MODEE : smoothing branch and instruction cache miss penalties on deep pipelines

Nathalie Drach, André Seznec

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MIDEE: Smoothing Branch and Instruction Cache Miss Penalties on Deep Pipelines

Nathalie Drach, André Seznec

N° 2038

septembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

MIDEE: Smoothing Branch and Instruction Cache Miss Penalties on Deep Pipelines

Nathalie Drach, André Seznec

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet CALCPAR

Rapport de recherche n ° 2038 — septembre 1993 — 20 pages

Abstract: Pipelining is a major technique used in high performance processors. But a fundamental drawback of pipelining is the lost time due to branch instructions.

A new organization for implementing branch instructions is presented: the Multiple Instruction Decode Effective Execution (MIDEE) organization. All the pipeline depths may be addressed using this organization. MIDEE is based on the use of double fetch and decode, early computation of the target address for branch instructions and two instruction queues. The double fetch-decode concerns a pair of instructions stored at consecutive addresses. These instructions are then decoded simultaneously, but no execution hardware is duplicated; only useful instructions are effectively executed. A pair of instruction queues are used between the fetch-decode stages and execution stages; this allows to hide branch penalty and most of the instruction cache misses penalty.

Trace driven simulations show that the performance of deep pipeline processor may dramatically be improved when the MIDEE organization is implemented: branch penalty is reduced and pipeline stall delay due to instruction cache misses is also decreased.

Key-words: branch instruction penalty, deep pipeline, MIDEE organization, instruction cache miss

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

MIDEE: Réduction des pénalités de branchements et de défauts de cache instruction sur les pipelines profonds

Résumé : La technique du pipeline est utilisée dans les processeurs à haute performance. Mais les aléas de contrôle dus aux instructions de branchement peuvent dégrader sensiblement les performances du pipeline.

Dans ce rapport, