fetched and part of it is flushed in a branch occurrence. Thus, it is not possible to fetch the target followed by its subsequent instructions and, at the same time, fetch the instructions in the next line necessary to complete the fetch width. The fetch width is harmed once again and only the instructions located in the same cache block are used. In the worst case, the fetch width is reduced to only one instruction. This can occur if the target is at the last position in the line. The so called target-word-first mechanism treats this problem in cache accesses (HENNESSY; PATTERSON, 2003), hence optimizing the access time, but not avoiding the harm to the effective fetch width.

The branch predictors are generally based in tables that store the pattern followed by each branch in previous executions. Those tables have a limited size that do not support all branches found in a program. The prediction tables and caches are similarly structured, presenting a given associativity and mapping more than one entry to the same position.

After introducing a branch predictor with several levels, the architecture is sensible to 2 different kinds of mispredictions. The first one occurs when the predictor returns a target of another branch, mapped onto the same position of the table. The second type of misprediction is worse and the penalty is even larger. This penalty applies when the branch prediction mechanism finds the right branch, but the result of the prediction is wrong. Nevertheless, even with these additional penalties when using more than one prediction table, mechanisms like this achieve good performance and they are the most widely used in current microprocessors.

In the occurrence of a miss in the addresses table, only the direction is returned by the prediction. This is yet another problem found in branch predictors. In this case, the fetch cannot be redirected properly and the subsequent instructions keep on being fetched. The penalties are also different, depending on the branch. If a branch with direct addressing causes the miss, it is possible to find the correct target after the decode stage, in the dispatch. And if the target found in the instruction is different from the sequential one, the fetch is redirected in the next cycle. On the other hand, if the branch is register dependent, its result can be just found only after execution. In this case, the penalty is larger and affects the whole pipeline. The effect is worse when the pipeline is deep (INTEL, 2001).

Figure 2.3 presents the penalty for the hit of a different target mapped to the same position of the fetched branch. In Figures 2.3 and 2.4 the fetch width is assumed to be equal to 4 instructions.

In this example, when a given branch is found among the fetched instructions the predictor indicates a target to be fetched in the next cycle. This target address is located in the branch table, at the same position that the right address would be, if no conflicts had happened. The branch predictor returns this address to the fetch stage. Then, the fetch stage stalls and redirects to that address. The instruction incorrectly found as target, as well as its subsequent instructions, are fetched from the cache and delivered to the decode stage. The decode stage performs normally, decoding those instructions as usual. Meanwhile, the branch hit the dispatch stage and it is possible to identify whether the predicted target is the same one found in the instruction opcode. This, however, applies only for direct branches and does not happen with register dependent branches, as mentioned before. If the two addresses are different, the dispatch sends a signal to the fetch stage. The pipeline is then

flushed, but only up to the dispatch stage. Instructions in the dispatch stage fetched ahead of the branch do not need to be squashed. In this stage, the instructions are still in order and it is possible to squash just the instructions from the misfetched path. The instructions in the dispatch, issue, execution and write back continue processing normally.

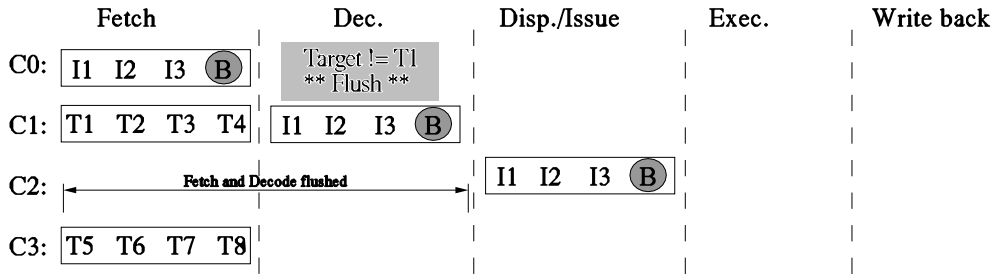

Figure 2.3: Misfetch occurrence

On the other hand, a much larger penalty applies when the direction predicted is wrong. Figure 2.4 presents the number of lost cycles due to only one misprediction in a short, five stage pipeline.

It is possible to see that, after a misprediction, the fetch is incorrectly redirected and a sequence of useless instructions is brought into the pipeline. The branch is solved only three cycles after its prediction, at the end of the execution stage. While this result is not known, the pipeline performs unnecessary operations. And even knowing the result, the fetch will be redirected again only in the write back, when the mispredicted branch reaches the top of the Re-order Buffer (RoB). As discussed before, this is necessary in order to assure that instructions fetched before the branch will not be flushed. The pipeline is then squashed and re-started in the next cycle. Additional cycles are needed to re-load all the stages again.

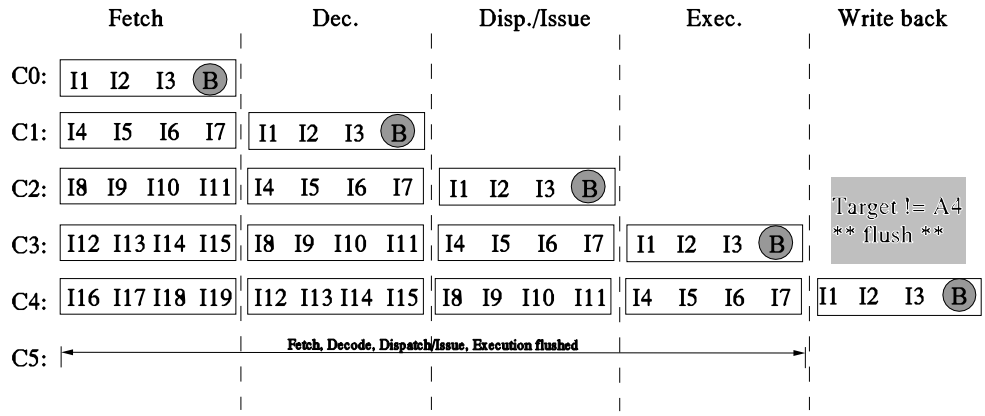

Figure 2.4: Misprediction occurrence

Simulations were performed in order to illustrate the issues discussed. The results are shown in the next Sections.

2.1 Simulation Environment

In order to quantify the hazards described in the previous Section, simulations were ran using the SimpleScalar Tool Set (BURGER; AUSTIN, 1997). The superscalar architecture simulator, called *sim-outorder*, is very detailed and implements state-of-the-art microprocessor features, such as out-of-order execution and 3-level memory hierarchy. Also, the simulator is highly configurable, allowing the user to decide many elements like architecture width, memories capacity, number of Functional Units (UFs) and others.

Figure 2.5 presents the *sim-outorder* pipeline. The instructions decode is performed in the second stage with the dispatch. The scheduler is responsible for the instructions issue. The stages basically work according to a conventional superscalar architecture.

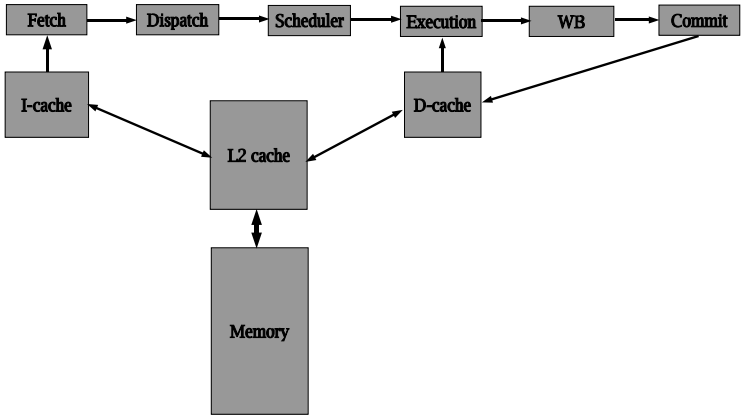

Figure 2.5: *Sim-outorder* pipeline

Table 2.1 presents the basic configurations used in the set of experiments. The fetch, decode, dispatch and issue widths are equal to 8 instructions. The Reorder Buffer (RoB) has 128 entries, which means that a maximum of 128 instructions may be waiting to be committed. The load and store instructions are handled differently in a 64-instruction queue. Ten FUs are available (4 integer ALUs, 2 integer mult/div, 3 FP ALUs, 1 FP mult/div). Cache level 1 is splitted into I-cache and D-cache, both with 64 Kbytes, 2-way set associative. L2 cache is unified with 512 Kbytes and 4-way set associative. Hits in L1 caches, L2 cache and main memory are resolved in 1, 12 and 80 cycles, respectively. The misprediction penalty is, at least, 7 clock cycles.

A perfect branch prediction was used in a base architecture. However, a hybrid predictor was used to analyze the impact of branch occurrence and mispredictions.

The hybrid predictor (MCFARLING, 1993) combines two other predictors, the 2-level adaptative (YEH; PATT, 1991) and the bimodal (SMITH, 1981). Moreover, there is an additional table responsible to choose which kind of the predictions will take place in a given moment. In the simulations performed in this part of the work, this table was configured as a 2048-entries table. The 2-level adaptative is the global predictor, with a history register (first level) that indexes a 4096-entries table (second level). On the other hand, the bimodal mechanism has a 2048-entries table.

A subset of SPEC2000 programs was used as benchmarks for these simulations (HENNING, 2000). They are: *bzip2*, *ggc*, *equake*, *mesa*, *parser*, and *vpr*. Table 2.2

Table 2.1: Configurations used in experiments

Parameter	Configuration
Fetch width	8 instructions
Decode width	8 instructions
Dispatch width	8 instructions
Issue width	8 instructions
RoB entries	128 instructions
Load/Store queue	64 instructions
Integer FUs	4 FUs
Integer Mult/Div	2 FUs
FP FUs	3 FUs
FP Mult/Div	1 FUs
Memory bus width	128 bytes
L1 I-Cache	64 Kbytes
L1 D-Cache	64 kbytes
L2 unified cache	512 Kbytes
Misprediction penalty	7 cycles
L1 latency access	1 cycle
L2 latency access	12 cycles
Main memory latency	80 cycles

shows the benchmarks followed by the inputs and arguments used in the simulations.

Table 2.2: Benchmarks and inputs used in the experiments

Benchmark	Inputs
bzip2 (Int)	input.source 58
gcc (Int)	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O cp-decl.i -o cp-decl.s
equake (FP)	< ref.in
mesa (FP)	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm
parser (Int)	2.1.dict -batch < ref.in
vpr (Int)	ref.net ref.arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9 412 -inner_num 2

For each configuration, 400 million instructions were simulated. The results, however, were processed after the execution of the first 100 million of instructions.

2.2 The Effect of the Control Dependencies

The fetch stage of the pipeline may stop due to four reasons: I-cache misses, instruction buffer full, mispredictions and misfetches.

Figure 2.6 presents the number of cycles lost due to each one of those reasons. The first bar means the number of cycles wasted by I-cache miss occurrence. The

second bar depicts the number of cycles wasted with mispredicted branches. The third bar shows the number of cycles lost due to misfetches, while the last bar means the cycles lost with instruction buffer full. These results were achieved using the hybrid predictor.

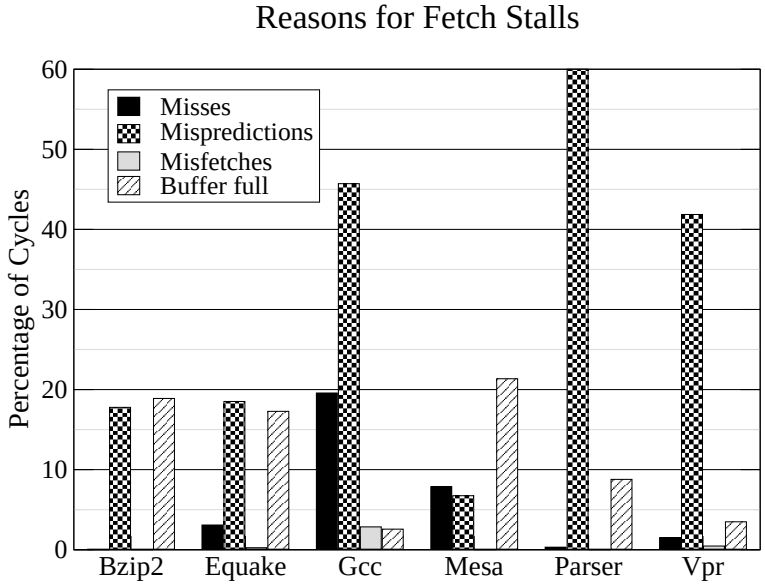

Figure 2.6: Fetch stage stalls

It is possible to see that, for this configuration, which is closer to a typical modern superscalar architecture, the main problem is relative to mispredicted branches occurrence. Also, I-cache misses are a concern, but mainly in large applications such as *gcc*.

Buffer full is the main reason for stalls in benchmarks *mesa* and *bzip2*. In fact, for benchmark *mesa*, mispredictions are only the third main reason for pipeline stalling. This is due to the high accuracy rate achieved by the predictor, which reduced the number of wasted cycles.

Figure 2.7 shows the number of cycles lost in case of mispredictions. The dark bar shows the percentage of cycles wasted with mispredictions. The most extreme case is the benchmark *parser*, with almost 60% of the cycles spent recovering from mispredicted branches. This means that, in this case, the architecture spent 60% of its time fetching, decoding, dispatching, issuing and executing instructions from the wrong path.

Other benchmarks, such as *gcc* and *vpr*, also achieved very negative results. And even with a high accuracy rate in the prediction, benchmarks *bzip2* and *equake* spent, respectively, 17% and 18% of the time recovering from mispredictions.

Misprediction is not the only problem caused by branches. Branch occurrence, even if correctly predicted, may harm the architecture performance. In fact, instruction alignment in the I-cache is also a significant problem and these effects can be visualized in Figure 2.8. The dark bar shows the number of instructions in the fetch buffer using the hybrid predictor, while the gray bar depicts the same number, but using perfect prediction. The hybrid predictor combines two different prediction schemes, alternating their usage according to the branch pattern (local prediction) or to the recent branch history (global prediction). This hybrid scheme is used in several state-of-the-art microprocessor and guarantees accurrate prediction rates of

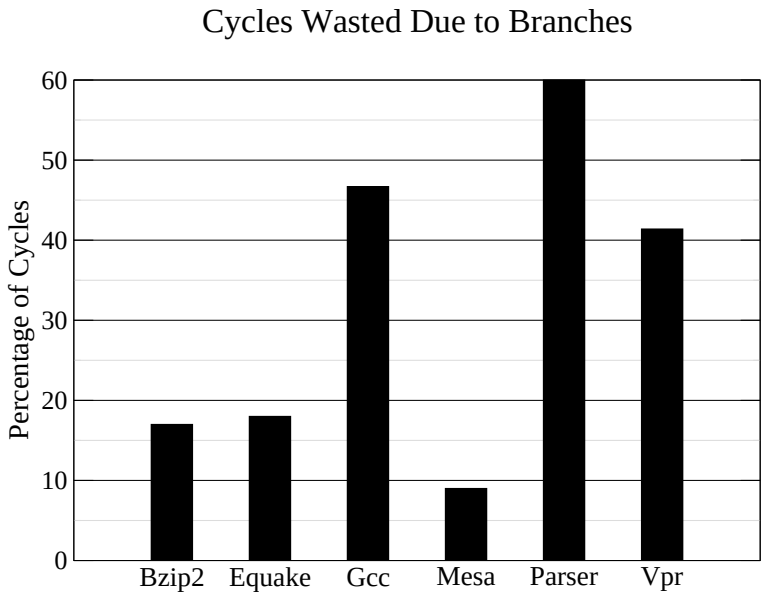

Figure 2.7: Percentage of cycles wasted due to branch occurrence

up to 98%. In the simulations performed in this work the hit rates in the prediction tables ranged between 82% and 93% for the hybrid scheme.

In average, the number of instructions delivered to the dispatch stage is lower than 5, even for perfect prediction. Ideally, both architectures have the potential to deliver 8 instructions per cycle. This means that, in average, a little over half of the fetch resources are used during execution. This number is extremely low, if compared with the number of instructions that could be fetched. As a consequence, the number of instructions executed per cycle (IPC) is going to be smaller, reducing the overall performance.

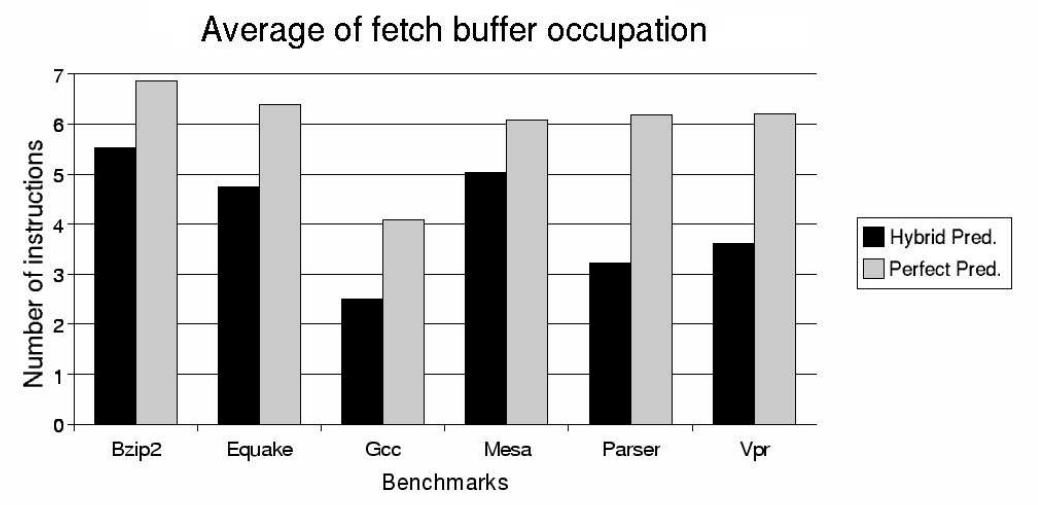

Figure 2.8: Instructions fetched per cycle

Figures 2.9, 2.10 and 2.11 depict this problem, presenting the distribution of instructions fetched during simulations in each benchmark. All horizontal axis show the number of instructions fetched, while vertical axis present the percentage of the

distribution. The dark bars show results with hybrid prediction, while the gray bars are the results with perfect prediction. In an ideal architecture, all benchmarks should be spending 100% of the time dispatching 8 instructions, the maximum number supported by the architecture.

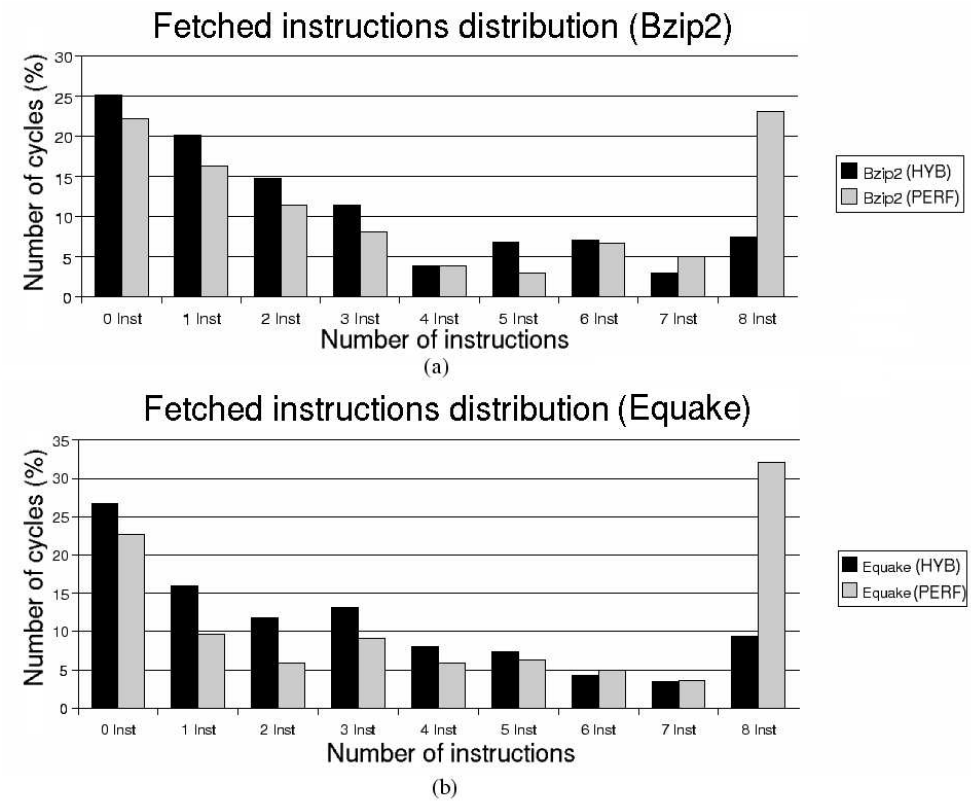

Figure 2.9: Distribution of instructions in benchmarks (a) bzip2 and (b) equake

Unfortunately, the distribution of instructions fetched is very different from the ideal. It is possible to observe that, in all cases, there are a significant percentage of cycles in which no instructions were fetched. This situation is worse when using hybrid prediction because, in this case, the fetch stage also stalls due to mispredictions. The difference between using perfect and hybrid prediction is significant and, in some cases, such as benchmark *vpr*, achieves 60%. However, I-cache misses may also increase the percentage of cycles in which no instructions were fetched. This explains the high rates reached by the cases of zero or few instructions fetched per cycle.

Even using perfect prediction, the number of cycles spent fetching 8 instructions is small. The best case occurs in benchmark *vpr*, which could ideally fetch 8 instructions in 35% of the cycles. In benchmark *bzip2*, however, 8 instructions were fetched in just 23% of the cases. These rates are low because perfect prediction solves only mispredictions. Instruction alignment and I-cache misses are additional problems preventing a larger fetch bandwidth in the superscalar architecture.

During most of the time, a lower number of instructions is fetched when hybrid prediction is used. The best case was achieved by benchmark *mesa*, which fetched 8 instructions during 10% of the time. In general, however, for more than 40% of the cycles the architecture is fetching 1 to 4 instructions per cycle.

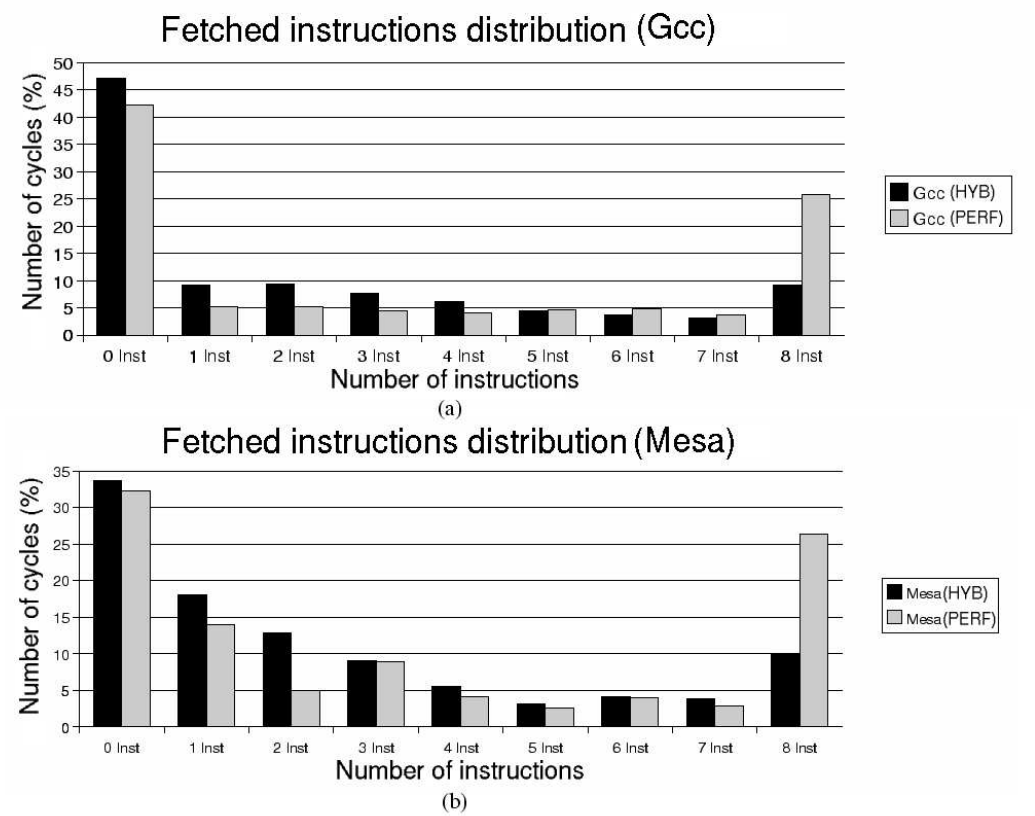

Figure 2.10: Distribution of instructions in benchmarks (a) gcc and (b) mesa

Table 2.3 shows the effective fetch bandwidth for each benchmark. The first column presents the benchmarks simulated. The remaining columns mean the average number of instructions fetched per cycle by the architecture with the hybrid predictor followed by the one using a perfect predictor. In all cases, the number of instructions fetched by the architecture with the perfect predictor is significantly higher. In benchmark *vpr*, where the larger difference was observed, the architecture with hybrid predictor fetched only 54% of the instructions usually fetched by the one with a perfect predictor.

Table 2.3: Effective fetch bandwidth

Benchmark	Hybrid Predictor	Perfect Predictor
Bzip2	2.60	3.66
Equake	2.70	4.16
Gcc	2.18	3.34
Mesa	2.35	3.30
Parser	2.44	4.53
Vpr	3.02	5.42

Finally, the IPC (Instructions Per Cycle) is shown in Figure 2.12. The dark bar means the performance reached by the architecture with hybrid prediction, while the gray bar shows IPC achieved by the architecture with perfect prediction. In spite of problems with alignment and misses, the architecture with perfect prediction is

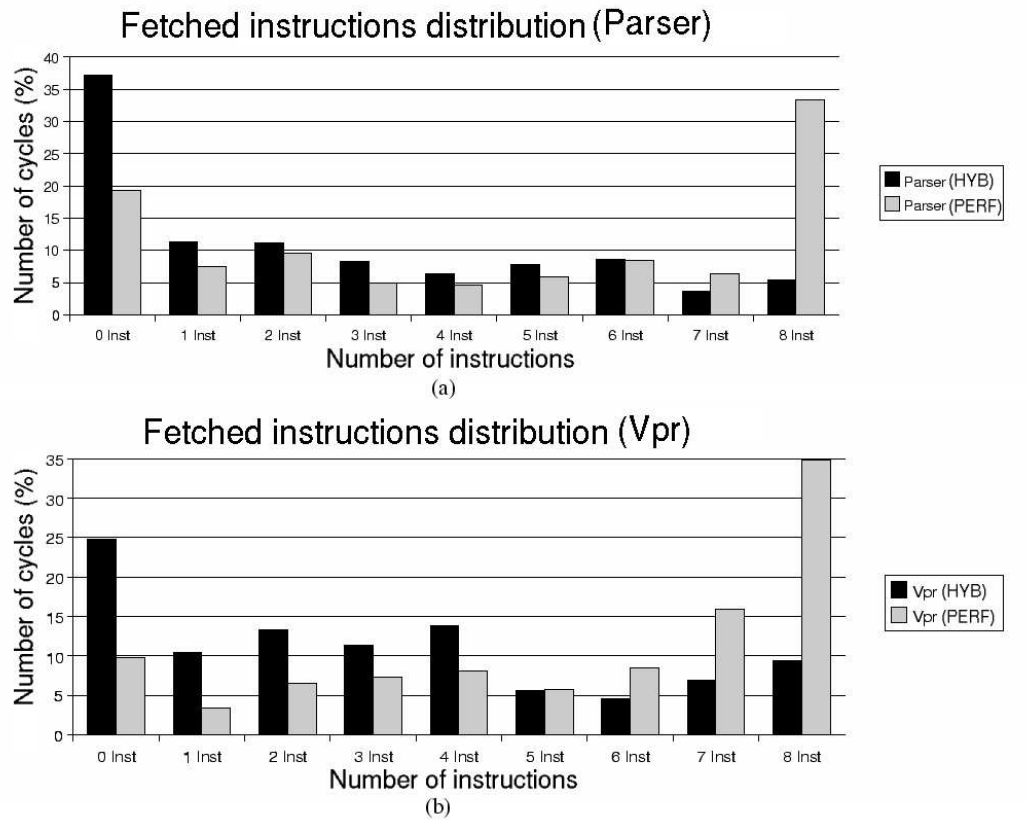

Figure 2.11: Distribution of instructions in benchmarks (a) parser and (b) vpr

better in all cases. In benchmark *parser* the difference between both cases is close to 100%.

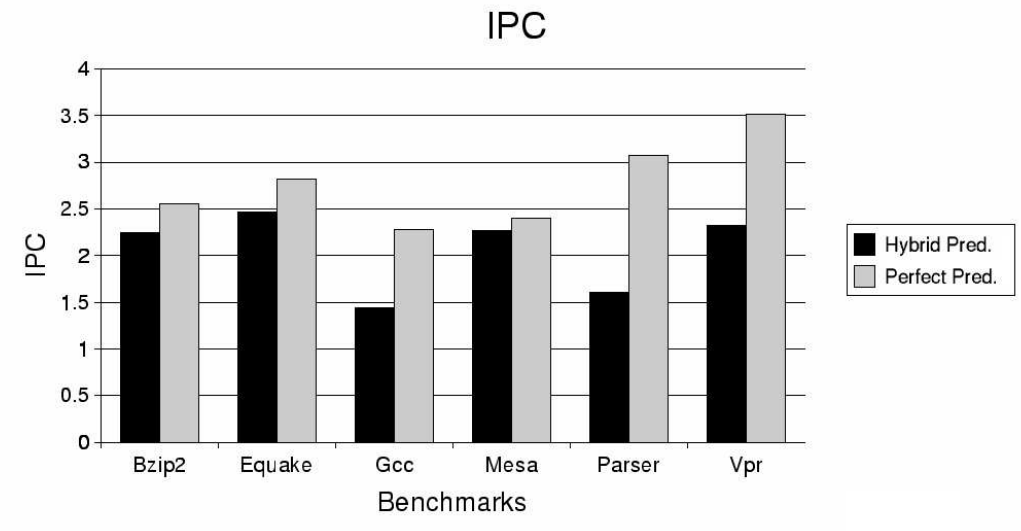

Figure 2.12: Instructions Per Cycle achieved with hybrid and perfect predictions

2.3 Summary

The goal of this Chapter was to show how branches limit performance of modern superscalar architectures.

The simulations presented in this Chapter clearly show how the simple occurrence of branch instructions can harm superscalar architectures performance and prevent the exploitation of the Instruction Level Parallelism (ILP). The misprediction penalties are getting higher with even deeper pipelines and the occurrence of a single misprediction may cause the loss of more than 15 cycles, depending on the architecture.

Mispredictions are the main cause of fetch stalls. In some applications, such as benchmarks *parser* and *vpr*, mispredictions are responsible for almost the totality of fetch stalls. For these benchmarks and also for *gcc*, more than 40% of the cycles were lost due to branch occurrence. This, of course, causes a large loss in performance and, in some cases, there is a two-fold gain in IPC when using a perfect predictor.

In summary, branch occurrence is a real and difficult problem to be solved. Although many efforts have been made over the last 10 years, no definitive solution was found. In the following Chapters of this work a new approach to reduce this problem will be discussed. DCE, or Dynamic Conditional Execution, intends to reduce mispredictions by fetching and executing multiple paths of conditional branches.

3 DCE – DYNAMIC CONDITIONAL EXECUTION

Several studies are focused in the development of new and aggressive multipath mechanisms (KELLER, 1975; NEMIROVISKY, 1990; UHT; SINDAGI, 1995; SKADRON, 1999; SANTOS; NAVAUX, 1998), which intend to improve the performance by increasing the number of executed instructions (AHUJA et al., 1998). All of them, however, need an efficient fetch stage to feed the wide execution engine.

The Dynamic Conditional Execution model is based on the concept that multiple paths may be extracted from few cache lines, in a small number of accesses, and then sent to the execution engine. In other words, multipath execution is performed conditionally and instructions are committed according to the dynamic behavior of the branches.

DCE combines dynamic predication and multipath to reduce the complexity and disruptions of the fetch. This is achieved by fetching sequentially through branches that qualify for predication.

The first step of the dynamic conditional execution is to determine whether the branch has to be predicted or predicated. Basically, a branch has to be predicated if both targets are inside a given distance and if there is no other complex instruction, such as procedure call, inside that structure (*branch-then-else-join*). All other cases, i.e., long structures, structures with calls, etc., are supposed to be normally predicted. The analysis of the code structures is done statically, by the compiler. Then, when the code begins the execution, the architecture will predicate all branches marked at compile time, since it is expected that resources are available to do so. During the predication state of the architecture all instructions are fetched sequentially up to the join point of the paths. After the join point, the architecture returns to behave like a conventional superscalar architecture, until the next branch marked to be predicated is reached.

In order to determine if a branch qualifies for predication, an extension of the selection mechanism proposed in (KLAUSER et al., 1998) was developed. In that selection mechanism, only simple branches qualify for predication. The model used in this work also qualifies complex branches, i.e., nested branches and other structures different from a simple non-nested *if-then-else*. The classes of complex branches allowed for predication are presented later in this Chapter.

The selection mechanism used in DCE is static and runs at compilation time, marking branches that can be predicated according to the target locality. The compiler does not change the original code, it only marks instructions valid for predication (SANTOS, 2003). At execution time, the fetch engine decides whether a selected branch will be predicated or not, based on the availability of resources.

Therefore, DCE is a combination of a static selection mechanism and hardware

support to execute branches eagerly. In DCE, selected branches are treated as regular instructions by the fetch engine and they never disrupt the fetch. As no prediction of control transfer is made at fetch time, they may not cause mispredictions.

The main difference between Klauser et al. (KLAUSER et al., 1998) and DCE (SANTOS, 2003) is that DCE dynamically 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 of the dependent instruction. 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 predicated path. As DCE does not use conditional moves, it does not block the issue of the dependent instructions. In DCE, the replicated instructions can be issued as soon as the appropriate physical registers are ready (i.e. the source data is available).

Resources saturation in DCE is reduced by predicating just part of the branches. The number of predicated branches is acceptable, but if all paths are fetched into the pipeline, several instructions may be replicated. This happens because both paths converge to a single join point and many instances of the same instruction may be created to assure correctness. However, only one of these instances is committed and all others are canceled as soon as the outcome of the branch is known. Those copies are pollution and they must be avoided.

Figure 3.1 shows the example of how DCE fetch stage works. In the upper left portion of the Figure, there is a very simple example in C, while the upper right portion presents the code usually generated by the compiler for that example. The lower part of the Figure shows how this structure is fetched in a conventional architecture and in DCE. It is possible to see that, for a conventional architecture, the fetch will be always executed in two cycles, with two accesses to the same cache line. In DCE, only one access is done and all instructions are fetched at once.

After the fetch stage, the instructions are renamed according to their position in the tree dynamic path of execution. This control is extremely useful to determine which instruction comes from which path, and so which instructions have to be committed later on. The dispatch, issue and execution stages work as conventional superscalar stages, except for load and store instructions that may be predicated. The write back and commit stages are designed to identify and complete only instructions from the correct paths.

3.1 Hammock Classification

As discussed before, DCE reduces the pipeline saturation by predicating only a set of branches executed by a given program. In order to identify the best set for prediction, several applications were profiled according to the branches behavior. This study, described in (SANTOS et al., 2003), pointed a significant occurrence of

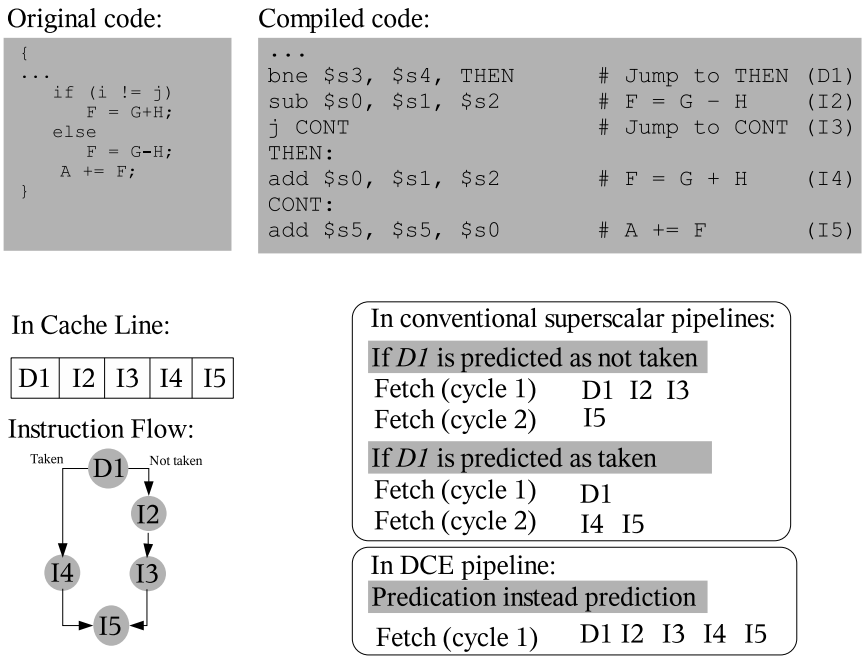

Figure 3.1: Fetch cycle example in DCE

four different classes of nested structures (hammocks), besides the *Simple* class of branches predicated in the work proposed by Klauser et al. The variations of those complex hammocks are the extended model proposed in DCE, presented bellow.

A conditional forward branch that has no nested branches may have one (if-then) or two sides (if-then-else). These branches are shown in Figure 3.2 and are called simple hammocks single sided (*a*) or double sided (*b*), respectively. This was the same subset of branches approached by (KLAUSER et al., 1998).

Conditional forward branches that have other conditional forward branches inside may be classified as follows and are illustrated in Figure 3.3:

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)*

Figures 3.2 and 3.3 present the six different hammock classifications. The diagrams presented are such that 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.

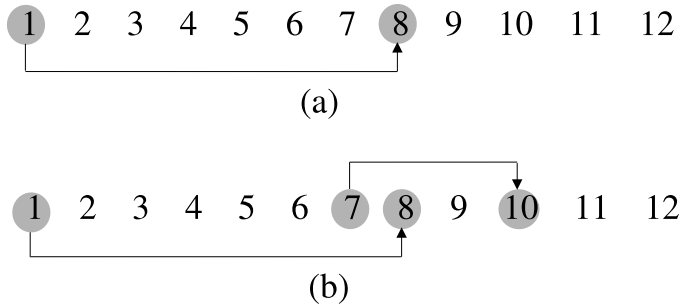

Figure 3.2: Example of simple hammocks

Figure 3.2 (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*.

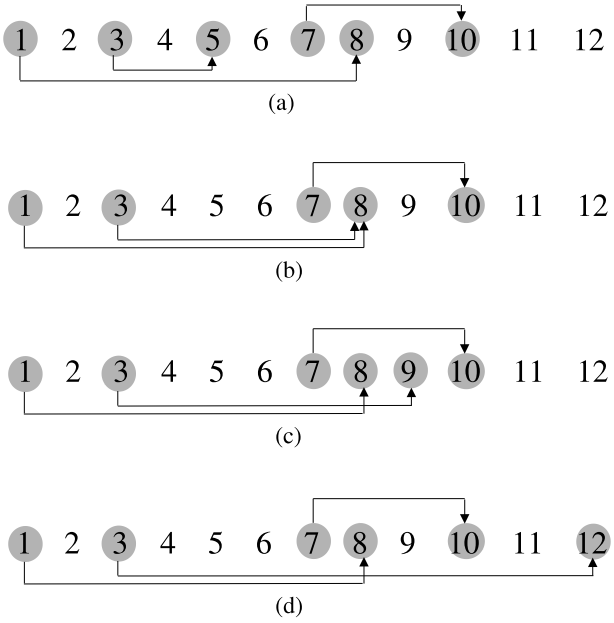

Figure 3.3: Example of complex hammocks

Figure 3.2 (b) shows a two-sided hammock. The example corresponds to an *if-then-else*, which has a condition evaluation in instruction *1* and an 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 this valid category.

If a hammock has any nested hammocks it is then called *Complex*. An example of a *Complex* pattern is shown in Figure 3.3 (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 3.3 (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 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 3.3 (c) and (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*.

Other combinations of nested hammocks are also possible. The combinations showed here are the basic 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 and system calls.

3.2 DCE Pipeline

Although DCE is considered a multipath architecture, its pipeline does not need much additional support to execute. This Section will describe each one of the stages.

Figure 3.4 shows the basic stages in DCE pipeline. Although only eight stages are presented, several others may be introduced at simulation time in order to emulate deep pipelines. Stages *Ren 1* and *Ren 2* are responsible for register renaming. In

DCE this is done in two steps, as renaming is more complex than usual. The other stages, i.e., *fetch, dispatch, issue, execute, write back* and *commit* are similar to the ones found in regular superscalar architectures, with slight differences. Each stage is presented in the following Sections.

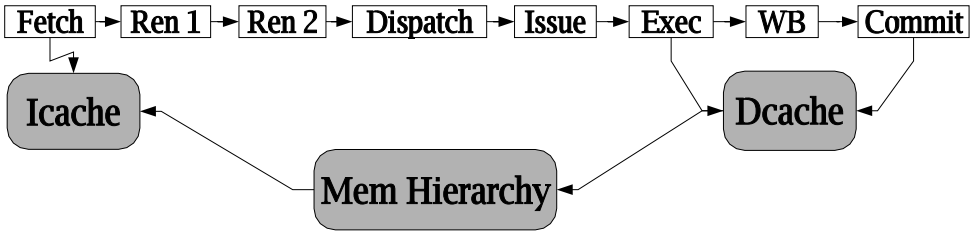

Figure 3.4: DCE pipeline

3.2.1 Fetch

As in a common superscalar architecture, the fetch stage is responsible for bringing instructions from the I-cache. Additionally, in DCE, the fetch stage decides whether or not a qualified branch will be predicated.

This stage is implemented as a finite state machine with two states: prediction and predication. When the *fetch* stage is under prediction mode, it works exactly as a common fetch, performing the following tasks:

- Bring in a line from the memory system;
- Predict a conditional branch, if any;
- Determine the predicted address and,
- Redirect the fetch if the branch is predicted as taken.

The prediction is done by a regular branch predictor, such as a hybrid mechanism (MCFARLING, 1993). The idea, however, is to predict only part of the branches, while predicating others, as discussed previously. Thus, the processor keeps on fetching and predicting branches as usual, until a qualified branch is fetched into the pipeline.

At this point, the architecture changes its state, going to predication. In this state the fetch will determine the join point, based on the compiler information, and access the I-cache in order to bring all instructions between the branch and this address. This means that both paths of the branch will be brought into the pipeline and delivered to execution. The idea is simply to eliminate the need to predict this branch.

After the join point is fetched, the architecture automatically returns to the prediction state until a new qualified branch is brought to the pipeline.

One cans condute that no significant logic is added. In fact, the branch predictor may be even simplified, because it is performing fewer predictions.

3.2.2 Register Rename – First Stage

As mentioned before the scheme to rename registers was derived from a previous work, which is described in (CHAVES FILHO et al., 1999). In that work, another multipath architecture was defined and a very similar register renaming scheme implemented.

Register renaming is the most complex stage in DCE architecture and this was the main motivation to have it divided into two separate stages. This higher complexity is mainly due to the replicas generation. Initially, all replicas are created and tagged according to their position in the instruction flow. Then, after that, they are actually renamed.

Thus, this first stage was designed to tag the instructions according to the path they belong and also to produce all replicas necessary to keep correctness among different data chains. This is extremely necessary in order to allow the architecture to keep track of all paths that are executing in the pipeline. At commit stage, the architecture will retire only instructions from the correct path and they are identified by their tags (or *tagids*). So, each path has a single and unique tag and all instructions from that path are tagged with that same id.

Moreover, after the join point several replicas may be created, each one corresponding to a different path. After the join point all instructions are control independent, but many are data dependent still. So, all these data dependent instructions are replicated in order to maintain semantics. There is one replica for each path available, i. e., DCE will replicate five times all data dependent instructions after a branch which originated five paths, for example. Replicas will be generated until the original branch is resolved and the architecture knows the correct sequence of replicas to choose.

3.2.3 Register Rename – Second Stage

The second part of the register renaming is where occurs the actual renaming. Each logical register is mapped to a physical register.

The main difference between DCE register renaming and a conventional renaming stage is that there are several mapping tables, one for each active path. This is done in order to make it easier to identify which registers are being used by which path. This is especially useful when retiring and flushing correct/incorrect paths.

As discussed earlier, as the number of mapping tables limits the maximum number of paths, it also limits the maximum number of replicas, because there will be one replica for each active path.

The register renaming itself is very similar to the original algorithm described in (TOMASULO, 1967), where logical registers are mapped (and re-mapped) to physical registers every time a given register is written.

Moreover, these tables do not store large amounts of data and they do not represent a significant implementation issue.

3.2.4 Dispatch

The dispatch stage in DCE works similarly to a common dispatch stage found in any superscalar machine. It takes renamed instructions from the previous stage and moves them (in order) to the ReOrder Buffer (ROB), allocating a new entry to each instruction dispatched. After this, the instructions are ready to be issued to

the execution.

3.2.5 Issue

As many others superscalar architectures instructions are issued out–of–order and the issue stage is responsible for scheduling the instructions. The idea is to issue instructions as soon as possible, depending on the availability of the operands and functional units.

Instructions with ready operands are delivered to execution, if there are functional units available. As all WAR (Write After Read or false dependencies) and WAW (Write After Write or output dependencies) were solved by register renaming, only RAW (Read After Write) dependencies are to be observed by this stage. So, an operand is ready when there are no other pendent instructions writing on it.

In DCE, as several paths may be in course, issue stage is also responsible to assure correctness among load/store instructions. In this stage, addresses of all load instructions are checked against the addresses of the store instructions not completed. These store instructions may be located in the same path or in any other active path that originated the one being issued. In this case, addresses have to be compared. If addresses match, new values are updated directly in the ld/store queue. The idea here is to propagate store values to all loads that came after them. This process, known as memory disambiguation, is also performed in conventional superscalar architectures. The difference is that in a conventional superscalar only the addresses are tested, while in DCE, addresses and tagids are verified. Hence, just an additional comparison is required by DCE. This comparison is less complex than the one necessary to the addresses and both of them can be executed in parallel.

3.2.6 Execution

The execution stage has specialized functional units and the execution is performed out–of–order. Also, as soon as the results are ready, they are broadcasted to any instruction in issue which needs this result.

3.2.7 Write Back

The write back stage in DCE is also very similar to one in any superscalar architecture. This stage is responsible to mark in ROB all instructions that finished execution and are ready to commit. This stage is fundamental to assure the correct commit order. A given instruction is going to be delivered to commit only when all previous ones were executed and delivered as well. This is especially relevant during the flush due to mispredictions. When a misprediction is detected, all instructions after the mispredicted branch are squashed. This order is guaranteed by the ROB.

3.2.8 Commit

Commit stage in DCE is slightly different from a regular commit. In DCE, this stage is responsible to retire all correct instructions, i. e., all instructions from correct paths, no matter if branches were predicted or predicated.

As in fetch stage, predicted branches are handled as usual. The branch predictor is updated according to the result of the branch and if a misprediction occurs the pipeline is flushed and all instructions squashed. The fetch will be redirected to the correct path, while all buffers and tables are cleaned as well as physical registers are

freed. This will affect all instructions executed speculatively after the mispredicted branch.

A predicated branch may be executed speculatively, as part of a mispredicted path. This hammock is treated as any other structure and is flushed as well. In this case, all paths are going to be flushed, because all of them were speculatively fetched to the pipeline.

However, when a non-speculative predicated branch is ready to commit, the pipeline follows a different approach. In this case, the pipeline has to squash all incorrect paths fetched and executed before. This is done using the *tagid* of each path as a reference. Basically, all instructions originated by the correct path are marked to be committed and all others are marked as invalid. The commit stage has to broadcast this result to avoid that instructions from incorrect paths get issued and executed. As soon as the branch is resolved, only the correct path is maintained in the pipeline. All other instructions are invalidated.

Finally, this stage is also responsible to free all physical registers from the committed instructions.

3.3 DCE Optimizations – the CIDI Approach

As many replicas are created in order to maintain correctness, a large number of new instructions is created. In some cases, these instructions are exactly the same and there is no reason to keep all of them in the pipeline. Hence, DCE architecture features a mechanism to detect and invalidate such instructions. This mechanism is called CIDI (Control Independent, Data Dependent) and it is very useful to handle at least some of the pollution produced by DCE replicas.

Figure 3.5 shows an example of code with CIDI instructions.

```
if (a == b) {// I1
   c = a + b;// I2
   d = a - b;// I3
}
else {
   c = a;    // I4
   d = b;    // I5
   e = a + c;// I6
}
a = b;       // I7
b = c + d;   // I8
```

Figure 3.5: Code example with CIDI and non–CIDI instructions

It is possible to see that instructions *I2, I3, I4, I5* and *I6* depend on the result of the instruction *I1*. Instructions *I1* and *I2* are going to be executed just in case the branch *I1* is not taken. Similarly, instruction *I4, I5* and *I6* are going to be executed when the branch *I1* is taken. Thus, the correspondent registers for variables c and d are going to be assigned with different values, depending on the branch outcome. Moreover, the correspondent register of variable e will not be assigned, if the branch is not taken. These instructions are, then, known as control dependent, i.e., they directly depend on a given branch.

Instructions *I7* and *I8* are not control dependent because they are going to be executed anyway, i.e., it does not matter if the branch is taken or not. Instruction *I8*, however, depends on the data which is being generated differently on each path of the branch. This instruction is control independent, but data dependent as it belongs to both data chains. On the other hand, instruction *I7* is said to be control independent and data independent, as it is going to be executed either way and it does not depend on data produced by instructions in one of the branch paths.

Control Independent Data Independent instructions do not need to be executed eagerly by DCE, as their correctness are not a concern. DCE architecture detects these instructions and invalidates them during execution. This reduces some of the pollution, saving resources for non-cidi instructions.

4 DCE LIMITATIONS

DCE effectively decreased the misprediction occurrence but this was not enough to improve performance significantly (SANTOS, 2003). The main problem is the side effects introduced by the predication mechanism.

In this Chapter, the DCE limitations are studied.

4.1 Simulation Environment

The results presented in this Chapter were achieved by simulations using the *sim-dce* simulator. *Sim-dce*, a *sim-outorder* based simulator, is fully described in (SANTOS, 2003). This simulator implements the pipeline described in the previous Chapter, including register renaming and a configurable number of pipeline stages. In fact the later aspect was the only additional feature added to the *sim-dce* implementation in order to run the simulations shown in this Chapter. In the original DCE simulator the number of stages was restricted to the seven basic ones. In order to simulate deep pipelines, additional cycles for misfetches and mispredictions could be introduced. In this new version, however, this is not necessary, because there are actually *virtual stages* (delays) in between the stages. These delays may be introduced in two different points of the pipeline: between the register rename (stage two) and the dispatch, and between the write back and the commit. In all the simulations five cycles were introduced between the write back and the commit and three additional ones between the register rename and the dispatch. This means that a 15–stages deep pipeline was used in all simulations.

This Chapter presents two sets of experiments. The first one, showed in Table 4.1, considers an aggressive wide-issue architecture. These configurations are non–practical and are not to be compared to a state–of–the–art commercial microprocessor organization. The idea in using such configurations is two verify the results expected for the next microprocessor generations. The second set is described in Table 4.2 and it is based on current superscalar architectures.

The benchmarks used in these experiments were: *cc1, ijpeg, go, perl* and *m88ksim*, from SPECint95. These benchmarks were a subset of the ones used in (SANTOS, 2003).

In these simulations 600 million of instructions were executed, but the samples for statistics were counted only after 300 million of instructions executed. Table 4.3 shows the benchmarks and the inputs simulated.

In order to optimize the DCE hardware, it is necessary to better understand its hazards. The goal of these simulations is to identify the points where the DCE architecture shows performance improvement as well as its bottleneck(s).

Table 4.1: Configurations used in original DCE experiments (large architecture)

Parameter	Configuration
Fetch width	16, 32 and 64 instructions
Decode width	16, 32 and 64 instructions
Issue width	16, 32 and 64 instructions
Load/Store queue	128 instructions
Instruction window	512 instructions
Integer FUs	32 FUs
Integer Mult/Div	32 FUs
FP FUs	32 FUs
FP Mult/Div	32 FUs
Memory bus width	512 bytes; 32 cycles first chunk
L1 I-Cache	512 Kbytes; 1 cycle hit latency
L1 D-Cache	64 Kbytes; 1 cycle hit latency
L2 unified cache	16 Mbytes; 1 cycle hit latency
Branch prediction scheme	2 level adaptative with 2048 history registers and 14 bits history; BTB with 512 sets, 4-way associative
Return address stack	128 entries
Renaming tables	128, 256, 512, 1024 tables
Maximum branch size to predicate	64 instructions
Classes of predicated hammocks	simple and complex (all)

4.2 The Effect of Replicas in DCE Performance

4.2.1 Effects in a Wide–issue Superscalar Architecture

As stated before, DCE main goal is to increase performance by reducing the mispredictions. Thus, the first aspect to be studied in this Chapter is the misprediction rates.

Figure 4.1 shows the misprediction reduction for the different benchmarks in DCE. The horizontal axis presents the different configurations simulated varying the number of mapping tables (M) and the architecture width (W). The number of mapping tables determines the maximum number of active paths supported and the width determines the number of instructions that can be renamed, issued, executed and committed per cycle. Each line in the Figure means a different benchmark simulated. All percentages were calculated over the baseline architecture, configured using the same parameters showed in Table 4.1, but with no predication allowed.

The Figure shows that there is really an effective misprediction reduction especially in benchmarks *cc1, go* and *ijpeg*. In benchmark *go*, for example, the misprediction reduction may achieve 23% in a 64–wide pipeline with 1024 mapping tables. Benchmark *cc1*, which is well known by its hard–to–predict branches, can achieve around 8% of misprediction decrease. Benchmarks *m88ksim* and *perl*, however, have shown a low misprediction reduction. In these benchmarks, the number of predicated branches was up to 45% lower than in *cc1*, for example. When a larger number of branches are predicating, there is also a larger chance to reduce misprediction.

It is also seen that there is no significant change when the number of mapping tables is increased. This means that the architecture reached its maximum number

Table 4.2: Configurations used in original DCE experiments (small architecture)

Parameter	Configuration
Fetch width	4 and 8 instructions
Decode width	4 and 8 instructions
Issue width	4 and 8 instructions
Load/Store queue	64 instructions
Instruction window	128 instructions
Integer FUs	2 FUs
Integer Mult/Div	1 FUs
FP FUs	2 FUs
FP Mult/Div	1 FUs
Memory bus width	16 bytes; 100 cycles first chunk
L1 I-Cache	32 Kbytes; 1 cycle hit latency
L1 D-Cache	32 kbytes; 1 cycle hit latency
L2 unified cache	512 Kbytes; 5 cycles hit latency
Branch prediction scheme	Hybrid with 2048 entries meta-table and 2 level adaptative gshare xor with 13 bits history and BTB with 512 sets, 4-way associative
Return address stack	128 entries
Renaming tables	4, 8, 16, 32 and 64 tables
Branch size to predicate	8 instructions
Classes of predicated hammocks	simple and complex (all)

Table 4.3: Benchmarks inputs used in the simulations

Benchmark	Input
cc1 (Int)	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O cp-decl.i -o cp-decl.s
go (Int)	50 21 9stone21.in
ijpeg (Int)	-image_file 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.findoptcomp
m88ksim (Int)	-c < ctl.raw
perl (Int)	primes.pl < primes.in

of spawned paths before *128*.

Figure 4.2 shows the speedup achieved by DCE over the reference architecture. The horizontal axis of the Figure presents the different configurations simulated varying the number of mapping tables and the architecture width. The vertical axis presents the percentage of gain/loss in DCE speedup over the baseline architecture. Each line in the Figure means a different benchmark simulated.

As expected, benchmarks *go* and *ijpeg* achieved the best results. In a 64–wide architecture with 1024 mapping tables, benchmark *go* achieves up to 12% of speedup

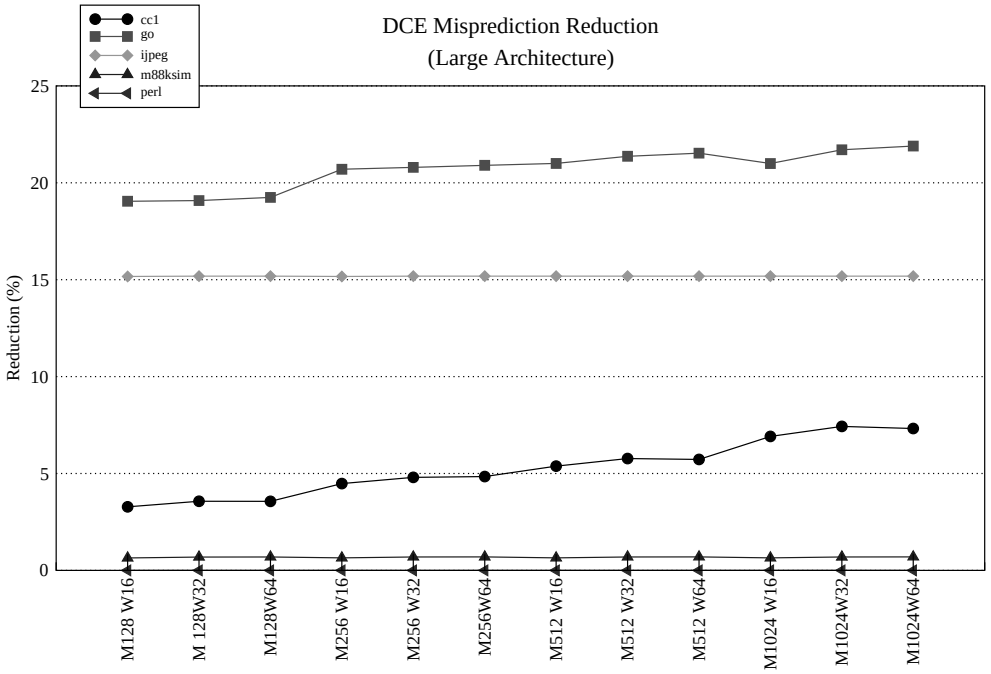

Figure 4.1: DCE misprediction reduction (large architecture)

increase. Benchmark *ijpeg* also presented significant results, achieving around 6% of improvement for all configurations. Benchmark *perl* has negative speedup in some cases, but it also achieves up to 3% of improvement, depending on the architecture width. Benchmarks *cc1* and *m88ksim* did not present results similar to the other benchmarks. In fact, for these benchmarks there was no speedup, except for 32 and 64-wide issue in *m88ksim* simulations.

Figures 4.3 and 4.4 shows the overhead produced by the introduction of predication. The overhead is the number of additional instructions executed by the architecture when DCE optimizations are turned on. In both Figures the horizontal axis are the configurations, while the vertical axis shows the percentage of overhead introduced. Each bar means a different benchmark in Figure 4.3, while 4.4 shows a single bar meaning the harmonic mean of all benchmarks for a given configuration.

Unfortunately, it is possible to conclude that the overhead has a more direct impact over speedup than the misprediction reduction. Benchmark *m88ksim* achieves the highest overhead, up to 178% in some cases. This is probably the main reason for its poor performance. These largest overhead rates correspond to larger width architectures, the same ones that *m88ksim* got its smallest speedup performances.

Benchmark *ijpeg* achieved the lowest rates of overhead (around 16%) and a great misprediction decrease (around 15%). This resulted in almost 7% of speedup increase over the baseline. And if no overhead were introduced, the speedup would probably be even better.

In average, even using a mechanism to identify control independent data independent (CIDI) instructions, the overhead introduced grows linearly with the architecture width increase. In the worst cases, achieved in 64–wide pipelines, the overhead can reach more than 70% of the instructions. This means that, in average, there are 70% more instructions being fetched, decoded, renamed, dispatched,

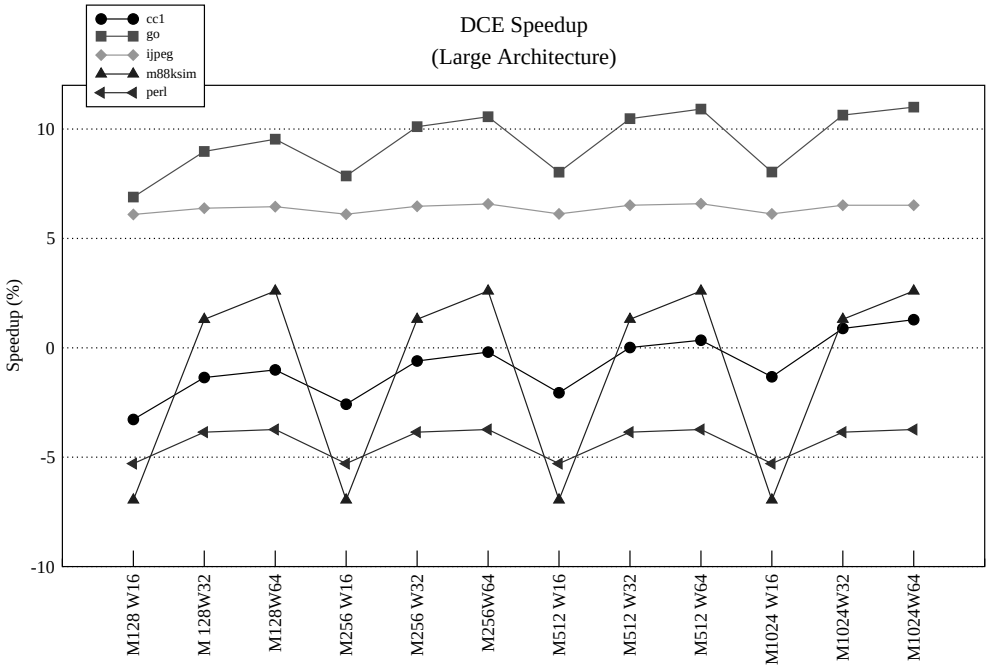

Figure 4.2: DCE speedup (large architecture)

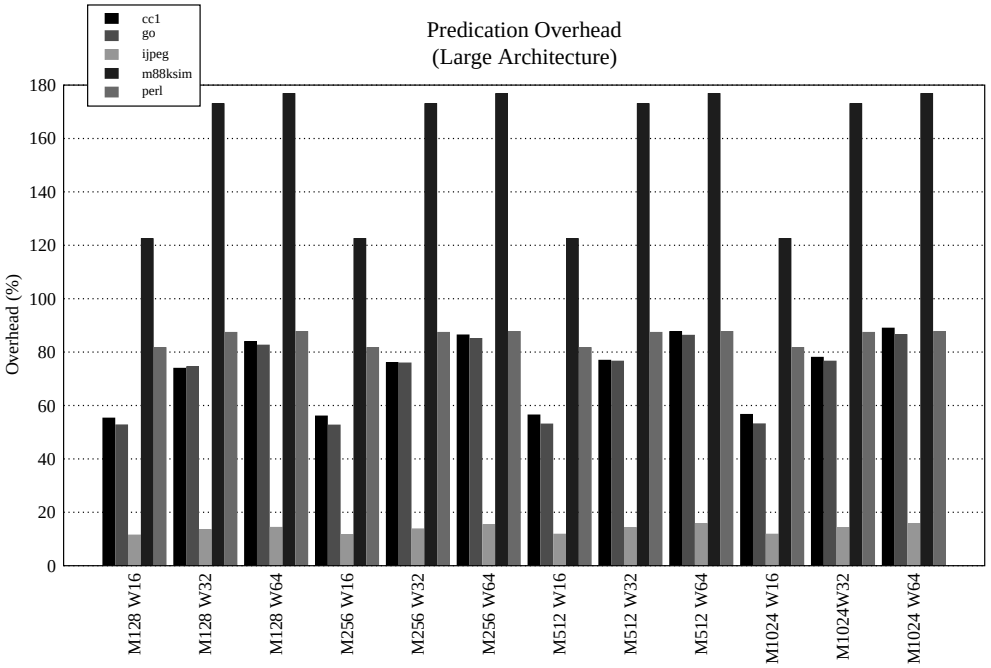

Figure 4.3: Overhead produced (large architecture)

issued, executed and committed in the DCE pipeline. Moreover, this can explain some of the results achieved in terms of DCE speedup.

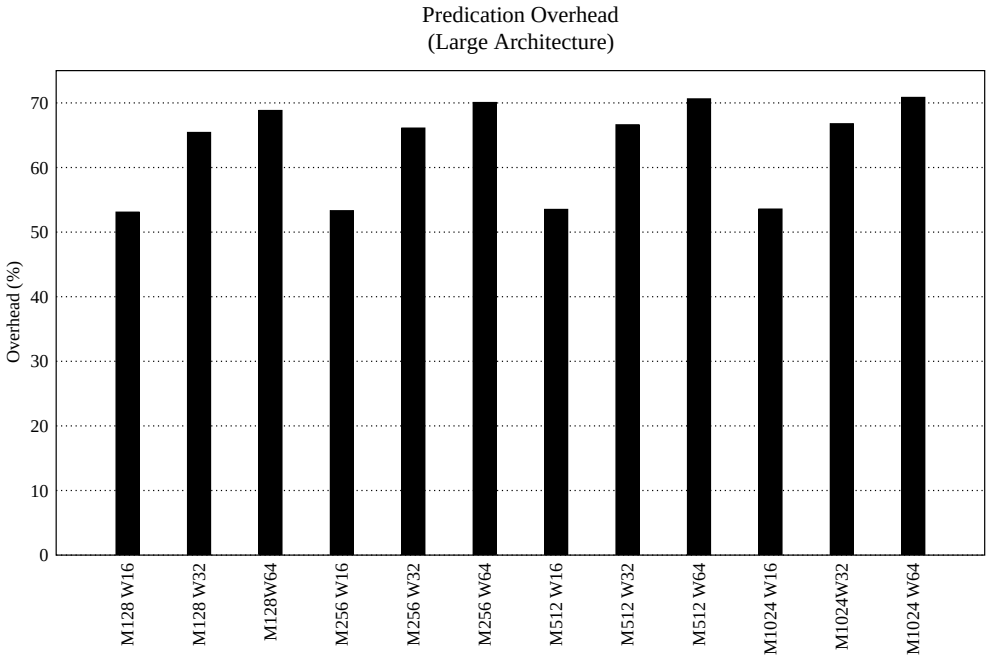

Figure 4.4: Harmonic mean of overhead produced in all benchmarks simulated (large architecture)

4.2.2 Effects in a Small Superscalar Architecture

This Section evaluates the results achieved by architectures configured to be similar to the state–of–the–art microprocessors. The number of mapping tables was also reduced compared to the previously executed experiments. The large number of tables did not produce a significant gain neither in misprediction reduction nor in performance.

The detailed description of the configuration used in this set of experiments is found in Table 4.2. The baseline architectures of this Section were configured using the same parameters showed in this same Table, but no predications were allowed.

Figure 4.5 shows the misprediction reduction for the small architectures. The horizontal/vertical axes are similar to the ones presented in the previous Section. The percentage showed in the vertical axis of the Figure was calculated comparing the misprediction occurrence of the architectures simulated with and without DCE optimizations. A misprediction reduction of 20% means that, using DCE, there were 20% less mispredictions, comparing to the conventional superscalar.

It is possible to see that DCE does not affect any configurations with only four mapping tables. In this case, when there are not enough resources for DCE to spawn new paths, and the architecture dynamically decides not to do so.

Benchmarks *cc1, go* and *ijpeg* have similar behaviors and decrease misprediction significantly and constantly in a range that varies from 7 to 18%. Benchmark *perl* has better results when the architecture width is increased from 4 to 8 instructions. However, the most peculiar result is reached by benchmark *m88ksim*. After increasing the number of mapping tables from 4 to 8, this benchmark reached a misprediction reduction of 100% and maintained this rate for all simulations, except for a 4–wide architecture with 8 mapping tables, where the reduction was around

96%.

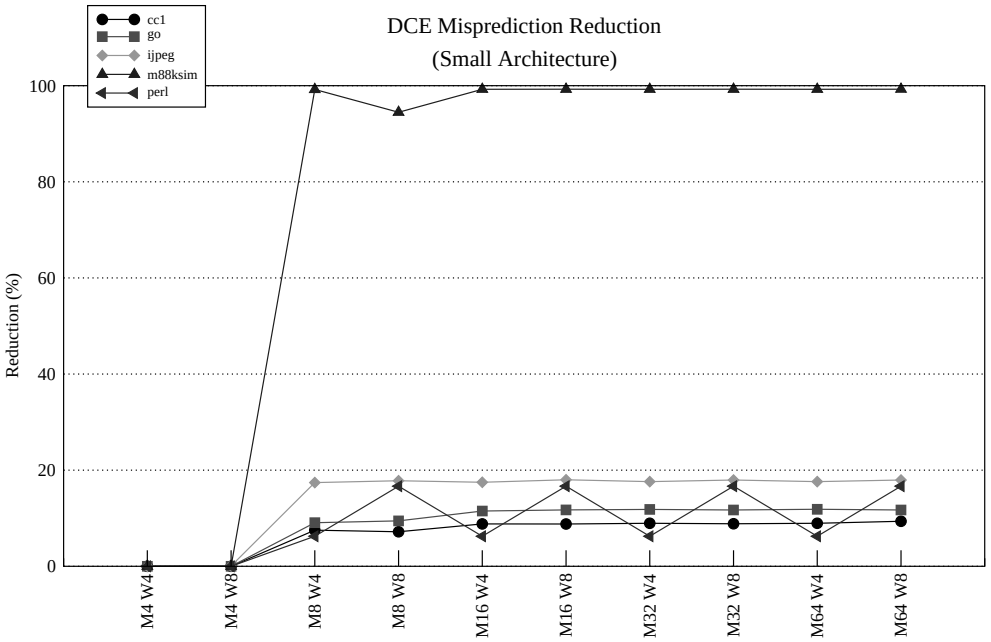

Figure 4.5: DCE misprediction reduction (small architecture)

Figure 4.6 depicts the speedup achieved by DCE in these simulations. Both axes are the same ones presented in the previous Section.

Like in previous Section, the best results were achieved by the benchmarks with largest misprediction reduction. Benchmark *m88ksim*, for example, achieves more than 15% of speedup over the baseline. Nevertheless, benchmarks *cc1* and *go* presented a performance degradation, which achieved up to 3% in *cc1*. In benchmark *go* this degradation was very low less than 1%. This degradations happened even with significant misprediction reduction, as shown in Figure 4.5. Again, this is due to the side effects generated by predicating instructions. Even with some low speedups, these results are better than the ones reached by the wide–issue DCE architectures.

Figure 4.7 shows the overhead produced by the small configurations architecture. The vertical and horizontal axis are again similar to the previous Section.

It is possible to see that benchmark *m88ksim* had a great decrease in the number of executed instructions. This means that the misprediction reduction rate was large enough to overcome the overhead problem and present a smaller number of instructions being executed. The overhead, however, is still significant on benchmark *cc1* and some cases of benchmark *perl*. These are the same benchmarks with poor performance as previously shown in Figure 4.6.

Figure 4.8 shows the harmonic mean of the overhead behavior. Both axis in this Figure are also similar to the ones showed in the last Section.

It is possible to see that, in general, the number of instructions executed were smaller in DCE. In general, for these small configurations, CIDI mechanism managed to reduce the overhead produced by DCE. Also, the misprediction reduction was enough to show some significant speedup increase.

Despite presenting an interesting potential, predicating branches, specially in

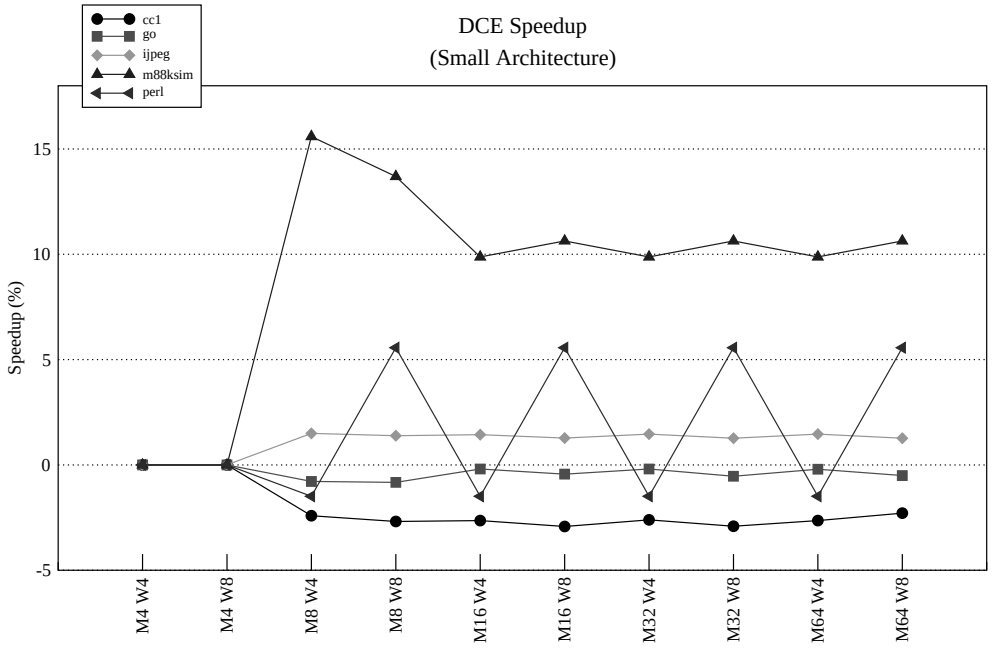

Figure 4.6: DCE speedup (small architecture)

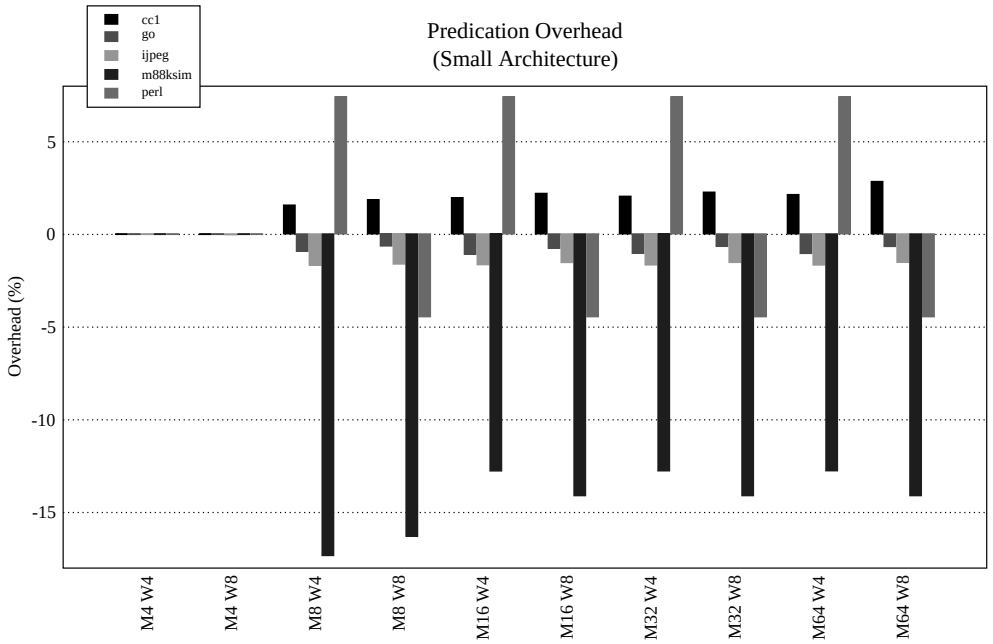

Figure 4.7: Overhead produced (small architecture)

wide–issue architectures, introduces a large overhead which hides the potential benefits of predicating them.

In DCE, the alternative to the pipeline saturation is to predicate just part of the branches. It is possible to see through the simulations results showed in this Chapter that this is not enough, mainly when simulating very wide–issue architectures.

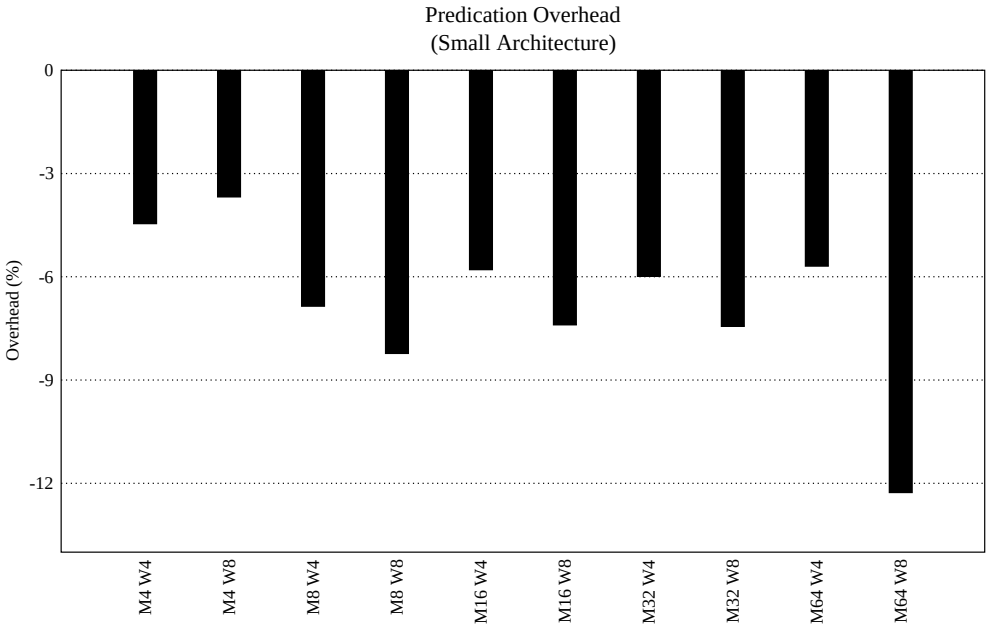

Figure 4.8: Average of overhead produced (small architecture)

If all paths are fetched into the pipeline, several instructions may be replicated. As discussed before, this happens because both paths converge to a single join point and many instances of the same instruction may be created to assure correctness. However, only one of these instances is committed and all others are canceled as soon as the output of the branch is known. Those copies are pollution and they must be avoided. The alternative is to find the join point of the paths and activate it just once.

This task can be performed through mechanisms that explore, for example, control independence (ROTENBERG; SMITH, 1999). Nevertheless, control independent instructions may be data dependent and the problem would not be solved. Each instance of a control independent path belongs to a different data chain.

Value prediction mechanisms could be used, dispatching data with no delays. In DCE, however, this approach has a great disadvantage: value prediction, as branch prediction, is a totally speculative mechanism, which can cause loss of cycles due to mispredictions. DCE architecture was conceived originally to decrease the number of mispredictions through the reduction of the predictor use. In this specific case, branch tables tend to be smaller, leaving more room available to resources for multipath execution.

An alternative to be considered in DCE concept is to reuse the values produced in previous executions. Previous works have shown that, in some cases, more than 50% of the instructions executed are re-executed later. Those experiments were performed in conventional superscalar architectures, with no predication (SODANI, 2000).

Figure 4.9 shows a simple example of redundant replicas in DCE. Assuming that instructions *A1* and *A2* are the join point of the predicated branch *D1* and they read values produced in both paths of *D1*, DCE will introduce replicas *A1'*, *A2'*, *A1"* and *A2"* to satisfy the dependences correctly.

If an instruction at the join point reads a register that is produced logically before *D1*, DCE will not introduce multiple replicas of that instruction since the instruction is control independent and data dependent with relation to the paths of the predicated branch (SANTOS, 2003).

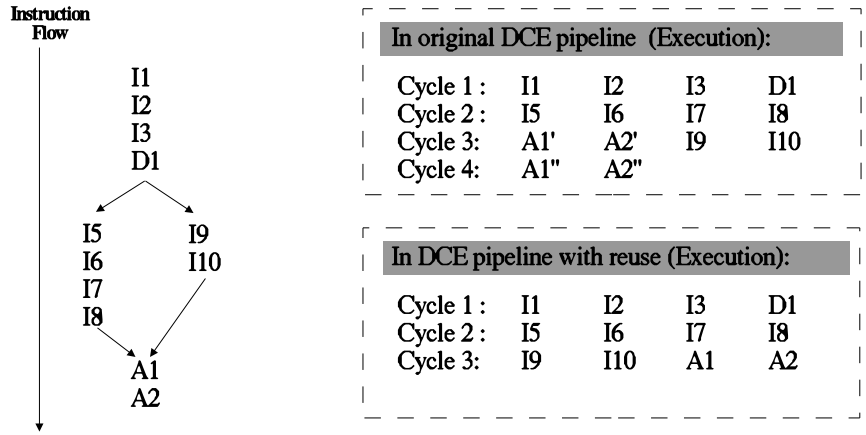

Figure 4.9: Redundant instructions introduced by DCE

Reuse values is a natural way to reduce this cases and the present work intention is to analyze the behavior of such mechanisms in DCE. The next Chapter introduces the topic and discusses how it is done in a Dynamic Conditional Execution architecture.

5 PREVIOUS WORKS

The idea of reuse instructions dynamically is not new and it was proposed for the first time by Sodani and Sohi in (SODANI; SOHI, 1997a), which later originated (SODANI; SOHI, 1997b, 1998) and (SODANI, 2000).

In these works the authors detected that many executed instructions are re–executed and that in some specific cases more than 50% of the instructions are fetched again into the pipeline.

In order to take advantage of this behavior, they proposed the introduction of a new buffer in the microarchitecture. This buffer, called Reuse Buffer or RB, stores previously executed instructions, which may have output values reused later. They evaluated three different organizations for the RB: the *Sv* scheme, the *Sn* scheme and the *Sn+d* scheme, which are explained later.

In all schemes, the instruction's PC indexes the RB. Thus, when a given instruction enters in the pipeline, it has its PC compared to the ones found in the RB. If they are the same, i.e., if this instruction is stored in RB, the source operands are investigated, performing what was called a *reuse test*. If the operands match, the instruction in the pipeline is reused, while it is executed as usual if their operands are different. Instructions are introduced in the RB as they are committed. Instructions with different operands are updated in this stage as well. Each scheme, however, has a different organization regarding to the operands and what kind of fields are to be tested.

The first scheme, called *Sv*, stores directly the value of the operands. When the instruction fetched is found to be the same as the one in RB, their source operands are compared and the output value is reused if this comparison is successful.

The second scheme, called *Sn*, stores the register names, instead their values. The goal of designing such scheme was to simplify the reuse test. Now, the RB does not need to have long fields to store the value of the operands. When an instruction writes in a register, all other entries in RB, which read this register, are invalidated. The authors, however, did not discuss the overhead caused by the search for these instructions. And, although the reuse test is based in smaller comparisons, there is also additional overhead to find the real physical registers mapped to those logical registers marked in the RB. The authors did not approach this problem.

The third and last scheme proposed was the *Sn+d*. This scheme is an extension of the second one and, besides the registers, it establishes and stores the data chains. The idea behind this scheme is to avoid the invalidation in instructions with destination registers updated. This is done by storing all physical registers in an additional table and linking the RB source operands fields to their latest value. When a given instruction writes in a register, all other entries in RB that read this register are

then updated, because now they have that link between physical registers.

The results of this work have shown that scheme *Sv* obtained the best results using larger tables. In a 1024-entries RB the harmonic mean of the reused instructions in all benchmarks simulated achieved 25%, against 12.5% in *Sn* scheme and 20.6% in *Sn+d*.

Although the speedups did not increase in the same rate as the number of reused instructions, they are also significant. For the same number of RB entries, *Sv* scheme reached almost 15%, *Sn* achieved 7.5%, while *Sn+d* achieved more than 10% of performance increase.

In spite of *Sv* scheme has reached the best results, its implementation has to be reviewed carefully. The main concern here is relative to the size of the operand fields as well as the complexity of the comparison in the reuse test.

After Sodani, several researches were started with the idea of reusing instructions. Nevertheless, several of these studies are focused in different levels of the reuse granularity.

Huang and Lilja in (HUANG; LILJA, 1998) have proposed block reuse, which was further studied in (HUANG; CHOI; LILJA, 1999; HUANG; LILJA, 1999, 2000a,b).

In these works, the authors try to reuse not only a single instruction, but also a whole basic block. The RB was replaced by a Block History Buffer (BHB), were previously executed basic blocks are stored. In this case, instructions located in between branches are stored together and reused in their next executions. This scheme is useful especially in a misprediction occurrence. Ideally, the mispredicted path will be stored in BHB and the cost to execute the right path is smaller. The reuse test is performed using the input registers values and execution is skipped if they are the same. The output register values are then saved as if they were also executed. The idea is very similar to instruction reuse, in a different level.

The size of the BHB is an issue, because basic blocks sizes varies depending on the application, as said before. In this study simulations point that 90% of the basic blocks have less than four input registers and five output registers. This means that each entry of the BHB will not have more than ten registers stored to cover 90% of the basic blocks. They also modified their compiler to reach better results and save resources. Their compiler was modified in order to mark registers which are actually part of the basic block, but are not going to be used after its execution, saving register positions in the BHB.

The authors found that the upper bounds of this technique range between 1% and 37%, with average around 15%, depending on the benchmark. They concluded that this kind of technique relies strongly on the value locality of the basic blocks. Further studies also extended this model to speculate values, which are not ready when the reuse test is performed.

Trace reuse is other alternative to reuse not just instructions, but part of the code at once. This was studied by González, Tubella and Molina in (GONZÁLEZ; TUBELLA; MOLINA, 1998; GONZALEZ; TUBELLA; MOLINA, 1999) and also by Costa, França and Chaves Filho in (COSTA; FRANÇA, 1999; COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA; FRANÇA; CHAVES FILHO, 2000). The ideas of those works are similar and they propose to reuse not a single instruction or a basic block, but a trace. Basically, a trace is a sequence of contiguous instructions dynamically extracted from a given program.

The study conducted by González et al. proposed that the RB would be replaced

by the Reuse Trace Memory or RTM. The traces are built on the fly, according to the execution. To start a trace, a reusable instruction has to be detected and the trace will end when the first non-reusable instruction is found after that. Reusable instructions may be detected through input values (or registers) comparisons, just like a single instruction reuse mechanism. The traces to be reused are identified at the fetch stage. After that, all output registers are updated and the fetch is redirected to the next PC after the trace. All these information, i.e., input registers/values, output register/values as well as the next PC after the trace are stored in RTM. If input values are different, RTM is then updated at commit stage.

Typically the trace sizes are in average small and not larger than 8 instructions. The speedup average produced by this mechanism over instruction reuse itself is around 4%, but it can reach as high as 20% in some specific benchmarks. Their study also pointed that larger speedups are achieved in benchmarks which produced larger traces, such as *hydro2d, su2cor, tomcatv* and *ijpeg*.

Costa, França and Chaves Filho proposed a similar mechanism, called the Dynamic Trace Memoization (DTM). They replaced the RB by two separate tables, the Trace Memoization Table and the Global Memoization Table (Memo_Table_G). In this approach, values may be reused in a trace or, alternatively, as single instructions. The traces are built according to the input and output contexts of each trace.

The registers used as source operands for the trace form the input contexts. The reuse test is performed over these registers. Hence, if the values in these registers are the same encountered in the trace being fetched, it can be reused. Moreover, the output context is the set of registers that are produced in the trace and their values may be used by other instructions after the trace. These values are the ones to be reused and updated in the destination registers. Instructions that are part of the trace but do not produce an output context, may be eliminated from the pipeline. Obviously, this cannot be done when reusing single instructions, because destination registers may be used by any subsequent instructions.

The trace is formed in a conservative way and all instructions have to be reused first in order to be included in a trace. When a instruction commits it is stored in the first table, the Global Memoization Table. This table works as a Reuse Buffer and stores single instructions. After this instruction is reused, the trace starts to be built and the last instruction in the trace is the last instruction reused or any other instruction that are not part of the reuse domain. The trace is then stored in the Trace Memoization Table (Memo_Table_T). Thus, next time the first instruction of the trace is fetched, DTM will perform the reuse test over the registers in input context, as discussed before. If the trace cannot be reused, DTM tries to do single instruction reuse.

This research pointed similar results as the ones found in (GONZALEZ; TUBELLA; MOLINA, 1999). The harmonic mean of all benchmarks speedups is around 5% greater than using single instruction reuse. This work later originated (COSTA, 2001).

A later work proposed in (PILLA, 2004) added value prediction to DTM. The main idea of this research is to speculate values for not ready inputs and therefore reuse traces that were not being reused in the original DTM.

Although value prediction is another good alternative to improve performance, it is not the intention to implement such approach is this research. The main problem with these mechanisms is the same observed in any branch prediction. Mispredic-

tions can cause a large number of lost cycles and besides the predictor itself, confidence mechanisms are also necessary in order to avoid that mispredicted values lead to performance decrease. Moreover, DCE main goal is to decrease mispredictions by avoiding predicting.

Sastry, Bodik and Smith also studied an approach with an even more aggressive coarse–grained instruction reuse in (SASTRY; BODIK; SMITH, 2000). In this work, they tried to increase the reuse granularity to the other extreme, reusing entire regions. A region is typically much larger than a trace and according to the authors 55% of all dynamic instructions may be reused using this approach. This work, however, is totally empirical and does not offer simulations results of performance.

Citron and Feitelson in (CITRON; FEITELSON, 2002) revisited the concepts of reusing instructions. They found that instruction reuse does not offer much improvement on overall performance, if fine–grain mechanisms are applied. They compared Sodani and Sohi work with two other schemes for single instruction reuse. The authors found that all mechanisms studied, which reuse only one instruction at once, do not achieve more than 1% of improvement in performance, in average. They suggested that previous works performed in the field abstracted very important details for a real implementation. They say, for example, that source operands are ready only at issue stage, which is not considered in previous studies. Thus, instructions could not be reused before this stage. They also discuss that mechanisms which employ such approach reach their best results in FP benchmarks due to the large latency saved by the reused instructions.

This is an interesting work that studied different reuse mechanisms, proposed in late 90's, under new perspectives and with new simulation techniques. Nevertheless, the authors did not consider that source operands may be really ready before issue stage. This is not true in all cases, of course, but if no dependences are observed among instructions, their source registers and values are known just after renaming stage.

Furthermore, the authors did not approach what is maybe the worst problem in reusing single instructions. When a instruction is reused resources and bandwidth from dispatch, issue, execution and writeback are saved. This means that a larger number of instructions maybe fetched into the pipeline and this may be done faster. However, those reused instructions are maintained in the ROB waiting for instructions that come before them to be delivered to commit. Even being completed, the instruction has to wait in the pipeline in order to keep semantic correctness.

Wallace et al. studied instruction recycling in a SMT architecture with multiple paths of execution (WALLACE; TULLSEN; CALDER, 1999). Instruction recycling is yet another technique related to instruction reuse. The idea here is to allow previously executed instruction to be re–inserted in the pipeline without fetching it. The recycled instructions may also be reused as usual, if operands are still the same.

Instruction recycling is specially interesting when using a Simultaneous Multi-Threaded approach or when running programs with a high rate of hard–to–predict branches.

Typically, architectures that fetch and execute several threads and/or paths always squash the wrong ones as soon as they process the outcome of the branch that originated the different paths. In this case, this is not true. A previously executed thread is not squashed right away, but it is just marked as inactive. Hence, when the processor detects a new path that has a similar one inactive, it allows recycling. This

means to eliminate a re–fetch (and a eventual I–cache miss), freeing fetch bandwidth for other instructions and paths. Also, if the new path and the one being recycled have identical operands, the reuse is allowed as well. This means to save resources and dispatch, issue, execution and writeback bandwidth.

The results of this work showed a performance improvement of up to 12%, when comparing to a conventional SMT architecture.

The experiments performed in that work assumed a very wide and aggressive SMT architecture. The authors, however, did not discuss in details the impact of maintaining several old contexts in the resources pool.

As discussed, previous works always proposed value reuse in conventional scalar and/or superscalar pipelines. This work aims to combine value reuse and dynamic predication in order to identify the pros and cons of pursuing such approach.

6 REUSING VALUES IN DCE

There are many advantages of instruction reuse. In DCE model there are three very significant ones (SANTOS et al., 2003):

- As well as DCE, instruction reuse is a non-speculative mechanism, and thus there is no need for complex routines of recovery;
- With the large instruction flow from the branch predication, reuse reduces the utilization of functional units, freeing those resources for non-redundant instructions;
- The granularity of the instruction reuse may be adjusted and it is possible to reuse several instructions (traces) in one single cycle, allowing a whole group of instructions located after a join point to be reused at once.

In DCE, instruction reuse may be speculative in a certain way. Speculation, in this case, does not regard to the output of a reused instruction, but whether this instruction is part of a correct path or not. This happens because many instructions identified as reusable come from different paths that may be flushed later. However, flush and recovery of a reused instruction is performed normally by the architecture, because reused instructions are treated as any other one. And even if the reused instructions are not useful, resources were made available to others and this is really relevant in a multipath architecture.

While the architecture is in the prediction state, i.e., working as a conventional superscalar architecture, the reuse mechanism presents its normal advantages. Among those advantages, there is one which is especially interesting to DCE. Depending on how the reuse mechanism is implemented, it is possible to correct the branch prediction using the reuse table (COSTA; FRANÇA; CHAVES FILHO, 2000). Therefore, when a branch is fetched and it has the same inputs as a previous execution, its output will be available in the reuse table. If the branch prediction does not match the output from the reuse table, it can be updated and the correct path may be fetched immediately. This feature is important in DCE context because DCE intents to reduce the branch prediction support. Now, the use of both predication and reuse may decrease even more the requirements for that support. Thus, the branch prediction may be adjusted again due to the side effects produced by reuse. Also, the execution engine, now overloaded with replicas and other instructions belonging to wrong paths, can be preserved and used mainly by instructions that cannot be reused.

6.1 Instruction Reuse in DCE

As discussed before, DCE is an architecture with support to dynamic predication. Moreover, instruction reuse mechanism is slight different from the ones proposed in previous work.

Figure 6.1 shows a superscalar pipeline with a reuse instruction mechanism.

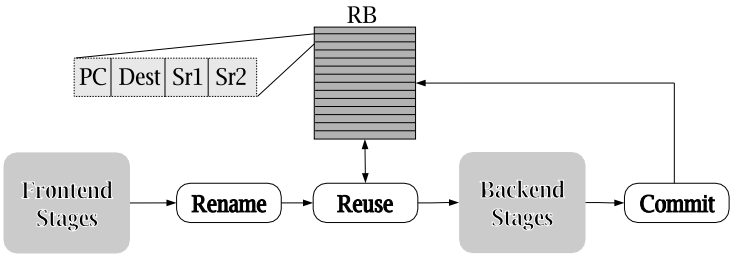

Figure 6.1: Pipeline with an instruction reuse mechanism

Figure 6.2 shows specifically the Reuse Buffer organization used in this work. Each field of the Reuse Buffer (RB) is described bellow:

- PC: address of the instruction. The PC indexes the RB;
- OP1, OP2, OP3: all three store the source operand values in order to make the reuse test. There are three fields due to double precision instructions;
- ADDR1, ADDR2: both fields store the addresses to be compared in the reuse test for ld/st address calculation instructions. There are two fields due to double precision instructions;
- RES1, RES2: both fields store the instructions results. There are two fields due to double precision instructions.

PC	OP1	OP2	OP3	ADDR1	ADDR2	RES1	RES2

Figure 6.2: Reuse Buffer (RB) organization

In DCE, the values of operands are known after register renaming, when the physical registers are mapped. So, at this point a new pipeline stage is introduced. In reuse stage, the source registers are accessed in order to identify whether they are ready or not. If they are ready and their actual input values are stored in one of the RB positions, the output value stored in the RB is reused. This means that the output value is written into the physical destination register and it is marked as ready. All subsequent instructions may read this register normally, as if it was produced during execution. The reused instruction is then inserted in the Re-Order Buffer.

The RB is updated according to the execution at the commit stage. If a given instruction was not reused because the inputs did not match, the RB is updated with this new data according to a replacement policy, just like a small cache memory. In DCE, the least recently used way is replaced, when all other positions are already occupied.

The most peculiar point of DCE reuse mechanism is that the reuse of branch instructions is not possible. Normally, instruction reuse would be used to detect mispredictions earlier in the pipeline, before execution. This is possible because the branch output may be stored in the Reuse Buffer (RB) and if the target fetched is different from the one in the RB, a misprediction is observed. The pipeline can flush the instructions and redirect the fetch. This means that a branch, which would be regularly mispredicted, is misfetched only.

In DCE this is not done. The problem is when a given branch is a data dependent instruction from a previously predicated branch. In this case, it has replicas and each replica has its own path. When the architecture detects that a branch is a misfetched one, it has to flush and redirect the fetch selectively, according to each path. This would be hard and very expensive. Thus, a design decision was to not reuse branch instructions, for simplicity sake.

Figure 6.3 shows an example of this problem. Instruction *D1* is a predicated branch, which produced replicas *I6', I6", D2'* and *D2"* due to data dependences. Instructions *D2'* and *D2"* are branches, each one with its own taken and not taken paths. After their prediction, each one has a flow to follow. If *D2"* is reused and detected as misfetched, it would be complex to flush only the instructions on its path. The pipeline would have to determine which instructions are from which path and squash only the right ones. This would generate several additional comparisons, especially if a large number of paths is in flight.

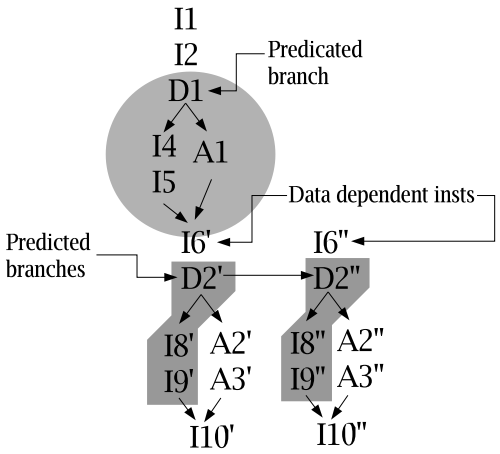

Figure 6.3: Data dependent branches

As well as branch instructions, system calls are not reused. Memory access reuse is handled in a different way, described in the following Sections. Address calculation instructions for load/store are reused as any other common instruction. Also, instructions that are part of a trace being reused or invalid instructions cannot be introduced in RB. This issue will be addressed in Section 6.2.

6.1.1 Memory Access Reuse

Reusing a memory access instruction is not a trivial task. First of all, it is clear that store instructions cannot be reused. These instructions update the memory and therefore must be processed as usual.

Load instructions, however, may be reused under special circumstances. The problem in reusing load instructions is how the architecture can determine whether or not any store instruction has written in this address since last update. A simple alternative to that is to invalidate all loads to the address in the buffer. Unfortunately, this would cause a great overhead, since the entire buffer has to be scanned and invalidated if necessary.

In order to make this practical, there are studies specialized in reuse such instructions (YANG; GUPTA, 2000; ONDER; GUPTA, 2001; BODIK; GUPTA; SOFFA, 1999). Nevertheless, it is not the goal of this work to implement such mechanisms. The intention here is to simplify and make them as simple as possible in order to reuse at least part of the load instructions, which would be reused if one of those sophisticated mechanisms were being applied.

As discussed in Chapter 3, in DCE architecture all loads are checked to verify if there is any store instruction to that address. This is already necessary in the original DCE to assure semantic among paths.

Hence, if there is a pendent store, it is not possible to reuse because a new value is ready to update the value of that given address. The load instruction is then processed as usual, accessing the data cache memory or TLB.

On the other hand, i. e., if there are no store instructions pendent, the Load Reuse Buffer (LRB) is queried and the value stored is reused if a match is found. The Load Reuse Buffer is indexed by the load address and holds its previous occurrence. This Buffer is also updated in commit stage, but stores instructions trigger this action.

After detect that the load instruction may be reused, the architecture copies the value stored in LRB to the destination register, marking it as ready and allowing other dependent instructions to be processed. The instruction is then marked as completed and waits for commit.

Table 6.1 summarizes the modifications in DCE pipeline stages in order to allow instruction reuse. It is possible to see that, besides the new stage introduced, only the issue and commit stages were modified. When memory access reuse is not in use, the issue stage has no changes comparing to DCE original pipeline.

Table 6.1: Summary of pipeline modifications to support instruction reuse

Stage	Modifications
Fetch	No changes
Rename (stage 1)	No changes
Rename (stage 2)	No changes
Reuse	New stage – performs reuse test, updates registers with reused values, forwards reused instructions to ROB
Dispatch	No changes
Issue	Besides issuing all not–reused instructions, memory accesses which can be reused are detected and reused
Execution	No changes
Write Back	No changes
Commit	Besides committing instructions, it updates Reuse Buffer (RB)

6.2 Trace Reuse in DCE

Previous works pointed that an increase in the granularity of the reuse mechanism can also increase its performance (CITRON; FEITELSON, 2002). The problem with reusing single instructions is that such instructions are maintained in the ROB, as any other regular instruction. The instruction is ready to completed just after rename stage, but it is not allowed to commit before all the previous instructions have been retired as well. Execution is out of order, but commit is an in order task.

Reusing a trace means to reuse several instructions at once and, in general, the larger the trace, the larger the number of released resources. In DCE, as discussed before, it is especially relevant to do so. The trace of replicas created by the architecture after a join point makes this approach even more attractive. Moreover, several traces from different predicated paths, reducing the effect of fetching several instruction flows.

The largest advantage, however, is that only instructions which produce values that may be used by others after the trace (output context) are stored in the ROB.

6.2.1 Building and Storing a Trace

Traces are built in the commit stage, as each instruction is being retired from the pipeline. In a conventional architecture this stage is less complex because only one path is in course. In DCE, several paths may be in the pipeline and this task more difficult to be performed. In fact, one may observe that DCE traces grow not just in the vertical, but also in the horizontal due to the replicas introduced. This means that besides the regular flow (vertical), there are several instances from the same instruction (horizontal) as well. As consequence, building a trace may be harder than the usual.

Before start to discuss how a trace is built it is necessary to understand the concept of output and input contexts.

The input context of a trace is formed by all the source registers, which are not being produced in that same trace. The output context of a trace is formed by all destination registers, which holds values that may be used by subsequent instructions.

Figure 6.4 shows an example of a trace. This example was extracted from *cc1* benchmark to illustrate these concepts. This Figure shows the PC of the trace, which is inherited from the first instruction; the next PC after the trace; the input context size; the output context size; the trace size; the values of the input/output scopes; as well as all instructions that are part of the trace.

It is possible to see that only registers *r2* and *r7* are part of the input context. And although *r3* is source register in the third instruction, it is not considered part of the input context. Register *r3* is being produced by the trace itself and it does not depends on any previous instruction. Moreover, registers *r2* and *r3* may be used by instructions that come after the trace and are part of the output context. It is also seen that only the last instruction producing register *r3* is marked. This is because this instruction holds the last value of *r3*, and if subsequent instructions need register *r3*, this is the value they should get. The trace needs only to store this last value. As consequence, instructions that produce intermediary values may be invalidated and retired from the pipeline.

In order to determine the input and output contexts, the architecture uses the

```
(TRACE SCOPE): Tag PC: 0x0056c798 : Next PC : 0x0056c7b0
In size 2 : Out size 2 : Trace size 3
In Scope:
R2: 0x00000039
R7: 0x1002eec0
Out Scope:
R2: 0x00000001
R3: 0x1002eec4

Instructions in Trace:
 0x0056c798 : 0000000000 : sra       r2,r2,5     *
 0x0056c7a0 : 0000000000 : sll       r3,r2,2
 0x0056c7a8 : 0000000000 : addu      r3,r3,r7    *
```

Figure 6.4: Trace example

same algorithm as in (COSTA, 2001). The basic idea is to maintain two bitmaps, one for each context and update them according to the instructions being introduced in the trace. In addition, two other bitmaps holding the values of the correspondent register are required. Each bitmap has 67 positions, which corresponds to all logic registers available in the architecture. When an instruction is being inserted in trace the following actions are taken (in this order):

1. If r is a source register and it is not in output context, mark its correspondent bit in the input context bitmap. Also update its value in the input–values bitmap;

2. If r is a destination register, mark its correspondent bit in the output context bitmap. Also, update its value in the output–values bitmap.

Figure 6.5 shows how the bitmaps are formed for the Figure 6.4 example. Input and output values bitmaps are not showed for simplicity, but they are updated in conjunction with the input and output registers bitmaps. When instruction *I1* is brought, register *r2* is initially marked as input context and then marked as output. The same thing happens on instruction *I2*. In this case, *r2* is already part of the input context and only its value is updated. On instruction *I3*, *r7* is then marked as input context and *r2* value is updated. In *I3* case, register *r3* is not marked as an input register, because it is part of the output context already, from *I2*.

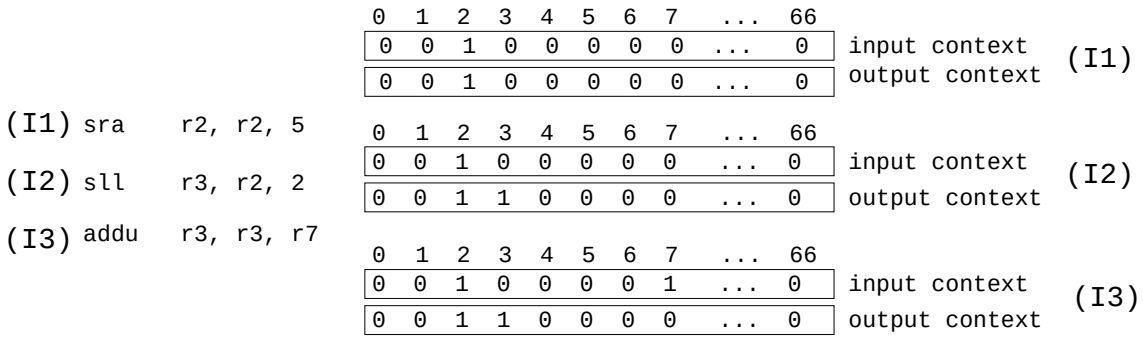

Figure 6.5: Input and output context formation

As in a conventional architecture, it is necessary to keep a temporary buffer where the trace being built is stored. The traces are permanently stored in the Trace Buffer (TB) just after the last instruction in the trace is detected.

There are several heuristics to determine when start to build a trace. Costa in (COSTA, 2001) decided to build traces only from previously reused instructions. This work will implement a more aggressive approach, where all instructions part of the reuse domain are candidates to be part of a new trace. For instance, the reuse domain considered for this work consists in all instructions part of the target ISA, except for load/store and branch instructions. Also, traces are not formed with previously invalidated instructions as well as instructions which are part of another trace in course.

Therefore, the last instruction in the trace will be the last one belonging to the trace reuse domain, i.e., the trace will stop if any of the following happens:

- A load/store instruction is found;
- A branch instruction is found;
- An invalid instruction is found;
- An instruction part of another trace being reused is found.

The trace reuse domain is very similar to the instruction reuse domain, as load/stores and branches cannot be part of a trace.

An invalid instruction is an instruction from a previously predicated hammock, which was found to be part of the wrong path and was selectively squashed. As stated above, those instructions are not considered as candidates to be part of a trace.

An instruction part of another trace in course cannot be included in another trace because its source operands may not be up to date. This is possible because instructions, which are in the pipeline and are being reused by a trace, are part of the output context. If the source operands are also being produced in the trace by other instruction not in the output context, they may not be correct because that previous instruction was not executed and those registers were not updated. Therefore, although the value in the destination register of the output context instruction is right, its source operands may be wrong. In Figure 6.5, *I3* may be a good example for what may happen. Instruction *I2* is going to be invalidated, because it is not producing the last value of the output context *r3*. Hence, instruction *I3* may have an old value for its source operand *r3*, even carrying the right value in its destination register. Thus, if this instruction is stored either in RB or TB as it is, the reuse test will be performed over wrong input values, and a wrong output value may be reused.

Even using the same algorithm to identify input and output contexts, the process to build a trace cannot be done as in a conventional architecture, as said before. In DCE, several instruction flows are available to build the traces. If only one path is active the trace is formed conventionally, i.e., after find all input/output contexts and reach the last instruction in reuse domain the trace is stored in TB. If several paths are in course, several traces are going to be formed. And each trace corresponds to a different path.

As discussed in Chapter 3, the first renaming stage is responsible to create the necessary number of replicas for each instruction. In DCE, all replicas are located just after the original instruction and they are all distinguished by their tagids. Regular traces (vertical traces) are built when two or more instructions with contiguous

PCs and same tagids are found. Traces from different paths (horizontal traces) are built when two or more instructions with same PC and different tagids are found.

Figure 6.6 shows an example of how traces from different paths are built. First and second instructions have the same PC, located in the left part of the Figure. Nevertheless, they have different tagids, which are located just after the instruction. When this occurs, the architecture sorts the instructions by their tagids, building different instances from the same trace. In Figure 6.6 instructions from different paths are highlighted with different gray scales.

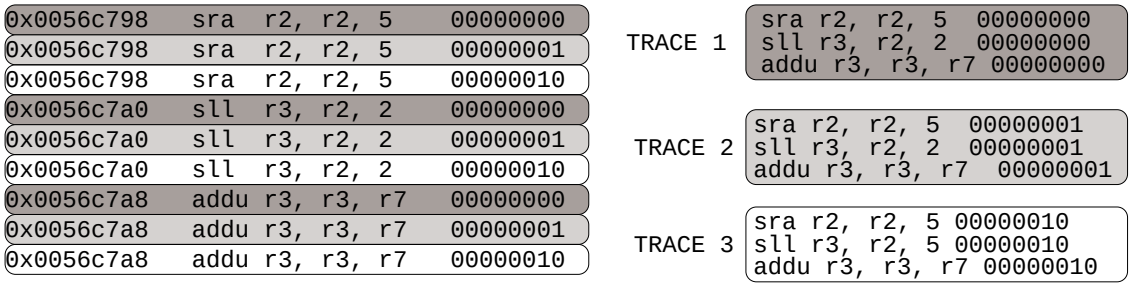

Figure 6.6: Building a trace

After build the traces and store them temporarily, they are really stored in the Trace Buffer. Each different instance created is stored in a TB way. If there are more traces than ways, only the first different ones are going to be stored. The temporary buffer is then cleared and it is ready to start to build another one. The TB structure used in this work is showed in Figure 6.7.

- PC: address of the instruction. As in a Reuse Buffer, the PC indexes the TB;
- Next PC: address of the first instruction after the trace;
- Out PCs: addresses of all instructions in output context;
- Input registers: logical registers part of the input context;
- Output registers: logical registers part of the output context;
- Input values: input register values to perform the reuse test;
- Output values: output register values to update the destination registers when reusing trace.

In previous implementations of trace reuse, the output PCs were not a requirement to be stored in TB. In DCE, however, predicated paths are to be squashed selectively, as it is already known. As a consequence, instructions part of the output context has to be tagged because they may be part of a predicated hammock and may be flushed later. This matter will be again discussed in the next Section.

6.2.2 Reusing a Trace

Reusing a trace in DCE architecture is different, specially because there may be several instances from the same PC in the pipeline. In a conventional architecture, when a trace is ready to be reused, the fetch is redirected to the next PC after the

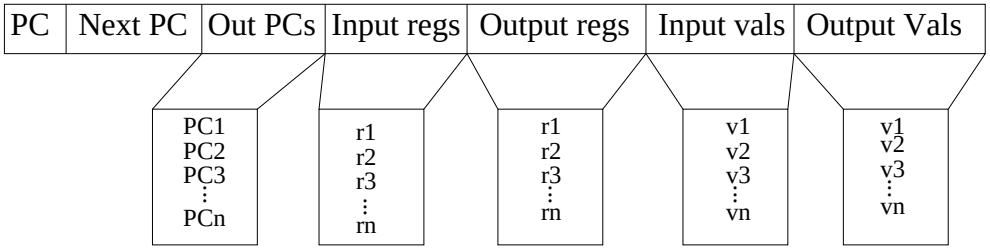

Figure 6.7: Trace Buffer (TB) structure

trace. The next PC of each trace is also stored in TB. All instructions between the beginning and the end of a trace are replaced by entries in the ROB for the output context only.

On the other hand, this cannot be done in DCE, as traces from different predicated paths may be reused. The problem is such as, when several paths are active, there will be more instructions between the beginning and the end of each trace. This is easily seen in the example in Figure 6.6. If the reuse was performed as in a conventional architecture, instructions belonging to other paths (different tagids) were going to be retired from the pipeline as well. And this would probably generate an inconsistent result for the different data chains.

In order to do this, the test for trace reuse is performed yet in the second rename stage. This was modified in the instruction reuse mechanism implemented because the architecture needs to avoid the renaming of registers which are not part of the output context. If the test was left to the reuse stage, an additional logic would be necessary to look for those registers and roll back to the previous status.

When a instruction is fetched, the TB is queried and if a match is found, the reuse test is performed. The reuse test consists in verifying if all input registers from a given path are ready, i.e., if they are mapped to physical registers and with no pending reads. Hence, the input registers of the trace are used to query the mapping table of the same path indicated by the tagid from the instruction in the pipeline.

When the reuse test matches, the trace can be reused. This means that the output values stored in the TB are going to be written to the physical registers mapped by the output context registers, also stored in TB. The mapping table used will be the same one queried during the reuse test, determined by the tagid of the instruction in the pipeline.

At this point all instructions from the trace are still in the pipeline, some tagged and some not tagged yet. Again, the trace reuse has to selectively remove all instructions not producing an output value. This task can be done only after the instructions pass the first renaming stage, where they are actually tagged. The instructions of a trace are commonly not tagged, being tagged in the current or in the next cycles.

For already tagged instructions, the architecture just verifies which ones produce output values. In this case, the instruction is kept, renamed, reused and committed later on according to the value stored in the TB. Instructions tagged but not producing output values are invalidated and removed from the pipeline. This occurs for instructions like the second one, presented in the example on Figure 6.4.

For not tagged instructions there are two basic alternatives. The first one studied was to consider just traces with all instructions already tagged. Nevertheless, this resulted in a very low number of traces reused.

The second alternative was to introduce a small buffer, called the pendent buffer. As previously stated, the most common situation is to find instructions in the trace being tagged in a given cycle or in the next one, at the latest. Thus, it is not necessary a large buffer. Preliminary simulations have shown that ten positions are sufficient to keep all instructions not tagged until they complete the tagging process. Thus, for this work, it was assumed a ten positions buffer.

Figure 6.8 shows the pendent buffer organization. It is possible to see that, only the PC, the tagid and the results values are the required fields. Additionally a bit called *OC* (Output Context) identifies whether the buffered instruction produces an output value or not. Depending on this bit, the instruction is handled differently. If an instruction does not produces an output value then is invalidated, like if it was squashed. On the other hand, if an instruction produces an output value, it will be maintained in the pipeline. This instruction is going to be renamed normally and the results in the pendent buffer will be assigned to the destination physical register.

Figure 6.8: Pendent buffer

Table 6.2 summarizes the modifications in DCE pipeline stages in order to support trace reuse.

Table 6.2: Summary of pipeline modifications to support trace reuse

Stage	Modifications
Fetch	No changes
Rename (stage 1)	Besides tagging, look for instructions part of a trace at the pendent buffer
Rename (stage 2)	Perform the trace reuse test – rename only instructions part of the output context and invalidate others
Reuse	No changes – keep looking for instructions to reuse
Dispatch	No changes
Issue	No changes – besides issuing all not–reused instructions, memory accesses which can be reused are detected and reused
Execution	No changes
Write Back	No changes
Commit	Besides committing instructions, it updates the Trace Reuse Buffer (TRB)

7 SIMULATION ENVIRONMENT

The goal of this Chapter is to present the new simulator developed for this work as well as to establish the simulation strategy that was adopted to obtain the results. The following Sections describe each one of these topics separately.

7.1 The Extended *sim-dce* Simulator

The simulator developed in this work is an extended version of the DCE architecture simulator. As stated before, *sim-dce* (SANTOS, 2003) was developed based on the *sim-outorder* simulator, from the SimpleScalar Tool Set (BURGER; AUSTIN, 1997). As *sim-outorder*, *sim-dce* is a highly detailed MIPS-like simulator which, besides the basic features, implements tagging and renaming of paths and instructions, code evaluation to mark instructions to be predicated as well as the predication itself. The rename stage is very complex and sophisticated with physical registers and a path control scheme. *Sim-dce* simulator was modified in two steps, obeying all the architecture constrains described in the previous Chapter. Moreover, it was first extended to allow instruction reuse only and after that, trace reuse was developed.

The modification to allow instruction reuse consists mainly in the implementation of a buffer, called Reuse Buffer (RB), capable of storing instructions of a given program. Each instruction may be mapped to a specific set/way in the buffer indexed by the PC.

In order to effectively reuse the instructions stored in the Reuse Buffer, the architecture has to lookup, find and ultimately reuse the output values. As discussed previously, this task is performed by a new stage just after register renaming, where the values of the operands are physically determined. After this point the RB is accessed with the instructions PC and the value of the operands are compared to the ones stored in the RB. If the values are the same, the output stored is reused. This means that the output register is assigned with the new value and this instruction is freed from passing in dispatch, issue, operands fetch, execution and write back.

The commit stage was also modified. Besides the retirement of all instructions from the correct paths, this stage is also responsible to forward instructions to the Reuse Buffer. The RB is always updated according to the last execution and LRU policy is used as replacement policy.

The second stage of the development was to implement the trace reuse. Again, the simulator was designed following the guidelines discussed in Chapter 6.

The implementation of a new table, called Trace Buffer (TB) was necessary to store all traces formed. The reuse test, however, is performed earlier in the

```
if (fetched_PC == PC_in_TB)
   for (i=0; i < input_scope_size; i++)
      if (value_in_TB == value_in_bank_of_registers and
          register_is_ready)
            ready_inputs = TRUE
       else
            ready_inputs = FALSE
            break
 if (ready_inputs)
     update least_recently_used_list_in_TB
     for (i=0; i < trace_size; i++)
         if (instructions_in_TB_are_in_rename_buffer)
                 if (register_produces_output_value)
                     register = value_in_TB
                     register_is_ready = TRUE
                else
                    fetched_PC = invalid_instruction
          else
             update pendent_buffer
```

Figure 7.1: Simplified algorithm for trace reuse

pipeline, in parallel with the second stage of the rename. The idea is to prevent the instructions that do not produce output values to get renamed. Moreover, traces from different paths are selectively reused, i.e., one path may be reused, while others may not. This is possible because the horizontal traces formed by each path of replicas are selectively reused.

The development of the extended *sim-dce* required also to implement a second buffer to store instructions which were not tagged yet. This ten entries buffer, called Pendent Buffer, store all non-tagged instructions. This is implemented just after the tagging, in the first renaming stage, allowing the architecture to mark the instructions as reused in trace when they are producing output values or to invalidate the ones that do not produce any useful output value.

Instructions marked as reused in trace are not queried to be reused by the instruction reuse mechanism. Thus, trace reuse is always the first choice. The RB is queried only when the instruction is not part of a trace in course. If an instruction is not part of a trace, the RB buffer is then queried and if a match is found, the instruction is reused by itself. Figure 7.1 presents the simplified algorithm for trace reuse.

The traces are built in the commit stage. A trace is a series of contiguous instructions contained in the reuse domain. In this work, traces are restricted to basic blocks due to the limitations described earlier. Also, each path is stored in a Trace Buffer way and the least recently used one is replaced when new traces are introduced in the TB. Figure 7.2 presents a simplified algorithm used for trace construction, while Figure 7.3 shows the algorithm for store the traces in TB.

Another important aspect, which may affect directly the results achieved by the value reuse mechanisms, is relative to the execution latencies of each instruction

```
if (committed_PC == last_PC + 1) or
  (commited_PC == last_PC and committed_tagid == last_tagid)
     if (first_instruction_on_trace)
        update input_context_table
        update output_context_table
     else
        if (source_register_1 > 0)
           if (not_part_of_output_context)
              update input_context_table
        if (source_register_2 > 0)
            if (not_part_of_coutput_context)
              update input_context_table
        if (destination_register > 0)
              update output_context_table
        update temporary_TB
        if (max_number_of_instructions or
            max_number_of_inputs or
            max_number_of_outputs)
              store_trace
        last_tagid = committed_tagid
        last_PC = committed_PC
  else
     if (trace_in_course)
        store_trace
     else
        first_instruction_on_trace = TRUE
        update temporary_input_output_context
```

Figure 7.2: Simplified algorithm for trace construction

```
for (i = 0; i < TB_associativity; i++)
   if (trace_in_set == trace_being_stored)
      store = FALSE
      break
   else
       store = TRUE
if (store)
    least_recently_used_block_in_TB = trace_being_stored
```

Figure 7.3: Simplified algorithm for trace storage

type. In a reuse event, those cycles are going to be saved, because a reused instruction will not require a FU to execute. Thus, the following latencies were considered in the simulations performed in this work:

- Logic and arithmetic instructions (Integer) – 1 cycle;
- Multiplication (Integer) – 3 cycles;
- Division (Integer) – 20 cycles;
- Logic and arithmetic instructions (Float Point) – 2 cycles;
- Multiplication (Float Point) – 4 cycles;
- Division (Float Point) – 24 cycles;
- Square root (Float Point) – 24 cycles.

7.2 Simulation Strategy

The strategy used in the simulations was to vary the resources in two different ways. Initially, execution resources were configured according to a state-of-the-art superscalar architecture. Basically, DCE resources as well as reuse resources were varied in order to find the best interaction between these two features.

In the first set, DCE resources were limited to 4, 8, 16, 32 and 64 mapping tables. The distance allowed for predication was fixed and equal to 16 instructions (from branch to join point).

The RB and the TB were also varied in all experiments. The goal was to study how the reuse buffer and trace buffer sizes may affect the performance and, above of all, find the best configuration for such resources. The maximum trace size was fixed in 4 instructions. This number was chosen because previous studies have shown that a trace is not larger than 3, in average (COSTA, 2001). Also, even with the expectation of having larger traces due to traces formed after the join point of predicated paths, the average size is not expected to be much larger than 4 instructions, as traces do not include branches. However, further investigation to find the best number for this configuration is still necessary. This discussion is also true for the maximum number of registers allowed in the input/output contexts.

The L2 unified cache as well as the memory bus, branch prediction, instruction window, instruction queue and functional units were not varied and they were configured like a modern superscalar architecture.

The first set of fixed configurations used in the experiments are presented in Table 7.1. As said before, execution resources were limited.

The variable configurations used in the first set of the experiments are also showed in 7.1. As stated before, the number of renaming tables was varied from 4 to 64. The simulations on the previous Chapter pointed that more than 128 tables do not produce a larger number of predications. Hence, the architectures simulated were configured in a range that is believed to be the most significative.

The second set of simulations defines a very aggressive superscalar architectures and the goal was to study the impact of reusing instructions in very wide microprocessors, as the ones approached in (SANTOS, 2003). Table 7.2 shows the fixed and variable configurations used in this second set.

Table 7.1: Configurations used in extended DCE experiments (first set)

Parameter	Configuration
Decode width	4 and 8 instructions
Dispatch width	4 and 8 instructions
Issue width	4 and 8 instructions
Renaming tables	4, 8, 16, 32, 64 tables
Branch size to predicate	16 instructions
Classes of predicated hammocks	simple and complex (all)
DCE optimizations	always on
Instruction fetch queue	16 instructions
L1 Caches	128 sets; 64 bytes line; 4 ways; 1 cycle hit latency
L2 unified cache	256 sets; 256 bytes line; 8 ways; 5 cycle hit latency
Memory bus width	16 bytes; 100 cycles first chunk; 10 cycles remaining chunks
Branch prediction scheme	Hybrid predictor with 2048 meta-table; with BTB with 512 sets, 4–way associative
Return address stack	64 entries
Load/Store queue	64 entries
ROB size	128 entries
Integer FUs	2 FUs
Integer Mult/Div	1 FUs
FP FUs	2 FUs
FP Mult/Div	1 FUs
Reuse Buffer size	1024, 2048 and 4096 entries
Reuse Buffer associativity	2 and 4
Trace Buffer	512, 1024, 2048 entries
Trace Buffer associativity	2 and 4
Max number of instructions in trace	4 instructions
Max number of inputs context regs	4 registers
Max number of output context regs	4 registers
Reuse optimizations	on and off

As the first one, the second set of simulations varied the architecture width as well as the DCE and reuse resources. In this set, however, the goal was to evaluate reuse in future generations of microprocessors.

The benchmarks used in all simulations are shown in Table 7.2. All integer benchmarks previously simulated in Chapter 4 were used, such as *cc1, go, ijpeg, m88ksim, perl.* Moreover, three additional Float Point (FP) benchmarks were simulated. They are:*applu, mgrid, turb3d.* The idea is study the impact of reusing instructions in the presence of float point operations as well. Previous studies (CITRON; FEITELSON, 2002) pointed that reuse mechanisms may be specially attractive to instructions with high latency execution.

In these simulations 600 million of instructions were executed, but the samples for statistics were counted only after 300 million of instructions, avoiding the initialization portion of the benchmarks.

Including all configurations showed in this Chapter, more than 1000 simulations were performed, which means several billions of instructions committed in the new

Table 7.2: Configurations used in extended DCE experiments (second set)

Parameter	Configuration
Decode width	16, 32 and 64 instructions
Dispatch width	16, 32 and 64 instructions
Issue width	4 and 8 instructions
Renaming tables	128, 256, 512, 1024 tables
Branch size to predicate	64 instructions
Classes of predicated hammocks	simple and complex (all)
DCE optimizations	always on
Instruction fetch queue	512 instructions
L1 Caches	256 sets; 512 bytes line; 4 ways; 2 cycle hit latency
L2 unified cache	2048 sets; 1024 bytes line; 8 ways; 4 cycle hit latency
Memory bus width	512 bytes; 32 cycles first chunk; 1 cycles remaining chunks
Branch prediction scheme	2 level adaptative gshare xor with 13 bits history and BTB with 512 sets, 4-way associative
Return address stack	64 entries
Load/Store queue	128 entries
ROB size	512 entries
Integer FUs	32 FUs
Integer Mult/Div	32 FUs
FP FUs	32 FUs
FP Mult/Div	32 FUs
Reuse Buffer size	1024, 2048 and 4096 entries
Reuse Buffer associativity	2 and 4
Trace Buffer	512, 1024, 2048 entries
Trace Buffer associativity	2 and 4
Max number of instructions in trace	4 instructions
Max number of inputs context regs	4 registers
Max number of output context regs	4 registers
Reuse optimizations	on and off

architecture simulations.

In order to display the results in an understandable and significant form, the results had to be carefully analyzed and evaluated. The summary and conclusions of the analysis are presented in the next Chapters of this work.

7.3 Summary

The goal of this Chapter was to present the simulator developed during this PhD Dissertation and also to discuss the strategies used for the simulation experiments and to achieve the results presented in the next Chapter.

Sim-dce (SANTOS, 2003), a *sim-outorder* based simulator (BURGER; AUSTIN, 1997), was further extended to support value reuse. First, only instruction reuse, with one table only was allowed. This first part of the implementation was very similar to the ones fully presented in Sodani's work (SODANI, 2000) except that it was implemented over an architecture with dynamic predication.

As a second step, trace reuse was allowed as well. In order to do this, a second

Table 7.3: Benchmarks inputs used in the simulations

Benchmark	Input
cc1 (Int)	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O cp-decl.i -o cp-decl.s
go (Int)	50 21 9stone21.in
ijpeg (Int)	-image_file 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.findoptcomp
m88ksim (Int)	-c < ctl.raw
perl (Int)	primes.pl < primes.in
applu (FP)	< applu.in
mgrid (FP)	< mgrid.in
turb3d (FP)	< turb3d.in

table was implemented and a scheme to store several traces, from different paths, was completely developed in this work. The architecture of this scheme is described in Section 6.2. of this work. The simplified algorithms used are shown in the first Section of this Chapter.

The simulation strategy used was to simulate two different kinds of architectures. Initially, the hardware configurations used were similar to the ones found in current superscalar architectures. These configurations are referenced as small architectures in the subsequent Chapters. Next, very aggressive architectures were also simulated. The goal of this second set of simulations was to provide an overview of the impact of value reuse for future generations of microprocessors. Also, these simulations were used to find the upper bounds of the architecture with dynamic predication and value reuse. These configurations are later referenced as large architectures in this work.

All configurations were run using deep pipelines configurations, with a fixed number of pipeline stages equal to 18. This is extremely relevant in order to provide a more realistic analysis of the problems caused by control dependences.

Eight benchmarks of the well known and established SPEC CPU were used, with the input set shown in Figure 7.3. From those, five of them are basically composed by integer instructions (*cc1, go, ijpeg, m88ksim* and *perl*), while the remaining three were from the float-point benchmark set (*applu, mgrid* and *turb3d*). 600 millions of instructions were run in each simulation, which means that several billion of instructions were executed and committed when all benchmarks simulations were completed.

8 SIMULATION RESULTS

8.1 Reusing Instructions as an Alternative to DCE overhead

This first Section evaluates the reuse of single instructions only. The goal is to determine whether the reuse of single instructions is effective in DCE. The analysis of the results was focused in the most relevant points where the application of reuse may affect the DCE architecture. These results are shown in the following Sections.

8.1.1 Number of Reused Instructions

Initially, the number of reused instructions was observed. Figure 8.1 shows the harmonic mean of all benchmarks for each configuration simulated with an 8–instructions wide architecture. The vertical axis depicts the percentage of reused instructions. This was calculated over all instructions brought to the pipeline, reused or not. The horizontal axis shows the configurations used in each set of the experiments, like *M4A2* equals to 4 mapping tables and RB associativity equal to 2; *M4A4* equals to 4 mapping tables and RB associativity equal to 4; *M8A2* equals to 8 mapping tables and RB associativity equal to 2; and so on. Each vertical bar presents the harmonic mean of all six benchmarks for that configuration.

It is possible to see that, in all cases, the percentage of reused instructions exceeds 10%. It is also possible to see that the percentage of reused instructions do not change significantly among the configurations, even with higher numbers of mapping tables available. In some specific benchmarks, as *perl*, this rate is extreme high and achieved more than 50% of instructions. This means that, in all those cases no resources, such as FUs, were allocated to execute instructions. The difference between using 1024, 2048 and 4096–entries tables range between 1% and 2% from each other. Also, the largest differences are between configurations using tables with 1024 and 2048 entries.

A similar behavior is also observed in configurations using a 4-wide architecture. The percentage of reused instructions, however, is 2 to 5% smaller than the ones using a 8–wide architecture. Thus, for the next Sections, only configurations with 8–instructions–wide architectures were used to calculate the statistics. This means that for reuse, increasing the architecture width may bring more improvements than increasing the number of mapping tables.

8.1.2 Overhead Reduction

Figure 8.2 depicts the reduction in the overhead produced by DCE. The horizontal axis shows the configurations used in the simulations. The vertical axis presents

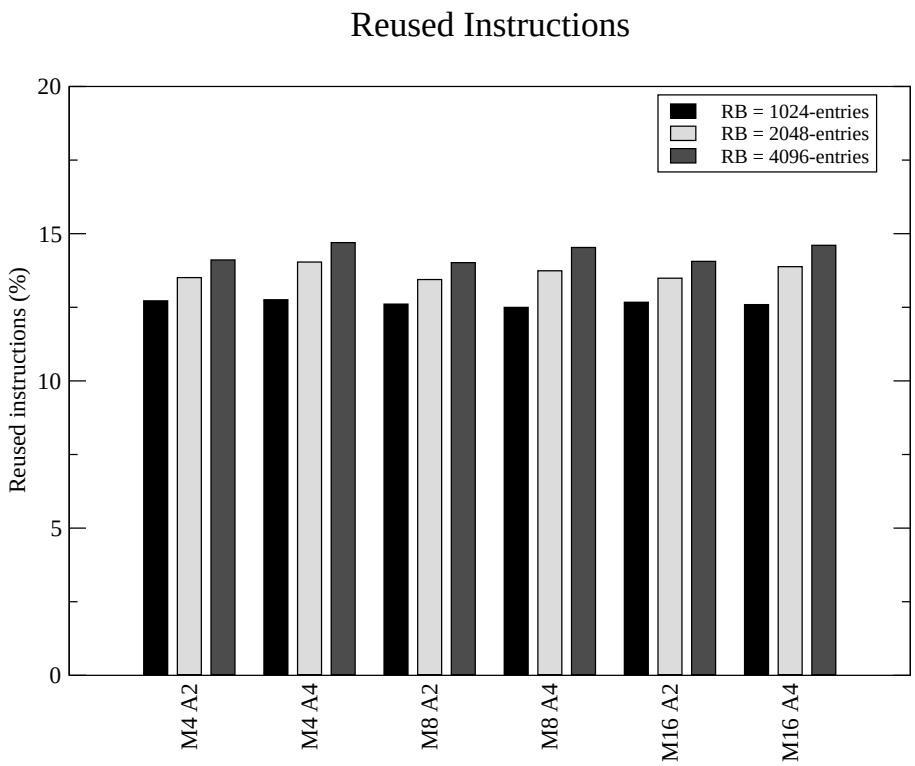

Figure 8.1: Percentage of reused instructions (8 wide architecture)

the percentage of the overhead reduction. Each bar shows the harmonic mean of all benchmarks for a given configuration.

As discussed before, the overhead caused by the fetch of multiple paths and also by the creation of replicas is one of the major problems in DCE. And this is the main motivation for this work. In Figure 8.2 this result may be seen. Depending on the configuration, the harmonic mean may achieve 16% of decrease in the number of executed instructions. Even for a 1024-entries buffer more than 12% of instructions are fetched but not executed, because they are reused. This means a great resource release and it is a direct effect from the instruction reuse.

8.1.3 Overall Performance

The harmonic mean for the increase in DCE performance using instruction reuse can be seen in Figure 8.3. The vertical axis shows the percentage of increase in DCE speedup. The horizontal axis shows the configurations simulated. Each vertical bar presents the harmonic mean of all benchmarks for that configuration.

The speedup presented here is the relation between the IPC achieved by the original DCE and the one reached by DCE using the instruction reuse mechanism.

It is possible to see that the final performance does not change significantly, in average. Benchmark *perl* presented the most significant rate, achieving almost 10% increase in some specific configurations. *Cc1* and *mgrid* benchmarks, however, were the main responsible for such low rate, in average. These benchmarks did not achieve more than 0.5% of speedup increase.

Even with similar results in the number of instructions reused among configurations, the speedup did not follow such pattern. In the cases where the speedup is

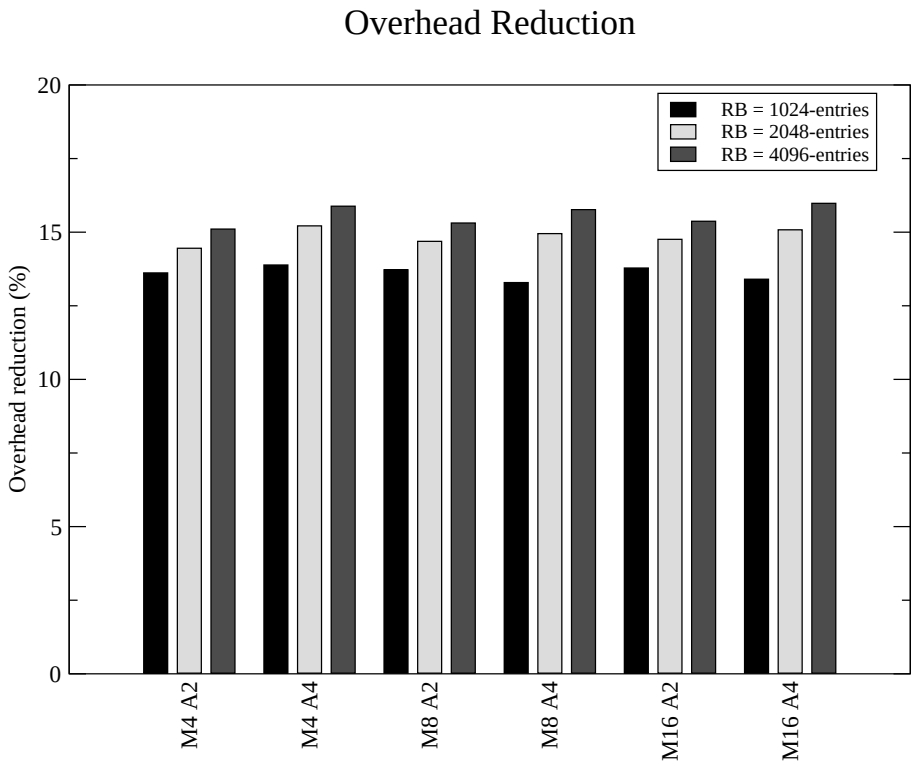

Figure 8.2: Percentage of overheard reduction

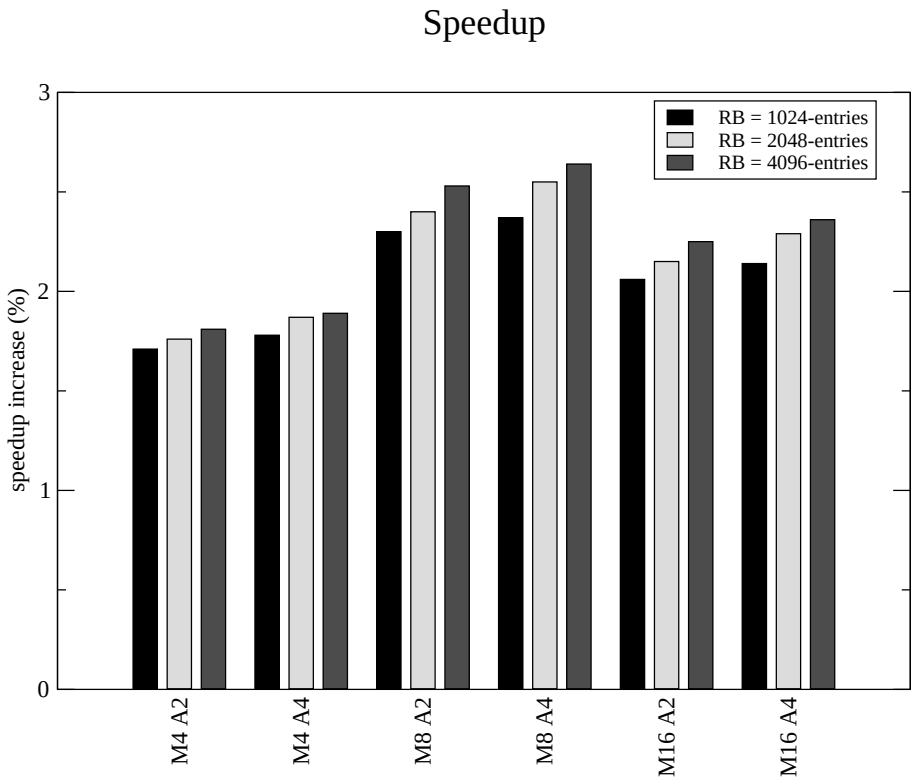

Figure 8.3: Percentage of speedup increase

higher, reused instructions are being committed faster. This is true when there is a larger number of tables available, except for configurations with 16 mapping tables and a 1024–entries, 4–way associative RB. For a 4–instructions–wide architecture this difference is even larger and the smaller performance (1%) was achieved by configurations with 4 mapping tables.

The problem with reusing instructions is that even saving resources in the execution stages, the instructions are still in the pipeline, in the ROBs. This means that even decreasing in the order of 15% the number of instructions executed, those instructions are maintained in the pipeline to assure semantic correctness in the commit stage.

The simulations executed in this work confirm that resorting to instruction reuse in conjunction with DCE is a good alternative to reduce the predication overhead and to increase the resources availability for more useful instructions.

The increase in the number of renaming tables, however, did not produce a significant increase in the number of reused instructions, as one may expect. When a instruction is reused, pipeline stages and functional units are released, as it is clearly observed by the large overhead reduction. However, these reused instructions keep on using the ROB, to assure that the same logical execution order is observed. These simulations indicates that in DCE, the ROB size as well as instruction re-ordering may be now important bottlenecks of the system.

Furthermore, the speedup achieved by the experiments did not increase in the same proportion as the overhead reduction. In fact, the percentage of increase was not very significant. These low performance results were also observed in other reuse studies (CITRON; FEITELSON, 2002) and they may be another side effect from the ROB occupation.

Finally, it is possible to say that instruction reuse can effectively reduce the great bottleneck in DCE, i.e., the overhead caused by the fetch of multiple paths and the replicas generation. The next steps, however, is to solve the problem of ROB saturation. The alternative pursued by this research was to increase the granularity of the mechanism and reuse entire traces, extracted mainly from the join point up to the resolution of the branch as well as from the several instruction flows available. In the case of trace reuse, a significant number of ROB entries can be released because only part of the instructions is forwarded to the ReOrder Buffer. This is possible because the architecture just needs to know the values of the traces output contexts.

The results achieved by reusing whole traces are more deeply studied in the following Sections.

8.2 Reusing Traces as an Alternative to DCE overhead

Reusing single instructions only did not significantly increased performance. As discussed before, although instructions are ready in earlier stages, they still have to wait for all previous instructions to be ready to commit.

Reusing traces naturally reduces this problem. When a trace is reused several instructions are permanently removed from the pipeline, avoiding commit stage and ROB. This is possible because only instructions that produce values to be used by subsequent instructions are kept, as explained in Section 6.2.

In the next Sections the results achieved by reusing traces in DCE are discussed. The baseline architecture in all cases is the original DCE, configured using the same

parameters used with trace reuse.

In all Figures, the horizontal axis means the configurations simulated. The abbreviations follows a similar standard used in previous Sections: *M4 W8* equals to 4 mapping tables in an 8-instruction-wide architecture; *M16 W4* equals to 16 mapping tables in a 4-instruction-wide architecture; *M64 W4* equals to 64 mapping tables in a 4-instruction-wide architecture; and so on. The remaining portion of the abbreviations used in Figures are described in Table 8.1.

Table 8.1: Reuse and Trace Buffers configurations showed in results

Configuration	Description
RB 1	RB: 1024 entries, 2-way associative; TB: 512 entries, 2-way associative
RB 2	RB: 1024 entries, 4-way associative; TB: 512 entries, 4-way associative
RB 3	RB: 2048 entries, 2-way associative; TB: 1024 entries, 2-way associative
RB 4	RB: 2048 entries, 4-way associative; TB: 1024 entries, 4-way associative
RB 5	RB: 4096 entries, 2-way associative; TB: 2048 entries, 2-way associative
RB 6	RB: 4096 entries, 4-way associative; TB: 2048 entries, 4-way associative

As discussed before, although simulations were performed using hundreds configurations, only part of them are showed. In general, the number of mapping tables was used to limit the visualization of the results. This was necessary in order to better visualize, analyze and understand all results reached by all simulations performed. The number of mapping tables was chosen as limit because it does not impact significantly in the results achieved.

The vertical axis, bars and lines differ for each case and they are described in each following Section.

8.2.1 Traces Characteristics

Initially, the average size of traces produced and reused by DCE architecture was observed. Traces in DCE tend to be larger than in a regular architecture due to traces built just after a join point. However, in this study traces are limited to basic blocks, as stated and discussed before. Even with such constrain, traces have at least the same average number of instructions as in a conventional architecture (COSTA, 2001).

Figure 8.4 shows the average size of traces in selected configurations. The upper portion of the Figure shows the results achieved by simulating configurations with limited resources, followed by the results reached with extremely wide architectures. The vertical axis means the harmonic mean of all eight benchmarks for the trace sizes.

It is possible to see that the size of the traces does not depend strongly on the architecture configuration. The harmonic mean is very similar among all and it is around 2.5 instructions for all configurations simulated, even allowing traces with

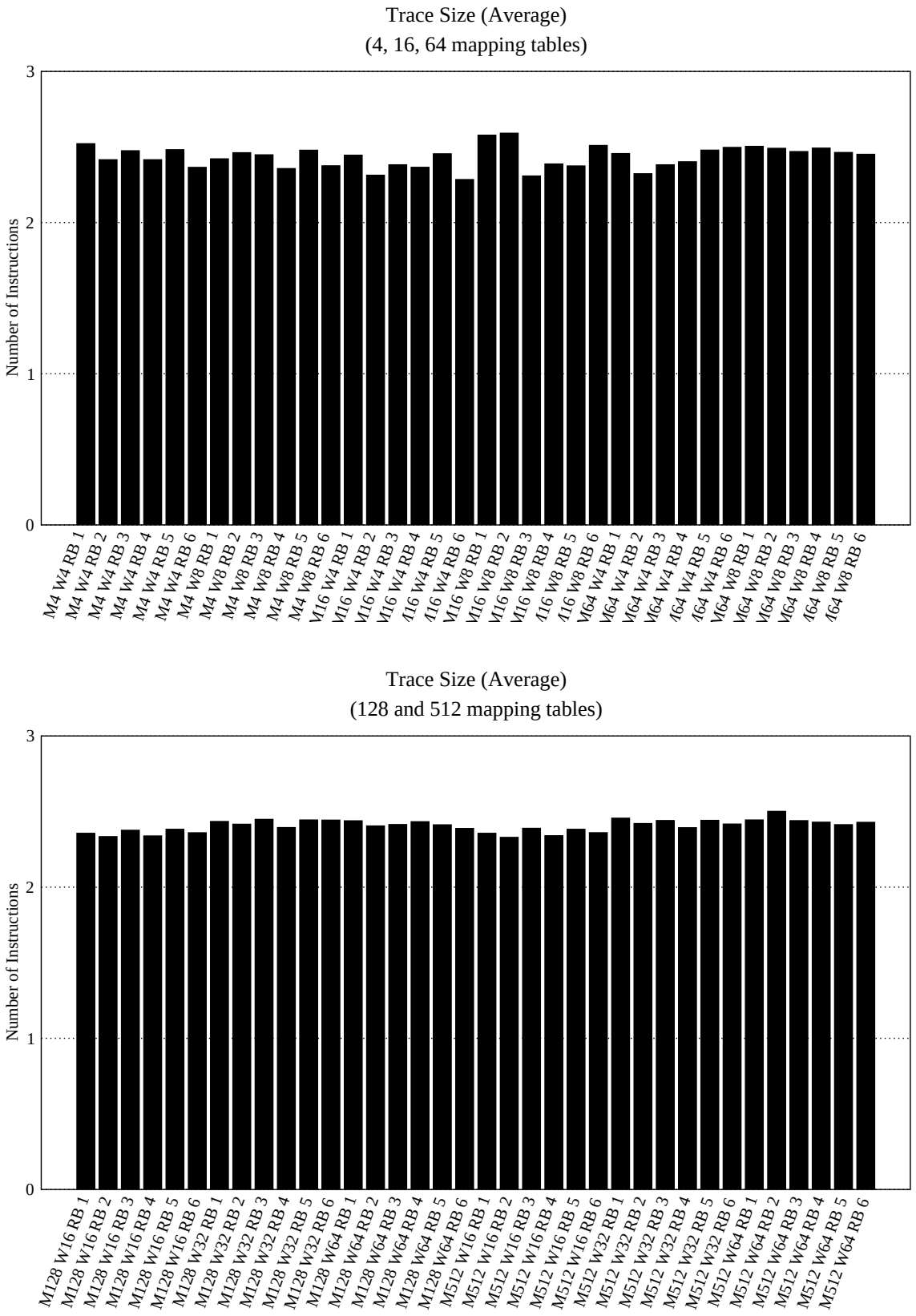

Figure 8.4: Trace average size (small and large architectures)

up to 4 instructions.

Branches are typically part of traces studied in previous works, but in DCE the reuse of branches needs to be carefully done, due to predicted branches in the data dependent piece of code from another previously predicated branch. This issue was detailed approached in Chapter 6. It is the intention to allow branches to be part of traces in a future work. Branches in traces may be allowed at least when all data dependent branches are predicted to the same target.

8.2.2 Overhead Reduction

As discussed before, the overhead reduction in DCE architecture is the main goal of this work. Thus, the reduction in the number of instructions executed when reusing traces is approached in this Section.

Basically, when an instruction is marked as reused, it is not executed and the overhead is decreased. In trace reuse, all instructions part of the trace and also instructions that are reused singularly are counted as non-executed instructions. Besides those, there are the ones that were removed from the pipeline, because they do not produce any outputs for subsequent instructions. Those *avoided* instructions are not part of the statistic showed here.

Figure 8.5 presents the overhead reduction for each benchmark in the selected configurations simulated. The upper part depicts the results achieved when using limited resources, while the lower portion shows the results reached with extremely wide architectures. The vertical axis in this case is the percentage of reduction, while each line in the Figure depicts a benchmark.

It is possible to see that, for the small configurations (upper part of the Figure), the overhead reduction varies significantly among benchmarks as well as among the configurations simulated. Also, benchmark *turb3d* was the one with the most remarkable results. In some cases, *turb3d* achieved 95% of reduction in the number of instructions executed. This means that only 5% of the original instructions were actually executed by the functional units. This is observed mainly for a small number of mapping tables. When the number of mapping tables increases, the reduction of the overhead is less significant and reaches a minimum of 60% of reduction when simulating an 8-instruction wide architecture, with 64 mapping tables. It is also observed that the associativity of the reuse buffers is also relevant and, in general, 4-way set associativity tables achieved more satisfactory results. This decrease in the overhead reduction while increasing the resources of the architecture is also verified in other benchmarks, such as *m88ksim*. In this case, the benchmark has an initial improvement when the architecture is widen from 4 to 8 instructions, in configurations with 4 mapping tables. When the number of mapping tables is increased, benchmark *m88ksim*, like *turb3d*, achieved lower numbers in the overhead reduction, going from 40% to 18% of reduction.

Benchmarks *cc1* and *go* presented a significant growth when resources are increased. After increasing the architecture width, the number of mapping tables and the reuse tables, benchmark *go* presented an increase in the overhead reduction rate from 25% to almost 55%. Benchmark *cc1* achieved one of the most irregular results and even with an improvement in the results when the resources are increased.

Benchmark *applu* and *perl* are the most sensitive ones to the associativity of the reuse tables. In general, it is possible to observe that *applu* presents a difference of almost 20% when the associativity is varied from 2 to 4-way. In benchmark *perl* this

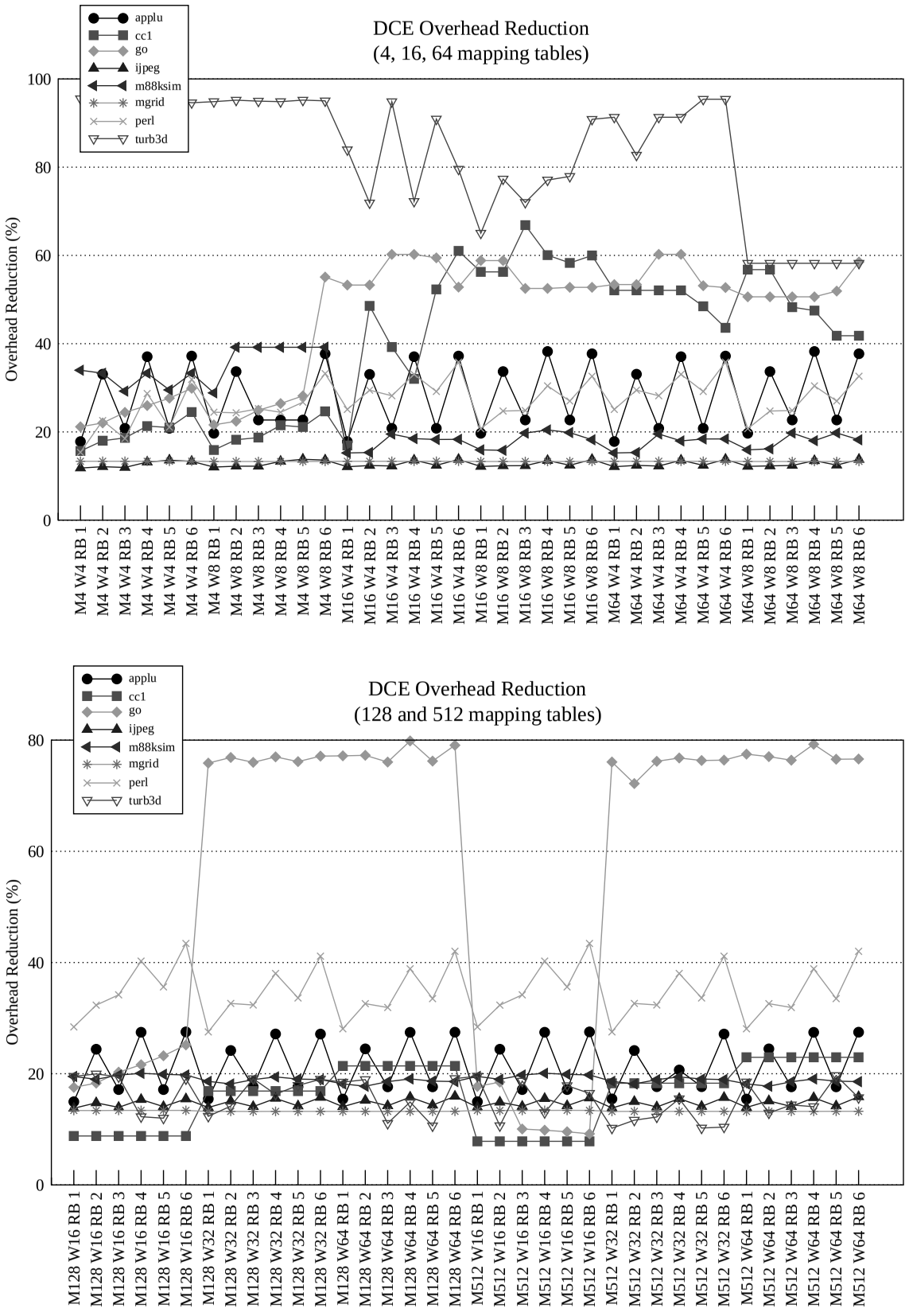

Figure 8.5: Overhead reduction (small and large architectures)

difference is smaller, around 5%.

Benchmarks *ijpeg* and *mgrid* achieved the most constant results reaching around 12% and 13%, respectively, of overhead reduction for all configurations.

On the other hand, for the larger architectures (lower part of the Figure), benchmark *go* was the one with the highest rates of overhead decrease. The rates are very significant, specially, in 32 and 64 wide architectures. In fact, for 16-wide architectures, the rates decreased from 80% to 20% or less. In wider architectures, there are more opportunities to reuse as there are more paths in course.

Benchmarks *cc1* and *turb3d*, however, presented a decrease in the overhead reduction rates, when compared to the smaller architectures. This trend is already observed when looking at architectures with fewer resources and 64 mapping tables. In that case, the decrease in the rates is already verified. This decrease is even higher in very aggressive architectures and *cc1* presents a maximum of 22% of overhead reduction, while *turb3d* achieves 19% of reduction.

Benchmark *applu* presented a similar behavior for the wide architecture configurations, as it was for the smaller ones. The rates, however, were less significant, not exceeding 27%. Benchmarks *m88ksim*, *mgrid* and *ijpeg* were also very similar, with *mgrid* presenting an almost flat rate of 13%, *m88ksim* varying from 17% to 20%, while *ijpeg* varied from 13% to 15% of reduction in the overhead.

Benchmark *perl* achieved better results in the case of wider architectures, achieving more than 40% of decrease in the overhead in some cases. This benchmarks is one of the most sensitive to the RB configurations. In general, increasing the size of the tables affects greatly the results and 4-way associative tables are better, except when they change from configuration 3 to 4.

Figure 8.6 shows the harmonic mean of all benchmarks for each configuration. As the previous Figures, the upper part is the results achieved when using limited resources, while the lower portion presents results of the large architecture configurations. The vertical axis means the average of the overhead reduction, while each bar depicts the harmonic mean of all benchmarks for a given configuration.

In general, larger architectures have a smaller reduction in the overhead. This is a very peculiar behavior and happens, mainly, because all traces, including the ones from the wrong paths, are formed, stored and reused. This means that in large architectures specially, the pollution brought by traces formed by paths and replicas never used again may affect the effectiveness of the mechanism.

It is possible to verify that the configurations with 4-way associative reuse tables are better in both, in general. This means that up to 4 traces of replicas may be stored in each way of the tables and they may be reused for several paths in a single step. Moreover, the difference among configurations are not very significant, not even when the reuse tables have their sizes doubled. The rates are all in the range of 15% to 26%.

It is also seem that the average of reduction is significantly greater than the reduction using single instruction reuse only, as described in Section 8.1.2.

8.2.3 Number of Cycles that an Instruction Remains in the Pipeline

One of the main problems of reusing instructions is such as even if a given instruction is not executed, it has to wait for all previous instructions before commit. This is necessary because execution is done out-of-order, but commit is an in-order task still.

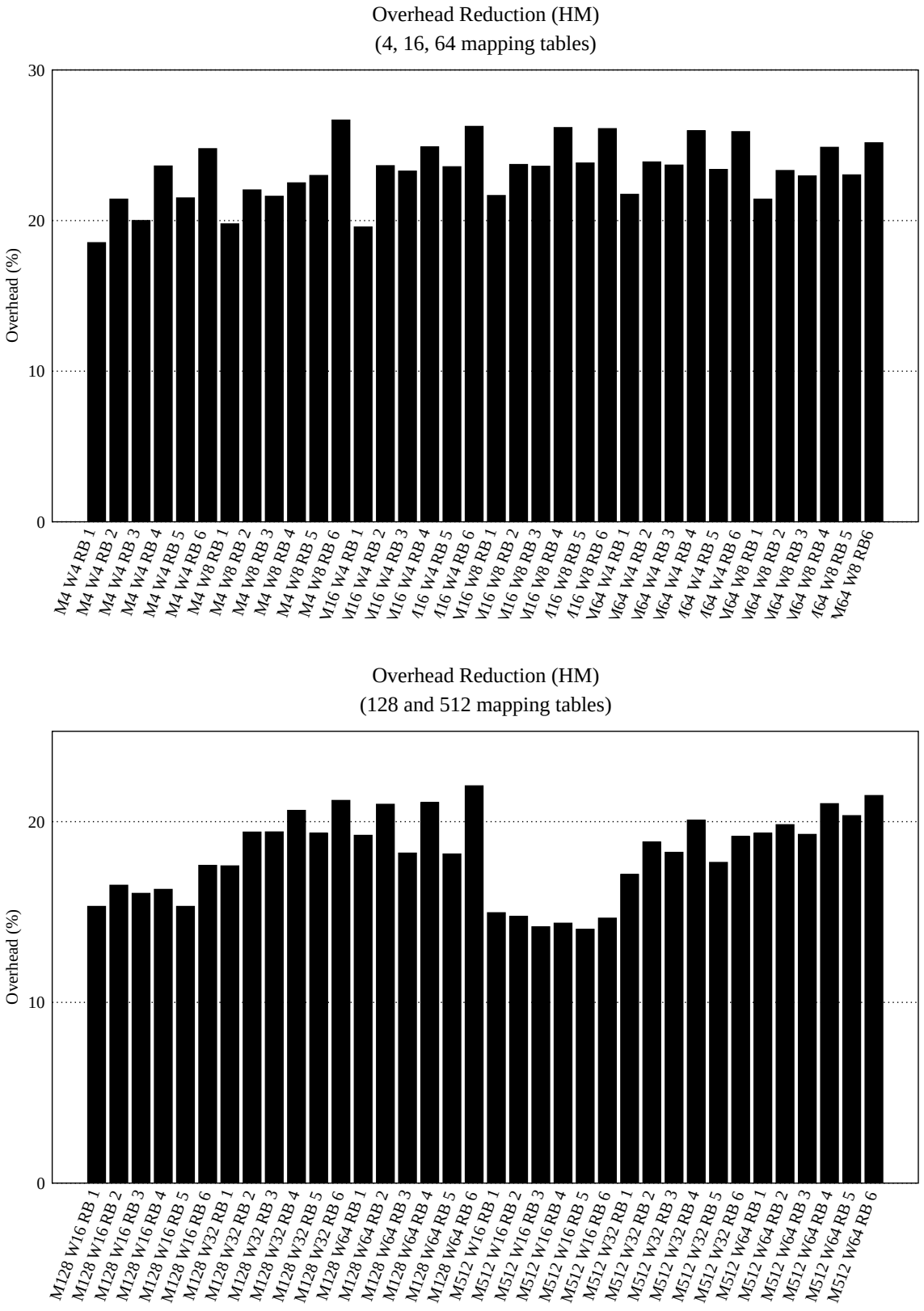

Figure 8.6: Average of overhead reduction (small and large architectures)

Even with such problem, the instructions reused tend to spend a smaller number of cycles in the pipeline.

Furthermore, allowing a reused instruction to be retired earlier also indirectly affects non-reused instructions retirement. In this case, those instructions will be allowed to be retired sooner as the previous (and reused) ones are committing as well.

Figure 8.7 shows the harmonic mean of cycles in which instructions remains in pipeline. The upper portion of the Figure depicts the results achieved when using limited resources, while the lower part means the results reached with wide architectures. Vertical axis means the average of the number of cycles spent between fetch and commit stages. The black bars depict the harmonic mean of all benchmarks for the number of cycles that a non-reused instruction remains in pipeline. The gray bar depicts the same metric, but for reused instructions.

In the case of smaller configurations, it is observed that, even committing in-order, the number of cycles that reused instructions remains in pipeline is significantly smaller than the number of cycles spent in pipeline by non-reused instructions. In some cases, especially for the larger configurations, the difference between reused and non-reused instructions is greater than 35%.

On the other hand, for aggressive configurations, there is no significant difference among the number of cycles used by an instruction reused and one not reused. In fact, there are some specific cases that not reused instructions are actually committing faster, in average. This may occur because the reused instructions have to wait longer for previous instructions to commit. As consequence, they end up committing later in some cases, if those not reused instructions are also taking long to execute and commit.

8.2.4 Speedup

One of the most important aspects when studying computer architecture is the performance reached by the new mechanisms introduced. This Section evaluates the speedup achieved by reusing traces in conjunction with DCE optimizations.

In this Section the speedup is calculated using the IPC of each configuration, with and without trace reuse. The baseline IPCs considered are the ones achieved by the original DCE, i.e., with single instruction and trace reused turned off. Thus, the speedup is calculated by using the following:

$$Speedup = \frac{IPC - baseline\ IPC}{baseline\ IPC}$$

Where IPC is the number of instructions per cycle achieved. *Baseline* keyword means the same statistic in the correspondent base architecture (with value reuse off).

Figure 8.8 shows the percentage of gain in speedup reached by the benchmarks with the selected configurations. As before, the upper portion of the Figure shows the results achieved by simulating configurations with limited resources, followed by the results reached with extremely wide architectures. The vertical axis means the percentage of increase in the speedup, when reusing values. Each line depicts the behavior of a given benchmark.

It is possible to see that the gain in speedup varies significantly according to the configuration and benchmark used in simulations, especially in the smaller architecture configurations.

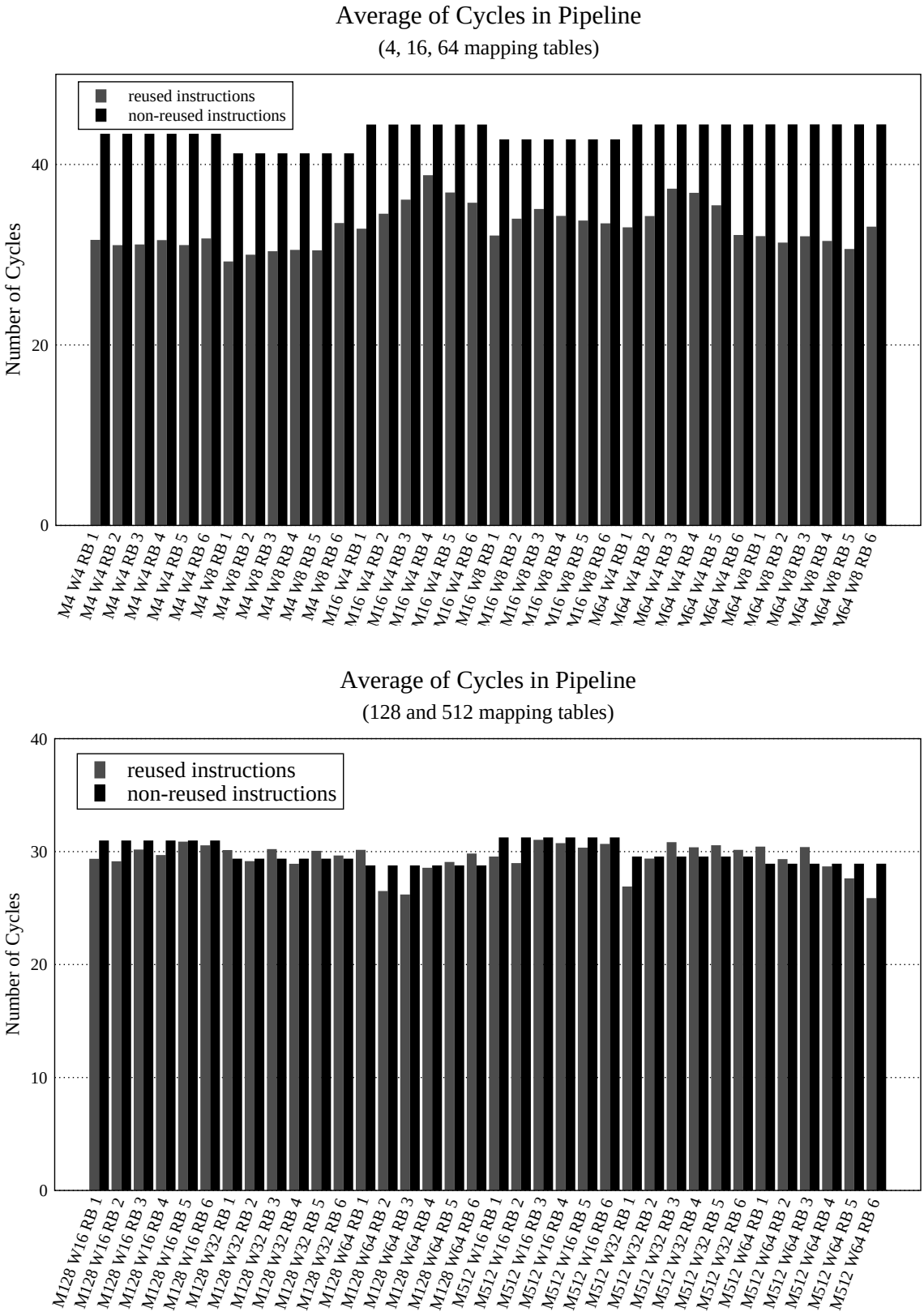

Figure 8.7: Average of cycles in pipeline (small and large architectures)

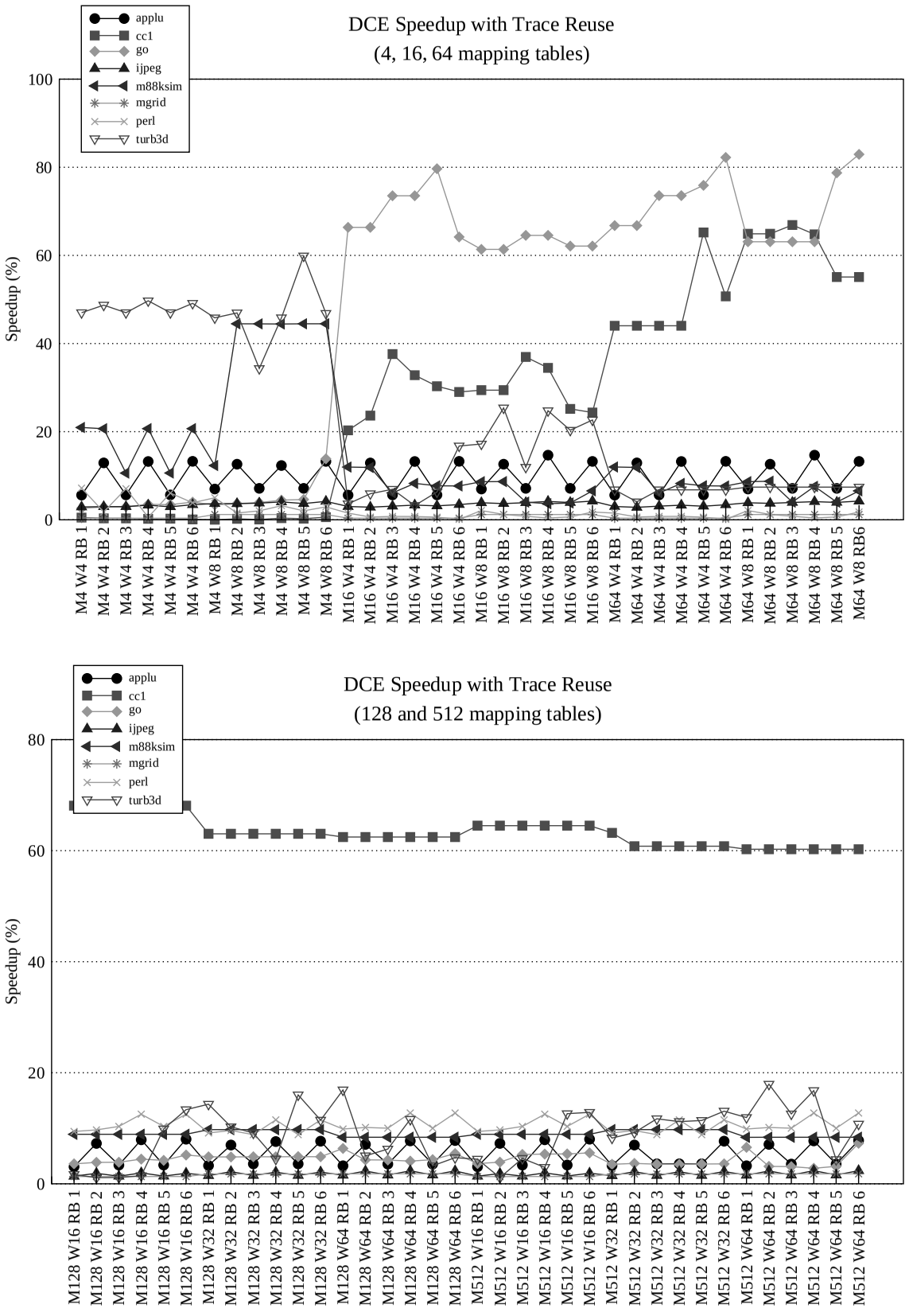

Figure 8.8: Percentage of gain in speedup (small and large architectures)

In these cases, i.e., small architectures, benchmark *go* increases its percentage gain from 13% to 66%, when the number of mapping tables goes from 4 to 16. This improvement is kept very high, in the range of 61% to 81%, during all other experiments. For wide architectures this benchmark did not present such speedup, not exceeding 7%.

Benchmark *cc1* also increased its speedup gain, when the resources were increased. For the architectures with 4 mapping tables only, the speed gain was not larger than 1%. When the number of mapping tables was increased, *cc1* achieved 60% of gain in some configurations. Trace reuse, in this case, was crucial to control the overhead introduced by DCE and increase performance. In case of large architectures, *cc1* achieved the highest speedups, exceeding 65% of gain. This is especially significative because benchmark *cc1* is well known to be a hard-to-predict application.

On the other hand, benchmark *m88ksim*, reduces its percentage of gain, when increasing the number of mapping tables in the smaller configurations. Nevertheless, the gains are kept flat for the aggressive architectures configurations. This benchmark has a sensitive improvement in 8-instructions-wide architectures with 4 mapping tables, achieving almost 45% of gain. However, this rate decreases for a range between 3% and 11% of improvement. It is possible to observe, as showed in Section 8.2.2, that the overhead reduction also decreased after DCE resources were increased. Thus, as there are more instructions executing regularly, speedup tends to be decreased. For the wide configurations, the gains are mostly constant, varying from 8.3% and 9.7%.

Benchmark *turb3d* also had the gain in performance decreased, in more robust configurations. Even with the highest rates in the overhead reduction, *turb3d*, did not reach such impressive results in speedup. In general, it reached very significant results for architectures with 4 mapping tables, almost 60% in some cases. For the remaining configurations, it achieved better results mostly in 8-wide-instructions architecture, with 25% of gain in some cases. In the wide architectures, the gains are even lower, not achieving 10%.

Benchmark *applu* behaves exactly as in the overhead reduction, but following a smaller scale. This happens in both small and large architecture configurations. The improvements are always more significant in architectures with 4-way associative reuse tables. In some cases, *applu* achieves almost 15% of gain in speedup.

For the smaller configurations, benchmarks *mgrid*, *ijpeg* and *perl* are the ones that presented the lowest performances. Benchmarks *mgrid* and *ijpeg* were the benchmarks with the smaller rates in overhead reduction. Benchmark *perl*, however, presented a significant reduction in the number of executed instructions, but this was not sufficient to increase performance and the gains are, in general, lower than 2%. Benchmark *perl*, however, achieved significant results in the wide configurations, achieving speedups from 8% to 12%.

Tables 8.2 and 8.3 show all speedups gains, in percentage, reached by the benchmarks in each selected configurations. The last column shows the harmonic mean of all benchmarks for the given configuration showed in the first column. In general, value reuse reached better results in very wide architectures, while some configurations even achieved more than 5% of gain. However, for some specific applications, such as *turb3d* and *mgrid* the impact of reusing instructions and traces is more positive in the smaller architectures.

Table 8.2: Harmonic Mean (in percentage) for the DCE speedup with trace reuse (4, 16, 64 mapping tables)

Configuration	applu	cc1	go	ijpeg	m88ksim	mgrid	perl	turb3d	HM
M4 W4 RB 1	5.60	0.50	2.59	2.91	20.96	0.37	7.18	47.01	1.37
M4 W4 RB 2	12.92	0.33	2.92	2.98	20.67	0.37	1.79	48.72	1.12
M4 W4 RB 3	5.65	0.33	2.95	2.98	10.59	0.37	7.05	47.01	1.16
M4 W4 RB 4	13.23	0.22	3.51	3.32	20.68	0.37	0.59	49.69	0.82
M4 W4 RB 5	5.65	0.25	3.43	2.96	10.61	0.37	6.07	47.01	1.03
M4 W4 RB 6	13.26	0.08	4.01	3.43	20.69	0.37	3.87	49.12	0.52
M4 W8 RB 1	6.99	0.02	3.38	3.69	12.31	1.20	5.01	45.83	0.18
M4 W8 RB 2	12.62	0.22	3.69	3.66	44.49	1.20	1.53	46.95	1.21
M4 W8 RB 3	7.13	0.07	3.90	3.78	44.46	1.20	2.03	34.33	0.49
M4 W8 RB 4	12.32	0.38	4.46	4.06	44.43	1.20	3.21	45.84	1.82
M4 W8 RB 5	7.13	0.25	4.62	3.68	44.49	1.20	2.03	59.88	1.32
M4 W8 RB 6	13.25	0.60	13.83	4.19	44.48	1.20	2.94	46.84	2.45
M16 W4 RB 1	5.60	20.30	66.37	3.01	11.97	0.36	1.52	3.62	1.84
M16 W4 RB 2	12.90	23.64	66.37	2.86	11.88	0.36	0.63	5.92	1.58
M16 W4 RB 3	5.65	37.61	73.53	3.10	6.10	0.36	0.76	6.93	1.62
M16 W4 RB 4	13.23	32.82	73.53	3.34	8.27	0.36	0.75	3.23	1.62
M16 W4 RB 5	5.65	30.31	79.67	3.16	7.72	0.36	0.61	6.32	1.53
M16 W4 RB 6	13.26	29.01	64.18	3.46	7.68	0.36	0.11	16.75	0.64
M16 W8 RB 1	6.99	29.42	61.37	3.86	8.65	1.20	2.10	17.20	4.13
M16 W8 RB 2	12.62	29.42	61.37	3.69	8.67	1.20	1.21	25.39	3.61
M16 W8 RB 3	7.13	36.96	64.56	3.89	4.01	1.20	0.80	11.92	2.80
M16 W8 RB 4	14.67	34.48	64.56	4.17	3.66	1.20	0.39	24.75	1.98
M16 W8 RB 5	7.13	25.17	62.12	3.92	3.82	1.20	0.64	20.33	2.54
M16 W8 RB 6	13.25	24.35	62.12	4.25	6.56	1.20	1.77	22.66	4.07
M64 W4 RB 1	5.60	44.03	66.77	3.02	11.98	0.36	1.52	6.80	1.91
M64 W4 RB 2	12.92	44.03	66.77	2.83	11.88	0.36	0.63	4.08	1.56
M64 W4 RB 3	5.65	44.03	73.57	3.11	6.10	0.36	0.76	6.80	1.62
M64 W4 RB 4	13.23	44.03	73.57	3.32	8.26	0.36	0.79	6.80	1.70
M64 W4 RB 5	5.65	65.21	75.90	3.09	7.72	0.36	0.61	6.80	1.53
M64 W4 RB 6	13.26	50.71	82.23	3.42	7.67	0.36	0.11	6.80	0.63
M64 W8 RB 1	6.99	64.90	63.10	3.86	8.65	1.20	2.10	7.42	4.01
M64 W8 RB 2	12.62	64.90	63.10	3.74	8.74	1.20	1.21	7.42	3.50
M64 W8 RB 3	7.13	66.90	63.10	3.94	4.01	1.20	0.80	7.42	2.77
M64 W8 RB 4	14.67	64.78	63.10	4.12	7.58	1.20	0.39	7.42	2.01
M64 W8 RB 5	7.13	55.09	78.73	3.95	4.01	1.20	0.64	7.42	2.50
M64 W8 RB 6	13.25	55.09	82.93	4.25	6.56	1.20	1.77	7.42	3.94

Table 8.3: Harmonic Mean (in percentage) for the DCE speedup with trace reuse (128 and 512 mapping tables)

Configuration	applu	cc1	go	ijpeg	m88ksim	mgrid	perl	turb3d	HM
M128 W16 RB 1	3.08	68.10	3.60	1.36	8.92	1.32	9.43	1.68	2.73
M128 W16 RB 2	7.28	68.10	3.86	1.90	8.92	1.30	9.71	1.11	2.84
M128 W16 RB 3	3.38	68.10	3.89	1.37	8.92	1.32	10.33	1.14	2.54
M128 W16 RB 4	7.93	68.10	4.45	1.96	8.92	1.31	12.54	1.46	3.18
M128 W16 RB 5	3.38	68.10	4.22	1.39	8.92	1.32	10.33	9.88	3.43
M128 W16 RB 6	8.03	68.10	5.18	1.89	8.92	1.31	12.54	13.35	4.24
M128 W32 RB 1	3.29	63.04	4.86	1.47	9.76	1.77	9.16	14.34	3.90
M128 W32 RB 2	6.98	63.04	4.83	2.19	9.76	1.78	9.60	10.27	4.73
M128 W32 RB 3	3.59	63.04	4.88	1.50	9.76	1.77	8.86	9.15	3.89
M128 W32 RB 4	7.61	63.04	4.86	2.10	9.76	1.78	11.51	4.46	4.43
M128 W32 RB 5	3.59	63.04	4.90	1.51	9.76	1.77	8.86	16.01	3.99
M128 W32 RB 6	7.67	63.04	4.88	2.11	9.77	1.78	11.51	11.47	4.81
M128 W64 RB 1	3.25	62.45	6.38	1.55	8.39	1.80	9.87	16.91	4.08
M128 W64 RB 2	7.10	62.45	4.31	2.30	8.39	1.80	10.15	5.04	4.46
M128 W64 RB 3	3.55	62.45	4.29	1.57	8.39	1.80	10.02	6.26	3.81
M128 W64 RB 4	7.74	62.45	4.08	2.33	8.39	1.80	12.75	11.61	4.83
M128 W64 RB 5	3.55	62.45	4.35	1.59	8.39	1.80	10.02	3.24	3.57
M128 W64 RB 6	7.80	62.45	5.45	2.33	8.39	1.80	12.75	4.71	4.65
M512 W16 RB 1	3.08	64.50	3.64	1.37	8.92	1.32	9.43	4.44	3.14
M512 W16 RB 2	7.28	64.50	3.90	1.85	8.92	1.30	9.71	1.12	2.83
M512 W16 RB 3	3.38	64.50	5.28	1.39	8.92	1.32	10.33	4.57	3.32
M512 W16 RB 4	7.93	64.50	5.35	1.97	8.92	1.31	12.54	2.96	3.76
M512 W16 RB 5	3.38	64.50	5.38	1.40	8.92	1.32	10.33	12.63	3.55
M512 W16 RB 6	8.03	64.50	5.57	1.91	8.92	1.31	12.54	12.87	4.27
M512 W32 RB 1	3.29	63.21	3.52	1.47	9.76	1.77	9.16	8.24	3.66
M512 W32 RB 2	6.98	60.80	3.68	2.19	9.76	1.78	9.60	9.23	4.53
M512 W32 RB 3	3.59	60.80	3.54	1.49	9.76	1.77	8.86	11.72	3.79
M512 W32 RB 4	3.59	60.80	3.50	2.18	9.76	1.78	11.51	11.20	4.25
M512 W32 RB 5	3.59	60.80	3.56	1.50	9.76	1.77	8.86	11.41	3.80
M512 W32 RB 6	7.67	60.80	3.60	2.22	9.76	1.78	11.51	13.09	4.70
M512 W64 RB 1	3.25	60.25	6.58	1.55	8.38	1.80	9.87	11.93	4.04
M512 W64 RB 2	7.10	60.25	3.14	2.31	8.38	1.80	10.15	17.94	4.61
M512 W64 RB 3	3.55	60.25	3.10	1.58	8.38	1.80	10.02	12.59	3.79
M512 W64 RB 4	7.74	60.25	2.80	2.36	8.38	1.80	12.75	16.79	4.60
M512 W64 RB 5	3.55	60.25	3.16	1.59	8.38	1.80	10.02	4.37	3.56
M512 W64 RB 6	7.80	60.25	7.30	2.37	8.38	1.80	12.75	10.76	5.16

Although the main goal of this work is to analyze the impact of value reuse in DCE is also relevant to verify the effects in the original baseline presented in Chapter 4, i.e., a conventional superscalar architecture. Figure 8.9 presents the comparison between the speedup gains produced by value reuse in the original baseline and also in DCE. The vertical axis shows the speedup gain. The black bars mean the speedup over the baseline for each configuration, while the gray ones depict the speedup over DCE architecture. These gains are the harmonic mean of the benchmarks simulated in Chapter 4, i.e., *cc1*, *go*, *ijpeg*, *m88ksim* and *perl*.

In general, the gains over the baseline architecture exceed the gains produced by value reuse over DCE. This is not true only for the smaller configurations, with 4 mapping tables. Possibly, this is caused by the pollution brought by traces from the wrong paths that are never reused. It is important not just to build and store traces, but to have the useful ones available for reuse. Further investigation is still necessary in order to verify the real impact of storing those wrong path traces. Moreover, mechanisms to filter those traces can be developed in order to avoid this problem.

It is possible to see that, in general, the reduction in overhead affects directly the speedup reached by the architecture. However, it is also important to analyze what happens in DCE architecture itself. The side effects in the most important features in DCE are approached in the next Section.

8.2.5 Side Effects in DCE

8.2.5.1 Number of Predicated Branches

Dynamic predication is the most important feature in DCE. The logic to allow branches to predicate is the main core of this architecture. Thus, the number of predicated branches is one of the most relevant aspects to be evaluated.

Figure 8.10 shows the behavior of the number of predicated branches, with and without trace reuse. As in the previous Figures, the upper portion of the Figure shows the results achieved by simulating configurations with limited resources, followed by the results reached with extremely wide architectures. The vertical axis means the percentage of increase/decrease in the number of predications performed by the architecture. Each line depicts a benchmark simulated.

It is seen that the number of predications in all benchmarks was not affected in configurations with 4 mapping tables only. In fact, benchmarks *applu*, *ijpeg*, *m88ksim* and *perl* were not affected significantly at all. This is true for both small and wide architectures.

Benchmark *mgrid* had a slight increase in the number of predications, around 5% to 8% in all smaller configurations. For the wide architectures, the number of predications is increased, especially in 16-wide architectures.

In both types of architectures, benchmarks *cc1* and *go* reached the most suggestive results, presenting increases of up to 100% in the number of predications. In both benchmarks, this means that value reuse allowed the dynamic predication mechanism to go further in branch spawning. Benchmark *go* was the benchmark that presented the best speedups among all the smaller architectures and, as showed in Section 8.2.2, it did not presented an overhead reduction to support such high speedup rates. Thus, the increase in the number of predications did affect greatly the overall performance in those first cases. On the other hand, spawning up to 512

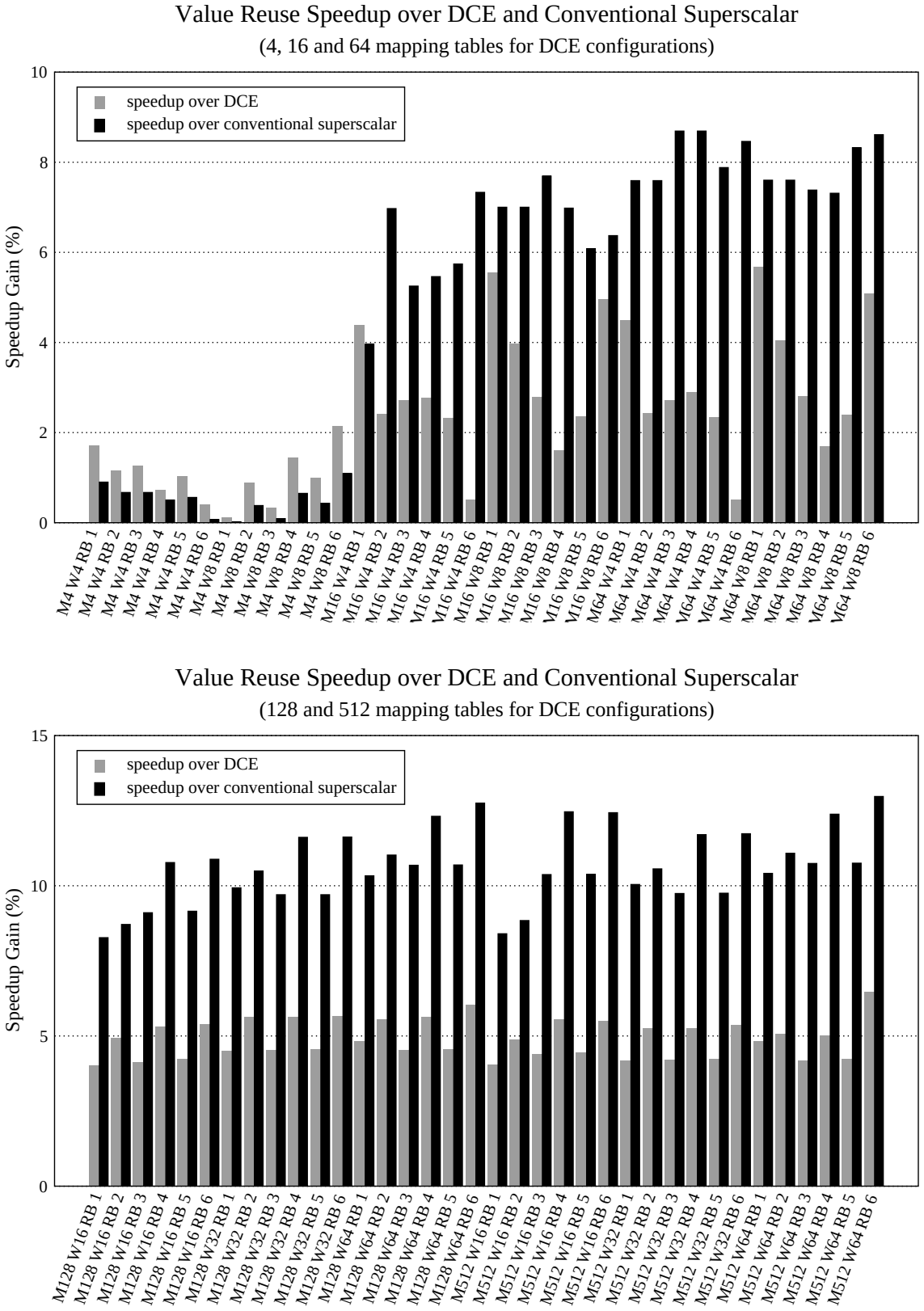

Figure 8.9: Percentage of gain in speedup of value reuse in DCE and in conventional superscalar architecture(small and large architectures)

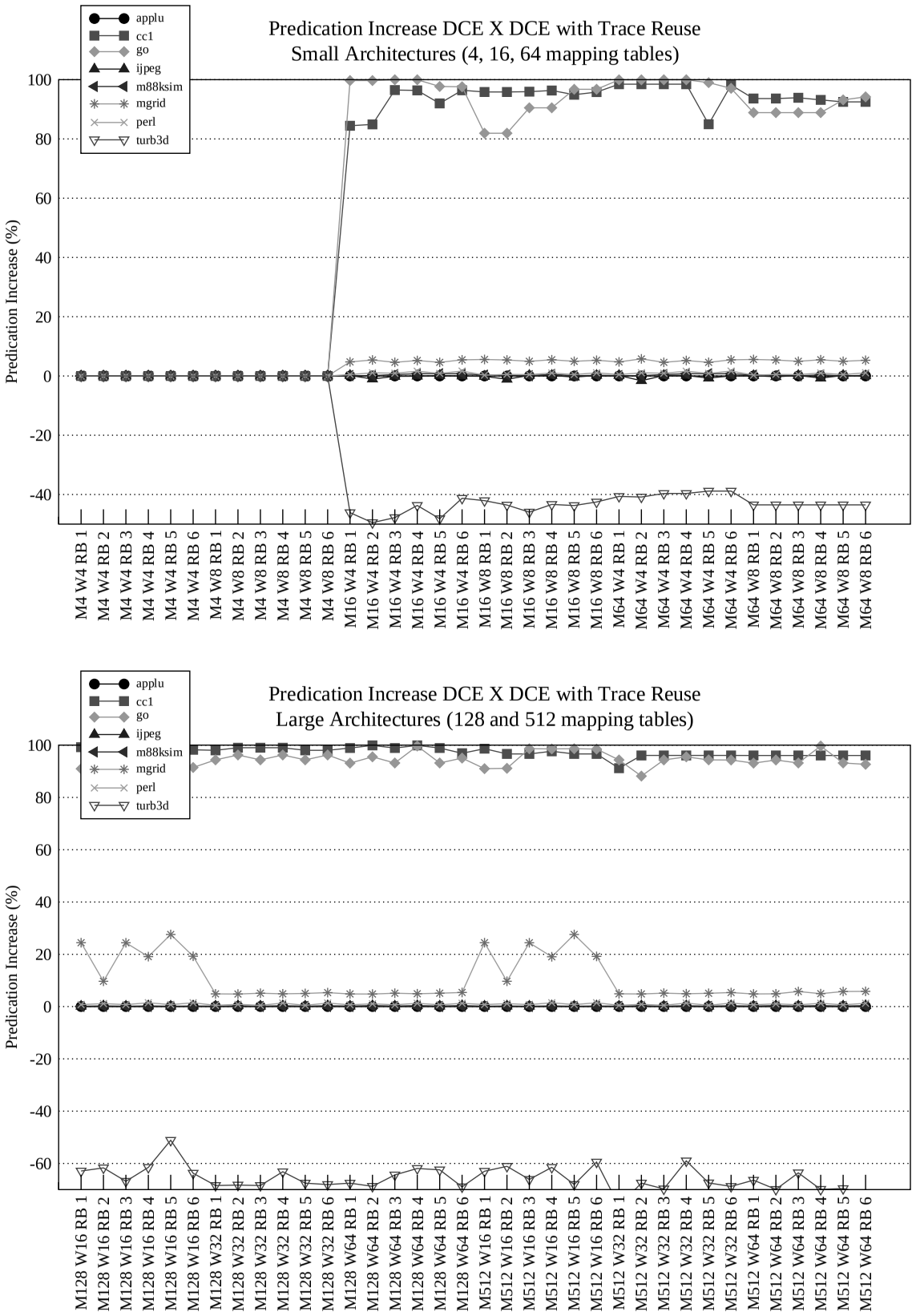

Figure 8.10: Number of predicated branches (small and large architectures)

paths did not bring the same benefits. Although achieving speedups improvements, there is a significant decrease in the gains.

For the small architectures, benchmark *cc1* behaves like *go*, in a lower scale. For the wide architectures, however, the effects of fetching so many paths are remarkably good.

Benchmark *turb3d* was the only benchmark that had a smaller number of branches being predicated. This may be one of the reasons the benchmark did not reach large speedup gain, even with a very high number of instructions not executed.

8.2.5.2 Mispredictions

The main goal of DCE is to control and ultimately reduce mispredictions. The intention is to improve performance avoiding predictions, using predication instead. As a consequence, a lower number of mispredictions is observed. This Section compares the occurrence of mispredictions with and without trace reuse.

Figure 8.11 shows the misprediction behavior achieved by the selected architecture configurations. The misprediction behavior is calculated using the following:

$$Misprediction\ Reduction\ Rate = \frac{baseline\ mispred\_number - mispred\_number}{baseline\ mispred\_number}$$

Where mispred_number is the number of cycles lost with the pipeline stalled due to mispredictions. *Baseline* keyword means the same metric achieved by the architecture with reuse optimizations off. It is possible to see that the reduction (or increase) of the misprediction is calculated over the absolute number and, therefore, they do not depict the percentage over the percentage.

The upper part depicts the results achieved when using limited resources, while the lower portion shows the results reached with extremely wide architectures. The vertical axis describes the percentage of the misprediction increase/decrease, when simulating DCE with and without value reuse. Each bar means the harmonic mean of the misprediction behavior, achieved when comparing original and extended DCE.

It is possible to see that, for the small architectures, reusing values actually increase mispredictions. This is an exception just for 4-wide instructions architectures with 4 mapping tables. And although the misprediction increase is not very large ranging, mainly, between 2% and 4%, it is an interesting point.

There is only one way in which a predicated branch may be counted as mispredicted. When a predicated branch is ready to commit and the last replica before the join point was not renamed, a misprediction is detected and the pipeline is flushed. This is necessary because when a predicated branch is committed, all tagids are required in order to retire the correct path from the pipeline. However, if the replicas are not yet in the pipeline this cannot be done. Also, stall just the commit stage would not work, because this may cause the whole pipeline to stall, preventing the creation of replicas and generating a deadlock. So, increase the number of predications is not enough, if they have not enough time to spawn completely.

As in some cases, especially in benchmarks *go* and *cc1*, more instructions are being predicated and many replicas do not have enough time to be generated. In the other cases, as the reuse actually anticipates the execution of all instructions, predicated branches reach the commit stage faster, and the event is again verified. Unfortunately, this reflects negatively in the overall results and certainly harms the

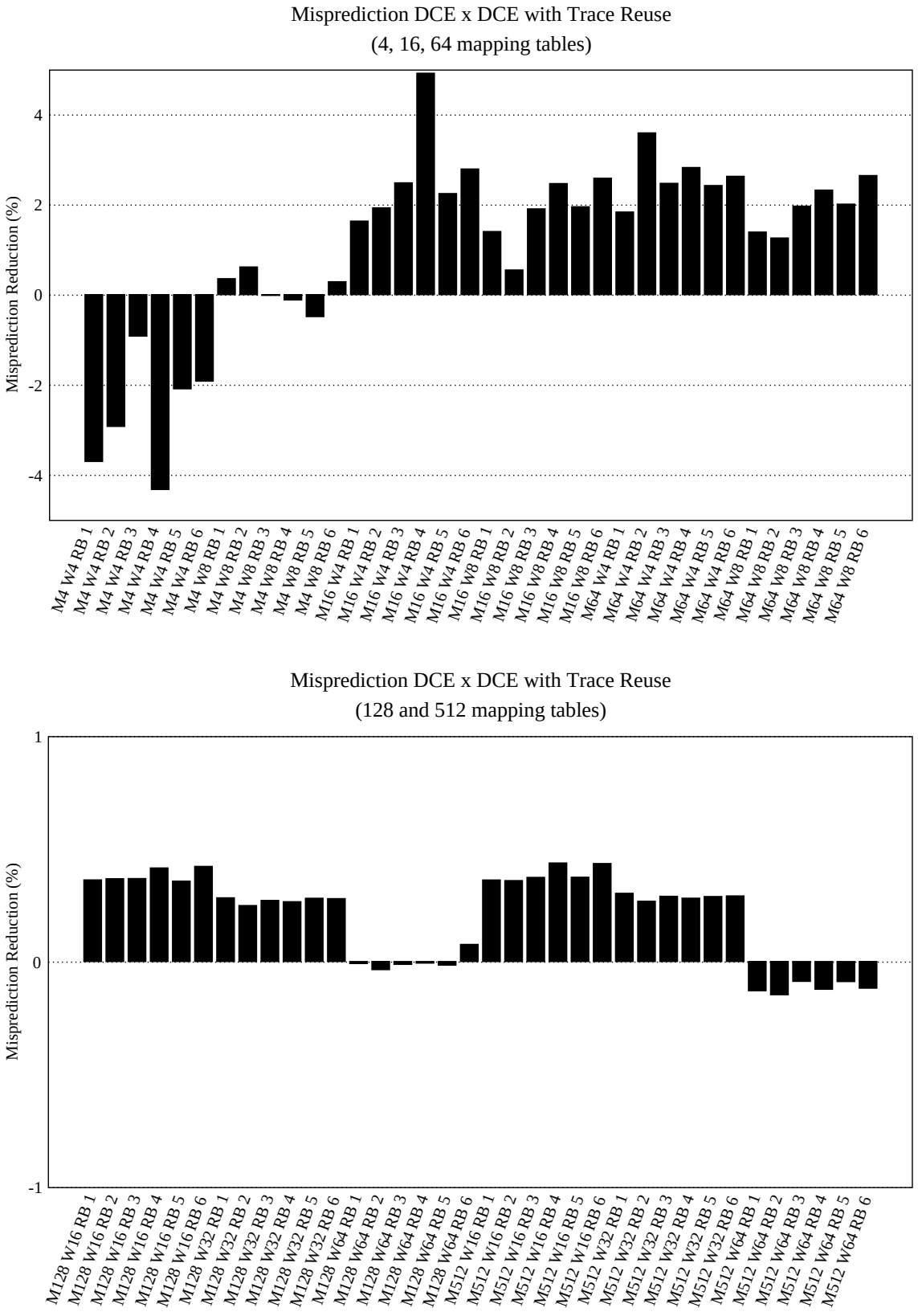

Figure 8.11: Number of mispredictions (small and large architectures)

performance when reusing values. This happens especially in the small configurations, where there is actually some significant rates.

In the case of wide architectures, the mispredictions rates are not significantly different, ranging from a decrease of 0.2% to an increase of 0.5% in the misprediction rates.

In general, however, instruction and trace reuse reduce the main problem of DCE, the fetch of several paths which will be later squashed and the replicas introduced by the architecture. And even presenting a low harmonic mean for the gain in speedup, the results are very representative and relevant.

8.3 Summary

The goal of this Chapter was to present the most significant results found in the simulations using the extended *sim-dce*, which allowed value reuse.

The first part of the Chapter showed results achieved by instructions reuse only. It is possible to verify that, even with a large overhead reduction, the speedup gains are not very significant and it is not more than 3%. The main problem with this approach is that instructions reused are kept in the ReOrder Buffer. They need to be kept because, in general, commit is performed in order, even in out-of-order architectures, such as the ones simulated in this work.

The second part, however, depicted the results using all optimizations, including instruction and trace reuse. Trace reuse is always prioritized, i.e., an instruction will be reused singularly only if there are no traces available (and ready) for that PC. In this case, the results were better, specially, because many instructions may be retired from the pipeline yet before the actual register renaming. Instructions that are part of the trace and do not produce output values used by subsequent instructions are useless and can be retired right the way. This actually reduces the number of instructions in the pipeline and allows more instructions to be fetched/decoded/renamed/executed.

It is possible to verify that the overhead reduction rates were all very significative. Trace reuse really decreases even more the number of instructions executed by the architecture. Nevertheless, this did not affected strongly and constantly the performance. The performance varies greatly according to the benchmark as well as the configuration simulated.

For the small architectures, some specific benchmarks reached impressive gain in performance, such as *m88ksim* and *turb3d*. The later one achieved 60% of improvement in a given configuration. On the other hand, some configurations ran in benchmarks such as *mgrid*, achieved less than 1% of gain. The large architectures, however, reached more steady results and, as consequence, reached a better speedup harmonic mean, achieving more than 5% of gain in some cases.

One may observe that, as in many others sophisticated high performance mechanisms, value reuse depends widely in the type of application run as well as in the architecture balance.

Next Chapter approaches the conclusions and remarks of this work.

9 CONCLUSIONS

Branch occurrence is still an issue that remains important to extract ILP and performance in superscalar processors, for which no definitive answer was found. The alternative to reduce these problems is, in general, to introduce complex and sophisticated predictions mechanisms. After all, there are no guarantees that the predicted path is the right one to follow and the pipeline pays a very high penalty when a misprediction occurs.

Also, this penalty is increasing constantly with deep state-of-the-art pipelines and even deeper ones scheduled to the next generations. The average of hit rate in current predictors is over than 90%, but the occurrence of 10% or less in mispredictions is sufficient to drastically decrease the performance.

The Dynamic Conditional Execution (DCE) is a new approach to reduce misprediction occurrence. The idea is to decrease misprediction by avoiding predicting branches, predicating some of them instead. The compiler is responsible to qualify all branch structures (hammocks) that may be predicated. Then, the architecture decides, in execution time, which ones of the qualified hammocks are really going to be predicated.

It is clear that only part of the branches may be predicated, otherwise the pipeline would saturate fast. For this reason, the compiler marks statically the hammocks that obey some constrains regarding, mainly, to their size and complexity. Typically, the DCE compiler qualifies five types of hammocks to be predication:

- Simple (one and two sided): branches with no nested structures;
- Complex pure: one or more nested structures totally contained in the most external one;
- Complex multiple join: one or more nested structures which have their target coinciding with their join points;
- Complex multiple target: one or more nested structures which have their targets after the join point;
- Complex overlapped: one or more nested structures which have their targets located in the taken path of the most external one.

Hammocks with backwards branches, indirect branches, unconditional jumps not related to the conditional branch flow control, system calls, subroutines call/return are never qualified for predication.

When a qualified branch is fetched, the architecture decides, based on resources availability, if that hammock is going to be predicated. If the architecture decides not to predicate the qualified branch, then it is predicted normally. On the other hand, when a branch is predicated, both paths are fetched in order to avoid a prediction. All paths are going to be fetched and executed and the right path is ultimately committed and the architecture squashes selectively the remaining paths.

The problem is to maintain the correctness among the different data chains, from the different paths. This is handled by the creation of several replicas of the same instruction. So, when the join point of a predicated hammock is found, the architecture introduces one replica for each path of each new instruction fetched. The replicas keep being created until the outcome of that branch is known, i.e., after the execution of the branch which originated the predication is completed.

Simulations have shown that misprediction occurrence was significantly decreased by DCE architecture (SANTOS, 2003). The speedup over a conventional superscalar, however, was not as impressive. In fact, for some cases there was no gain at all. Even selecting and qualifying just part of the branches, the pipeline still saturates, specially, after the rename stage when replicas are created.

This work aims to combine value reuse and dynamic predication in order to evaluate the performance of this approach. Therefore, the following can be pointed out as the main contributions:

- Evaluate the impact of the overhead in DCE architecture;
- Propose an alternative that effectively reduces this problem (value reuse);
- Validate the idea by the implementation of a detailed microprocessor simulator;
- Analyze all results achieved, identifying the pros and cons of pursuing such approach.

Among other alternatives considered, value reuse was found as the most suitable for DCE context. Thus, the main goal of this work was to introduce the idea of value reuse in DCE. Typically, architectures with multipath and predication require a wide execution structure. The pipeline stages work close to their limit and stalls due to the large number of instructions produced from the multiple paths are common.

Initially, only instruction reuse was studied. The idea was to develop a mechanism to store all previously executed instructions and reuse each single instruction when possible. An instruction may be reused when all its source operands are ready and match with the ones found in the Reuse Buffer (RB). In this event, the destination register is assigned with the value stored in RB and marked as ready. This happens just after the rename, in a new stage created to accommodate the reuse test and register assignment. The instruction is then forwarded to the ROB in order to wait for commit.

A simulator based on *sim-dce* was developed, allowing instruction reuse in DCE architecture. In this first part of the work, the DCE pipeline was extended to simulate configurable deep pipelines and also the reuse of single instructions only. The mechanism follows the same idea discussed above.

Simulations for reuse of single instructions have showed that the average for all benchmarks of overhead reduction ranges between 13% and 16%, depending on

the RB size. This means that the architecture is executing 13 to 16% less instructions. The harmonic mean for the speedups, however, was never greater than 2.5%, comparing to the original DCE. Also, the most significant rates were achieved by benchmark *perl*, around 10%. Benchmarks *cc1* and *mgrid* were the worst in performance gain and the improvements were in the order of 0.5% only.

Although the reused instructions do not need to execute, they still occupy ROB entries. Also, they need to wait for all previous instructions to commit because retirement is performed in-order. This is the largest limitation of this approach, as it transfers the bottleneck from the execution engine to the commit stage.

The second part of the work, allowed trace reuse. A trace is a sequence of n contiguous instructions that may be reused at once. This means that, based on a single reuse test, several instructions may be reused. In this case, a new buffer is necessary to hold the input and output context of the traces, used to identify whether a trace may be reused or not. The input context is the set of registers which all instructions of a given trace uses as source operands. On the other hand, the output context is the set of registers used as destination by all instructions of the trace.

The reuse test is performed comparing all registers part of the input context and not only the source registers of a single instructions, as it was before. Similarly, all registers part of the output context are assigned at once, solving different instructions at once.

As the Trace Buffer (TB) always stores the last value produced by an output context register, some instructions may be directly retired from the pipeline, with no need to wait for commit in the ROBs. An instruction part of a trace will be forwarded to the ROBs only if it is producing values used by subsequent instructions. Thus, all instructions which produce values to be consumed by the proper trace are withdraw from the pipeline. This reduces the problem of ROB saturation, verified when reuse of single instruction was introduced.

In DCE, as multiples paths are in course, different traces are also being formed at once, one for each tagid identified. The idea is to reduce even more the overhead of executed instructions, as several traces are available to reuse.

In order to validate the idea, a trace reuse mechanism was developed over the already extended *sim-dce*. The traces are built in commit stage and stored in TB. When traces of replicas are built, they are stored in the several ways of the same TB set. The reuse test is performed yet in the rename stage in order to prevent unnecessary renaming. Logical registers used by instructions which are going to be directly retired are not supposed to be renamed, otherwise they would be occupying resources useful for other instructions.

All instructions that are not part of a trace are queried in the Reuse Buffer as it was done before. Hence, once the instructions are not found to be part of a valid trace, they can still be reused as single instructions. The reuse test is performed just after renaming as explained previously.

Simulations showed that the overhead reduction rates were around 8 to 10% larger than the ones produced by single instruction reuse. Moreover, the instructions which were removed from the pipeline were not counted in this statistic and it possible to state that these rates are significantly higher, if those instructions were also considered.

The average size of the traces reused is also an interesting point. Although the

actual implementation is not allowing branches to be reused, the average size of the traces is around 2.5 instructions. This is around the same size reached by other studies (COSTA, 2001). Probably, when branches are allowed this average is going to be higher, as thought initially. As DCE has several replicas after the join point and the architecture keep on generating them up to the outcome of the original branch is known, the traces of replicas are potentially larger.

The speedup reached by the trace reuse mechanism over the original DCE varies widely according, specially, to the benchmarks. For benchmark *go*, for example, trace reuse reached more than 80% of gain in some configurations. Benchmarks *applu*, *cc1* and *turb3d* have achieved significant gains of performance as well. Nevertheless, in other cases such as *mgrid* and *perl* it was observed a very low or no gain in performance. The configurations also greatly affect performance. In general, 4-way-associative reuse tables achieve better results. Possibly, in these cases, more traces of replicas are available to the different data chains, speeding up the overall performance.

The side effects in the most relevant DCE features were also analyzed. First, the number of branches dynamically predicated by the architecture was observed. After increasing the number of mapping tables, only benchmarks *cc1* and *go* presented a significative increase in the number of branches predicated. For these benchmarks, value reuse allowed more paths to be spawned by the architectures. It is fair to say that this affected positively the performance. As stated above, benchmark *go* achieved around 82% of gain in some cases. This gain is not exclusively due to the increase in the number of branches predicated, but this certainly affected the performance.

On the other hand, benchmark *turb3d*, incurred a great decrease in the number of predicated branches. As more branches are being predicted, the misprediction rates tend to be higher. This explains why *turb3d* presented the highest rates in the overhead reduction, but it did not reached the same results in the overall performance.

As discussed before, misprediction occurrence is typically smaller when DCE is used. This happens because a lower number of predictions is being performed. However, there is a special case in which predication may lead to a misprediction. A predicated branch is said to be mispredicted when it reaches the top of the ROB and there are replicas which were not renamed yet. This happens because when the predicated branch is ready to commit, all wrong paths are supposed to be selectively squashed. This is possible only when all replicas are created, after the first stage of renaming. The commit stage is not stalled by itself because this could case the whole pipeline to stall, generating a deadlock. So, increase the number of predications and/or anticipating the execution of several instructions are not enough, if they have not enough time to complete the fetch/rename of all paths.

For benchmarks *go* and *cc1*, mainly, more instructions are predicated and, as consequence, many replicas are not being able to be created. For the remaining benchmarks a similar effect may be observed, but with a different cause. In those cases, as value reuse is anticipating the execution of all instructions, predicated branches reach the top of the ROB faster and, again, there is no time left to replicate and tag all instructions. Unfortunately, this effect, caused by either reason, may cause an increase in the misprediction rates, and consequently, a decrease in the performance gain.

9.1 Future Work

There are three main paths to explore for the next steps of this work. They are described in the next Sections.

9.1.1 Reusing Branches

One of the main limitations of this work is that it does not support branch reuse. It is known that reuse mechanisms may even decrease mispredictions, because it can detect a mispredicted branch after the reuse test. This happens because the correct result of a given branch is found when it is reused, just like any other instruction. This may turn a misprediction into a misfetch only, saving several cycles and resources.

In DCE architecture this task is more difficult to be performed. When a branch is data dependent from a previously predicated branch, it has replicas and each replica belongs to a control flow. When the architecture detects that a branch was misfetched, it has to flush and redirect the fetch selectively, according to each path. This would be hard and very expensive.

Nevertheless, there is an optimization that may be done. When all replicas of the data dependent branch were predicted to the same target, the squash, if necessary, would be easier to perform. This will be the next step to be developed in this research.

9.1.2 Trace Reuse Based on a Fixed Stride

The left side of Figure 9.1 shows an example of how outputs are generated in a certain code. A, B and C are the inputs registers, while X and Y are the outputs produced by the trace.

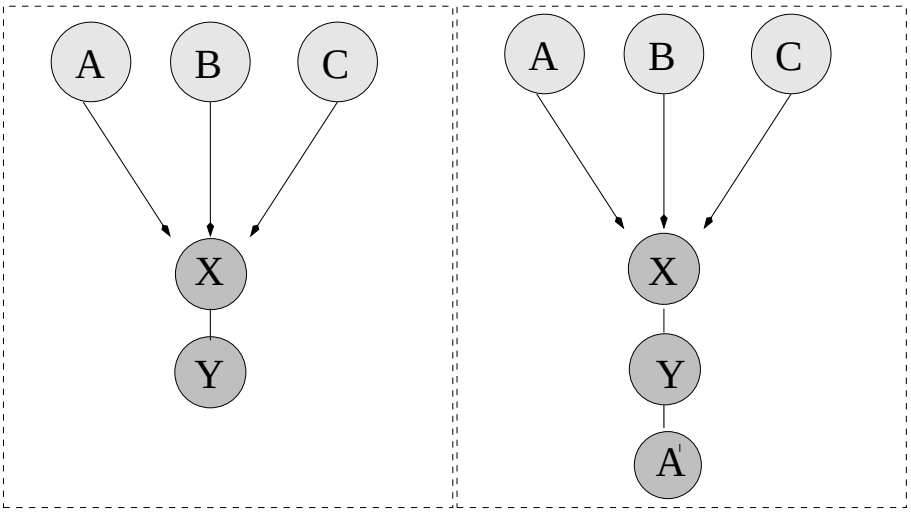

Figure 9.1: Outputs generation

Ideally, inputs A, B and C are never going to change and therefore, the outputs will always be the same. This Figure shows a typical example where trace reuse is going to be very effective, reusing that trace in its next occurrences. Nevertheless, it is very common to find a situation such as the one described in the right side of Figure 9.1. In this case, input A is modified each time that trace is executed,

producing *A'*. This means that next time this trace is fetched it will not be reused because input *A* will be different from the one previously stored in the reuse table.

This phenomenon may be noticed in smaller granularities, such as instruction reuse itself. If an instruction modify one of its source registers, it cannot be reused next time. However, if the difference between the last occurrences is detected, it is possible to adjust the correct value and reuse the instruction.

Figures 9.2 and 9.3 show examples of traces extracted dynamically from the execution of *cc1* benchmark.

Figure 9.2 shows a trace with two instructions. Register *r17* is the input register for this trace, while *r2* and *r17* are its output registers. The trace is repeated four times, meaning that it was fetched and identified as a previously stored trace four subsequent times. And even though the input registers were the same, their values got modified in the previous execution and the trace never got reused. Thus, it is possible to notice that *r17* is changing constantly, according to the earlier occurrence of the trace.

The reuse with stride should apply to cases like this, where is easy to define and adjust the value in the input registers. The idea is to update the reuse table before the next occurrence of the trace. An alternative for this is to update the table just after its access, not in commit stage like is performed usually.

So, when a given instruction is accessed in the reuse table and identified as one which has a stride, the reuse is executed normally but the table is updated accordingly. In Figure 9.2, each time that the first instruction is accessed in the reuse table the value of *r17* will be reused and updated based on the stride (adding 1, in this case). Therefore, in the following access the value will be correct and the reuse will take place.

This initial approach, however, has some restrictions relative mainly to the following:

- The stride must be constant;
- The reuse table is updated according the stride just after its access;
- The input source is modified in previous executions.

The main challenge in this initial approach is to detect the stride correctly and to find the pattern of its occurrence in the trace. The idea is to perform this dynamically, but some compiler support may be needed.

Another challenge is to find other cases where this approach may be used. Figure 9.3 shows an example where there is a constant stride, but none of the input registers are in the output scope.

9.1.3 Out–of–Order Commit

This work has shown that even being executed several cycles before, the average number of cycles that a reused instruction remains in the pipeline is just slightly lower than a conventionally executed instruction.

It is clear that the main problem in reusing instructions only is that even saving resources, reused instructions have to be maintained in the pipeline waiting for previous instructions to be completed. As discussed in previous Chapters, only

```
(TRACE SCOPE): Tag PC: 0x0056c820 : Next PC : 0x0056c830
In size 1 : Out size 2 : Trace size 2
In Scope:
R17: 0x00000001
Out Scope:
R2: 0x00000001
R17: 0x00000002

Instructions in Trace:
 0x0056c820 : 0000000000 : addiu      r17,r17,1      *
 0x0056c828 : 0000000000 : slti       r2,r17,64      *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c820 : Next PC : 0x0056c830
In size 1 : Out size 2 : Trace size 2
In Scope:
R17: 0x00000002
Out Scope:
R2: 0x0000001
R17: 0x00000003

Instructions in Trace:
 0x0056c820 : 0000000000 : addiu      r17,r17,1      *
 0x0056c828 : 0000000000 : slti       r2,r17,64      *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c820 : Next PC : 0x0056c830
In size 1 : Out size 2 : Trace size 2
In Scope:
R17: 0x00000003
Out Scope:
R2: 0x00000001
R17: 0x00000004

Instructions in Trace:
 0x0056c820 : 0000000000 : addiu      r17,r17,1      *
 0x0056c828 : 0000000000 : slti       r2,r17,64      *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c820 : Next PC : 0x0056c830
In size 1 : Out size 2 : Trace size 2
In Scope:
R17: 0x00000004
Out Scope:
R2: 0x00000001
R17: 0x00000005

Instructions in Trace:
 0x0056c820 : 0000000000 : addiu      r17,r17,1      *
 0x0056c828 : 0000000000 : slti       r2,r17,64      *
============================================================
```

Figure 9.2: Trace not reused due to not ready operands

```
(TRACE SCOPE): Tag PC: 0x0056c780 : Next PC : 0x0056c7a0
In size 1 : Out size 1 : Trace size 3
In Scope:
R17: 0x00000006
Out Scope:
R2: 0x00000006

Instructions in Trace:
 0x0056c780 : 0000000000 : addiu    r2,r0,r17
 0x0056c788 : 0000000000 : bgez     r17, 0x0056c798
 0x0056c798 : 0000000000 : sra      r2, r2, 5                    *
============================================================
TRACE SCOPE): Tag PC: 0x0056c780 : Next PC : 0x0056c7a0
In size 1 : Out size 1 : Trace size 3
In Scope:
R17: 0x00000005
Out Scope:
R2: 0x00000005

Instructions in Trace:
 0x0056c780 : 0000000000 : addiu    r2,r0,r17
 0x0056c788 : 0000000000 : bgez     r17, 0x0056c798
 0x0056c798 : 0000000000 : sra      r2, r2, 5                    *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c780 : Next PC : 0x0056c7a0
In size 1 : Out size 1 : Trace size 3
In Scope:
R17: 0x00000004
Out Scope:
R2: 0x00000004

Instructions in Trace:
 0x0056c780 : 0000000000 : addiu    r2,r0,r17
 0x0056c788 : 0000000000 : bgez     r17, 0x0056c798
 0x0056c798 : 0000000000 : sra      r2, r2, 5                    *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c780 : Next PC : 0x0056c7a0
In size 1 : Out size 1 : Trace size 3
In Scope:
R17: 0x00000003
Out Scope:
R2: 0x00000003

Instructions in Trace:
 0x0056c780 : 0000000000 : addiu    r2,r0,r17
 0x0056c788 : 0000000000 : bgez     r17, 0x0056c798
 0x0056c798 : 0000000000 : sra      r2, r2, 5                    *
============================================================
(TRACE SCOPE): Tag PC: 0x0056c780 : Next PC : 0x0056c7a0
In size 1 : Out size 1 : Trace size 3
In Scope:
R17: 0x00000002
Out Scope:
R2: 0x00000002

Instructions in Trace:
 0x0056c780 : 0000000000 : addiu    r2,r0,r17
 0x0056c788 : 0000000000 : bgez     r17, 0x0056c798
 0x0056c798 : 0000000000 : sra      r2, r2, 5                    *
============================================================
```

Figure 9.3: Trace not reused with different output and input scopes

after the completion of these regularly executed preceding instructions is that the reused ones will be delivered to retirement.

In order to reduce this problem an out–of–order commit mechanisms could be developed. These mechanisms are being continuously studied later and they basically allow that instructions are retired from the pipeline earlier, as soon as they are completed (MARTíNEZ et al., 2002; VIJAYAN; RAJENDRAN; VELUSWAMI, 2002; CRISTAL et al., 2004).

The basic idea is to modify the ROB structure, introducing checkpoints in between instructions committed speculatively. These checkpoints are very similar to the ones used in any branch mechanism.

Besides helping in the reuse bottleneck, this kind of mechanism will help to decrease the number of physical registers and ROB positions necessary to support Dynamic Conditional Execution architecture.

10 REUSANDO VALORES EM UMA ARQUITETURA COM EXECUÇÃO CONDICIONAL DINÂMICA

10.1 Introdução

O custo das dependências de controle em arquiteturas superescalares é ainda uma questão de pesquisa em aberto, e técnicas de atenuação dos efeitos negativos destas dependências são o foco de um esforço mútuo de pesquisas realizadas tanto na área acadêmica quanto na indústria de microprocessadores superescalares de propósito geral. Para *pipelines* profundos do estado da arte, em especial, o custo de previsões incorretas é ainda maior em decorrência do grande número de estágios existentes. A execução condicional dinâmica (DCE – *Dynamic Conditional Execution*) é uma nova proposta de arquitetura superescalar que busca reduzir esse problema. A idéia básica é buscar e executar todos os caminhos produzidos por um desvio que obedecem determinadas restrições relacionadas à sua complexidade e tamanho. Como conseqüência, um número menor de previsões é executado e assim, um número menor de desvios é previsto incorretamente.

Além de buscar múltiplos fluxos, a arquitetura DCE necessita produzir diversas réplicas de uma mesma instrução de forma a garantir a semântica correta de dados provenientes dos vários fluxos de execução. Essas réplicas são produzidas após o ponto de convergência dos vários fluxos e continuam sendo criadas pela arquitetura até que o desvio que as originou seja resolvido. Desse modo, uma seção inteira de código pode ser replicada, saturando o *pipeline* e afetando negativamente o desempenho. Uma alternativa natural para amenizar o *overhead* causado pela busca/execução de múltiplos fluxos e a criação de réplicas é o reuso de valores executados previamente. Dessa forma, os recursos da arquitetura podem ser liberados para as demais instruções úteis.

O objetivo principal desse trabalho é analisar o impacto do reuso de valores, em diferentes granularidades, na arquitetura DCE. Esta tese demonstra com extensas simulações e análises de resultados que a abordagem proposta reduz efetivamente o *overhead* produzido pela arquitetura, aumentando o desempenho global.

10.2 O Reuso de Valores na Arquitetura DCE

A ocorrência de instruções de desvio é uma questão cuja solução ainda é indefinida e que limita a extração de paralelismo em nível de instrução (ILP), bem como o desempenho de processadores superescalares. A alternativa para reduzir esse

problema é, em geral, introduzir mecanismos complexos e sofisticados de previsão. Mesmo assim, não há garantias que o fluxo previsto é o correto a seguir e o *pipeline* sofre uma alta penalidade quando uma previsão errada é observada.

Além disso, essa penalidade tem crescido enormemente com *pipelines* cada vez mais profundos. A média de acerto em preditores do estado-da-arte é maior que 90%, mas a ocorrência de 10% ou menos em previsões erradas é suficiente para reduzir drasticamente o desempenho, medido pelo numero de instruções executadas nas aquiteturas de microprocessadores superescalares.

A Execução Condicional Dinâmica (DCE) é uma nova abordagem para redução de previsões incorretas. A idéia é diminuir o número de previsões erradas pela redução do número requerido de previsões, predicando desvios em alguns casos específicos. O compilador é responsável em qualificar as estruturas de desvios (*hammocks*) que podem ser predicadas. A proposta do DCE é, então, que o microprocessador decide, em tempo de execução, quais dos *hammocks* qualificados serão realmente predicados.

É claro que apenas parte dos desvios condicionais existentes em um dado programa pode ser predicada. De outro modo, o *pipeline* de instruções saturaria rapidamente e a técnica não apresentaria nenhum ganho de desempenho. Por essa razão, o compilador marca estaticamente apenas *hammocks* que obedecem algumas restrições relativas, principalmente, ao seu tamanho e complexidade. Tipicamente, a arquitetura DCE qualifica cinco tipos de *hammocks* para predicação:

- Simples (um e dois lados): desvios sem estruturas aninhadas;
- Complexos puros: uma ou mais estruturas de desvio aninhadas totalmente contidas na mais externa;
- Complexos com múltiplas convergências: uma ou mais estruturas de desvio aninhadas que possuem os alvos coincidentes com o ponto de convergência;
- Complexos com múltiplos alvos: uma ou mais estruturas de desvio aninhadas que possuem seus alvos depois do ponto de convergência;
- Complexos sobrepostos: uma ou mais estruturas de desvio que possuem seus alvos localizados no caminho *tomado* da estrutura mais externa.

Estruturas de desvios contendo desvios *backward*, desvios indiretos, desvios incondicionais não relacionados com o controle de fluxo do próprio desvio, chamadas de sistema e/ou chamadas/retornos de sub-rotinas desqualificam estes desvios para predicação.

Quando um desvio qualificado é buscado, a arquitetura decide, baseada na disponibilidade de recursos, se aquele *hammock* será predicado. Se a arquitetura decide não predicar o desvio qualificado, então o mesmo é previsto normalmente. Por outro lado, quando um desvio é predicado, ambos os caminhos são buscados de modo a evitar uma previsão. Todos os fluxos serão buscados e executados e o caminho correto é graduado, enquanto a arquitetura descarta seletivamente os fluxos errados.

O problema, nesse caso, é manter a consistência entre as diferentes correntes de dados, provenientes dos diferentes fluxos de execução. É com essa finalidade que a arquitetura cria várias réplicas da mesma instrução. Assim, quando o ponto de um *hammock* predicado é encontrado, a arquitetura introduz uma réplica para cada

fluxo de cada nova instrução buscada. As réplicas continuam sendo criadas até que o resultado do desvio que originou a predicação seja conhecido, ou seja, até que a execução do desvio original seja completada.

As simulações executadas em trabalhos anteriores mostraram que a ocorrência de previsões erradas reduziu significantemente com a utilização da arquitetura DCE (SANTOS, 2003). O *speedup* sobre uma arquitetura superescalar convencional, contudo, não foi tão significante e em alguns casos não houve ganho algum. Mesmo selecionando e qualificando apenas parte dos desvios, o *pipeline* ainda satura, especialmente, após o estágio de renomeação onde as réplicas são criadas.

A intenção desse trabalho é combinar reuso de valores e predicação dinâmica de instruções, avaliando o desempenho dessa nova abordagem. Conseqüentemente, os seguintes pontos podem ser considerados como as maiores contribuições dessa tese de doutorado:

- Avaliar o impacto do *overhead* causado pela arquitetura DCE;
- Propor uma alternativa que reduza esse problema efetivamente (reuso de valores);
- Validar a idéia através da implementação de um simulador de microprocessadores detalhado;
- Analisar todos os resultados atingidos, identificando os prós e contras de tal abordagem.

Dentre outras alternativas consideradas inicialmente, o reuso de valores foi identificado como o mais adaptável ao contexto DCE. Assim, o principal objetivo desse trabalho foi introduzir a idéia de reuso de valores na arquitetura DCE. Tipicamente, arquiteturas multi-fluxo e predicação requerem uma grande estrutura de execução. Os estágios de *pipeline* funcionam perto do seu limite e paradas devido ao grande número de instruções produzidas a partir dos múltiplos fluxos são comuns.

Primeiramente, apenas o reuso de instruções foi estudado. A idéia foi desenvolver um mecanismo para armazenar todas as instruções previamente graduadas e reusar cada instrução unitariamente, quando possível. Uma instrução pode ser reusada quando todos os seus operandos de origem estão prontos e correspondem aos que estão armazenados no *Buffer* de Reuso (RB). Quando isso acontece, o registrador de destino é assinalado com o valor contido no RB e marcado como pronto. Isso ocorre imediatamente após a renomeação de registradores, em um estágio novo criado para acomodar o teste de reuso e a atribuição do valor destino. A instrução reusada é então encaminhada para o *buffer* de reordenamento, onde aguarda pela graduação de resultados.

Um simulador baseado no simulador *sim-dce* foi desenvolvido, permitindo reuso de instruções na arquitetura DCE. Nessa primeira parte do trabalho, o *pipeline* DCE foi estendido para simular *pipelines* profundos bem como para reusar instruções singularmente. O mecanismo implementado segue a mesma idéia acima descrita.

As simulações do reuso de instruções mostraram que a média de redução de *overhead* está entre 13% e 16%, dependendo do tamanho do RB, para todos os *benchmarks* utilizados. Isso significa que a arquitetura está executando de 13 a 16% menos instruções. A média harmônica para o *speedup*, contudo, não foi maior que 2,5%, comparando com a arquitetura DCE original. Além disso, as taxas mais

significativas foram atingidas pelo *benchmark perl*, em torno de 10%. *Benchmarks cc1* e *mgrid* obtiveram os piores ganhos de desempenho e a melhora não foi maior que 0,5%.

Apesar das instruções reusadas não serem executadas, elas continuam ocupando entradas no *buffer* de reordenamento. Além disso, essas instruções precisam esperar por todas as instruções antecessoras, pois a graduação é realizada em ordem. Essa é a principal limitação dessa abordagem, já que o gargalo é apenas transferido da execução para a graduação de instruções.

A segunda parte do trabalho permitiu o reuso de traços. Um traço é a seqüência de n instruções contíguas que podem ser reusadas de uma única vez. Isso significa que, baseado em um único teste de reuso, várias instruções podem ser reusadas. Nesse caso, um novo *buffer* é necessário para armazenar os contextos de entrada e saída dos traços, usados para identificar quando um traço pode ser reusado ou não. O contexto de entrada é o conjunto de registradores utilizado como operandos de origem por todas as instruções de um dado traço. Por outro lado, o contexto de saída é o conjunto de registradores usado como destino por todas as instruções de um traço.

O teste de reuso é realizado através da comparação de todos os registradores que fazem parte do contexto de entrada, e não apenas dos operandos de origem, como no caso do reuso de instruções original. Similarmente, todos os registradores que fazem parte do contexto de saída são assinalados de uma vez, resolvendo várias instruções simultaneamente.

Uma vez que o *Buffer* de Traços (TB) sempre armazena o último valor produzido por um registrador pertencente ao contexto de saída, algumas instruções podem ser diretamente retiradas do *pipeline*, sem precisar esperar por todas as instruções antecessoras no *buffer* de reordenamento. Uma instrução que faz parte de um traço será encaminhada para o ROB apenas se a mesma produzir um resultado que poderá ser utilizado pelas instruções subseqüentes. Desse modo, todas as instruções que produzem valores que serão consumidos dentro do próprio traço são tiradas do *pipeline*. Isso reduz o problema de saturação de entradas do ROB, observado quando reusando instruções unitariamente.

Na arquitetura DCE, vários traços podem ser formados paralelamente, já que também existem vários fluxos em curso. A idéia é reduzir ainda mais o *overhead* de instruções, já que diferentes traços de réplicas estão disponíveis para reuso.

Para validar essa idéia, um mecanismo de reuso de traços foi desenvolvido tomando como base o já estendido *sim-dce*. Os traços são construídos no estágio de graduação e armazenados no TB. Quando traços de réplicas são construídos, são armazenados nos diversos blocos do mesmo conjunto do *Buffer* de Traços. O teste de reuso é aplicado ainda no estágio de renomeação de registradores de forma a prevenir renomeações desnecessárias. Os registradores lógicos usados por instruções que serão retiradas não devem ser renomeados, liberando recursos para outras instruções úteis.

Todas as instruções que não fazem parte de um traço sendo reusado continuam sendo buscadas no *Buffer* de Reuso de instruções (RB). Então, instruções que não são parte de um traço válido, ainda podem ser reusadas através do mecanismo de reuso simples. O teste de reuso é realizado imediatamente após a renomeação, assim como discutido anteriormente.

As simulações realizadas no decorrer desta pesquisa de tese mostraram que a

redução de *overhead* foram entre 8% e 10% maiores, quando comparadas com os resultados produzidos pelo reuso simples de instruções. Além disso, as instruções que foram removidas do *pipeline* não foram contabilizadas nessa estatística e é possível afirmar que essas taxas de redução de *overhead* são ainda maiores, considerando tais instruções.

A média de tamanho dos traços efetivamente reusados é também um dado importante sobre o qual foram realizadas análises. Apesar da implementação atual não permitir reuso de desvios, a média de tamanho é de cerca de 2,5 instruções. Essa média é similar à atingida em trabalhos anteriores (COSTA, 2001) que incluíam reuso de desvios. Provavelmente, quando desvios forem reusados junto da arquitetura DCE, essa média será ainda maior, como cogitado inicialmente. Já que a arquitetura DCE possui várias réplicas sendo geradas após cada ponto de convergência, até que o desvio original seja resolvido, os traços são potencialmente maiores.

O *speedup* alcançado pelo mecanismo de reuso de traços sobre o DCE original varia largamente, sobretudo, de acordo com o *benchmark* simulado. Para o *benchmark go*, por exemplo, o reuso de traços atingiu mais de 80% de ganho em algumas configurações. Os *benchmarks applu*, *cc1* e *turb3d* também atingiram ganhos significativos de desempenho. Contudo, em outros casos, tais como *mgrid* e *perl*, foi observado um ganho mínimo de desempenho. Assim como os *benchmarks*, as configurações simuladas também afetam o desempenho. Em geral, tabelas de reuso de conjunto associatividade 4 atingem melhores resultados. Possivelmente, nesses casos, mais traços de réplicas estão disponíveis para as diferentes correntes de dados, aumentando o desempenho global.

Os efeitos colaterais nas características mais marcantes da arquitetura DCE foram também analisados. Primeiramente, o número de desvios dinamicamente predicados pela arquitetura foi observado. Após aumentar o número de tabelas de mapeamento, apenas os *benchmarks cc1* e *go* apresentaram um aumento significativo no número de predicações. Para esses *benchmarks* o reuso de valores permitiu que um número maior de fluxos fosse criado. Nesses casos, é possível afirmar que esse aumento no número de predicações afetou positivamente o desempenho final. Como visto anteriormente, o *benchmark go* atingiu cerca de 82% de ganho em algumas configurações. Esse ganho não é conseqüência exclusiva do número de predicações realizadas, mas esse índice certamente afetou o desempenho.

Por outro lado, no *benchmark turb3d*, uma grande redução no número de desvios predicados foi observada. Já que mais desvios estão sendo previstos, as taxas de previsões erradas tendem a ser maiores. Isso explica porque esse *benchmark* apresentou as maiores taxas de redução de *overhead*, mas não alcançou os mesmos resultados no desempenho global.

Como já discutido anteriormente, a ocorrência de previsões erradas é tipicamente menor na arquitetura DCE, quando comparada a um processador superescalar convencional. Isso acontece porque um número menor de previsões está sendo realizado. Contudo, existe um caso especial em que a predicação pode ser tratada da mesma forma que uma previsão errada. Um desvio predicado é tratado como previsto incorretamente quando o mesmo atinge o topo do *buffer* de reordenamento e ainda existem réplicas que não foram renomeadas. Isso acontece porque, quando um desvio predicado está pronto para ser graduado, todos os fluxos incorretos devem ser descartados seqüencialmente. Para que isso aconteça, todas as réplicas já devem ter sido criadas e etiquetadas, ou seja, todas as instruções relacionadas com a predicação

já devem ter passado pelo primeiro estágio de renomeação. O estágio de graduação não pára porque isso poderia ocasionar a parada total do *pipeline*, gerando um *deadlock*. Desse modo, é possível afirmar que aumentar o número de desvios predicados e/ou antecipar a execução de diversas instruções não é suficiente, caso não haja tempo hábil para completar busca/renomeação de todos os fluxos.

No caso dos *benchmarks go* e *cc1*, principalmente, um número maior de predicações está sendo realizado e muitas dessas réplicas não estão sendo criadas a tempo. Para os demais *benchmarks*, o mesmo efeito pode ser observado, mas com uma causa diferente. Nesses casos, o reuso de valores está antecipando a execução de todas as instruções e os desvios predicados chegam ao topo do *buffer* de reordenamento mais rapidamente, prevenindo o etiquetamento de todas as instruções necessárias. Infelizmente, isso pode causar um aumento na taxa de previsões incorretas e, conseqüentemente, um decréscimo no ganho de desempenho.

10.3 Sumário das Conclusões

Essa tese de doutorado teve como principal objetivo estudar o problema do *overhead* ocasionado pela busca/execução de múltiplos fluxos em uma arquitetura com execução condicional dinâmica, propondo um mecanismo de reuso de instruções e traços para reduzir esse problema.

A metodologia adotada para atingir esse objetivo foi desenvolver, inicialmente, um mecanismo de reuso de instruções. As simulações executadas nessa primeira fase, no entanto, apontaram um acréscimo muito baixo de desempenho, mesmo com uma média de cerca de 15% de instruções reusadas. O problema é que as instruções reusadas continuam ocupando recursos no *pipeline* do microprocessador superescalar com suporte para execução fora-de-ordem. Mesmo liberando unidades funcionais, essas instruções são mantidas no *buffer* de reordenamento para esperar a graduação de resultados em ordem.

Esse problema, porém, tende a ser menor quando a arquitetura realiza o reuso de traços, implementado na segunda fase deste trabalho de pesquisa. Instruções que fazem parte do traço, mas não produzem resultados que serão usados por instruções subseqüentes, podem ser retiradas imediatamente do *pipeline*, reduzindo o número efetivo de instruções no *buffer* de reordenamento. Desse modo, os resultados apresentados pelo mecanismo de reuso de traços foram melhores, se comparados com os produzidos pelo reuso de instruções puramente. Em alguns casos, o reuso de traços apresentou ganhos de mais de 80% sobre o DCE original.

Como trabalhos futuros, pode-se destacar, sobretudo, a implementação de reuso de desvios, a implementação da graduação fora-de-ordem assim como o estudo de mecanismos para o reuso de *stride* fixo.

REFERENCES

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

BODIK, R.; GUPTA, R.; SOFFA, M. L. Load-Reuse Analysis: Design and Evaluation. In: SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.64–76.

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

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.

CITRON, D.; FEITELSON, D. Revisiting Instruction Level Reuse. In: WORKSHOP ON DUPLICATING, DECONSTRUCTING AND DEBUNKING, 2002. **Proceedings...** [S.l.: s.n.], 2002.

COSTA, A. T. da. **Explorando Dinamicamente o Reuso de Traces em Nivel de Arquitetura de Processador**. 2001. Tese de Doutorado — COPPE–UFRJ.

COSTA, A. T. da; FRANÇA, F. M. G. **The Reuse Potencial of Trace Memoization**. Rio de Janeiro: COPPE–UFRJ, 1999. Technical Report. (ES-498/99).

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. Exploiting Reuse with Dynamic Trace Memoization: Evaluating Architectural Issues. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 12., 2000, São Pedro. **Proceedings...** São Paulo: SBC, 2000. p.163–172.

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. The Dynamic Trace Memoization Reuse Technique. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2000, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.92–99.

CRISTAL, A.; ORTEGA, D.; LLOSA, J.; VALERO, M. Out–of–Order Commit Processors. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 10., 2004, Madrid. **Proceedings...** Los Alamitos: IEEE Computer Society, 2004.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. **The Performance Potential of Data Value Reuse**. Barcelona: Universitat Politécnica de Catalunya, 1998. Technical Report.

GONZALEZ, A.; TUBELLA, J.; MOLINA, C. Trace-Level Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 28., 1999, Aizu-Wakamatsu. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.30–37.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: A Quantitative Approach. 3rd ed. San Francisco: Morgan Kaufmann, 2003.

HENNING, J. L. SPEC CPU2000: Measuring CPU Performance in the New Millenium. **IEEE Computer**, Los Alamitos, v.33, n.7, p.28–35, July 2000.

HOREL, T.; LAUTHERBACH, G. UltraSparc-III: Designing Third-Generation 64-Bit Performance. **IEEE Micro**, Los Alamitos, v.19, n.13, p.73–85, May/June 1999.

HUANG, J.; CHOI, Y.; LILJA, D. **Improving Value Prediction by Exploiting Both Operand and Output Value Locality**. [S.l.]: Laboratory for Advanced Research in Computing Technology and Compilers, 1999. Technical Report. (ARCTic 99-06).

HUANG, J.; LILJA, D. **Exploiting Basic Block Value Locality with Block Reuse**. [S.l.]: Laboratory for Advanced Research in Computing Technology and Compilers, 1998. Technical Report. (HPPC 98-09).

HUANG, J.; LILJA, D. J. Exploiting Basic Block Value Locality with Block Reuse. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 5., 1999, Orlando. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.106–114.

HUANG, J.; LILJA, D. J. Exploring Sub-Block Value Reuse for Superscalar Processors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2000, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000.

HUANG, J.; LILJA, D. J. Extending Value Reuse to Basic Blocks with Compiler Support. **IEEE Transactions on Computers**, New York, v.49, n.4, p.331–347, Apr. 2000.

INTEL. **Inside the NetBurst Micro-Architecture of the Intel Pentium 4 Processor**. Disponível em: <http://download.intel.com/pentium4/download/netburst.pdf>. Acesso em: janeiro 2001.

JOHNSON, M. **Superscalar Microprocessor Design**. Englewood Cliffs: Prentince Hall, 1991.

KELLER, R. M. Look-Ahead Processors. **ACM Computing Surveys**, New York, v.7, n.4, p.177–195, 1975.

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.

MARTíNEZ, J.; RENAU, J.; HUANG, M.; PRVULOVIC, M.; TORRELLAS, J. Cherry: Checkpointed Early Resource Recycling in Out-Of-Order Microprocessors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 35., 2002, Istanbul. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002.

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

NEMIROVISKY, M. **DISC**: A Dynamic Instruction Stream Computer. 1990. PhD Thesis — University of California, Santa Barbara, USA.

ONDER, S.; GUPTA, R. Load and Store Reuse Using Register File Contents. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 15., 2001, Sorrento, Italy. **Proceedings...** New York: ACM, 2001. p.289–302.

PILLA, M. L. **RST**: Reuse Through Speculation on Traces. 2004. PhD Thesis — PPGC–UFRGS.

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; COSTA, A. T. da. **Speculative Trace Reuse**. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2002. Technical Report. (RP–320).

POSTIFF, M. A.; GREENE, D. A.; THYSON, G. S.; MUDGE, T. N. The limits of Instruction Level Parallelism in SPEC95 Applications. **Computer Architecture News**, [S.l.], v.217, n.1, p.31–34, 1999.

ROTENBERG, E.; SMITH, J. Control Independence in Trace Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 32., 1999, Haifa. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.4–15.

SANTOS, R. R. dos. **DCE**: The Dynamic Conditional Execution in a Multipath Control Independent Architecture. 2003. PhD Thesis — PPGC–UFRGS.

SANTOS, R. R. dos; NAVAUX, P. O. A. Analysing a Multistreamed Superscalar Speculative Instruction Fetch Mechanism. In: INTERNATIONAL EUROPAR CONFERENCE ON PARALLEL PROCESSING, 4., 1998, Southampton. **Proceedings...** Berlin: Springer-Verlag, 1998. p.1010–1017.

SANTOS, R. R. dos; SANTOS, T. G. S. dos; PILLA, M. L.; NEMIROVSKY, M.; BAMPI, S.; NAVAUX, P. O. A. Complex Branch Profiling for Dynamic Conditional Execution. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 15., 2003, São Paulo, SP. **Proceedings...** São Paulo: USP, 2003. p.279–286.

SANTOS, T. G. S. dos. **Análise do Comportamento de Mecanismos de Pré-busca em Memórias Hierárquicas de Microprocessadores RISC Superescalares**. 2000. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SANTOS, T. G. S. dos; SANTOS, R. R. dos; NEMIROVSKY, M.; ; BAMPI, S.; NAVAUX, P. O. A. Analyzing the Limits of Trace Reuse in a Dynamic Conditional Execution Architecture. In: ACM WORKSHOP ON EXPLORING THE TRACE SPACE FOR DYNAMIC OPTIMIZATION TECHNIQUES, 2003, San Francisco. **Proceedings...** [S.l.: s.n.], 2003.

SASTRY, S.; BODIK, R.; SMITH, J. Characterizing Coarse–Grained Reuse of Computation. In: ACM WORKSHOP ON FEEDBACK-DIRECTED AND DYNAMIC OPTIMIZATION, 2000. **Proceedings...** [S.l.: s.n.], 2000.

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.

SODANI, A. **Dynamic Instruction Reuse**. 2000. PhD Thesis — University of Wisconsin, Madison.

SODANI, A.; SOHI, G. S. Dynamic Instruction Reuse. **Computer Architecture News**, New York, v.25, n.2, May 1997.

SODANI, A.; SOHI, G. S. Dynamic Instruction Reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24., 1997. **Proceedings...** New York: ACM, 1997.

SODANI, A.; SOHI, G. S. Understanding the Differences Between Value Prediction and Instruction Reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 31., 1998. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998.

THORNTON, J. E. Parallel Operation in the Control Data 6600. **AFIPS Fall Joint Computer Conference**, [S.l.], v.26, n.2, 1964.

TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. **IBM Journal of Research and Development**, Armonk, v.11, n.1, Jan. 1967.

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.

VIJAYAN, B.; RAJENDRAN, M.; VELUSWAMI, S. Out–of–Order Commit Logic with Precise Exception Handling for Pipelined Processors. In: INTERNATIONAL HIGH–PERFORMANCE COMPUTING CONFERENCE, 2002, Bangalore. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002.

WALLACE, S.; TULLSEN, D. M.; CALDER, B. Instruction Recycling on a Multiple-Path Processor. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 5., 1999, Orlando. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999.

WU, Y.; CHEN, D.-Y.; FANG, J. Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 28., 2001, Göteborg, Sweden. **Proceedings...** New York: ACM, 2001.

YANG, J.; GUPTA, R. Load Redundancy Removal through Instruction Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 29., 2000, Toronto. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.61–68.

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.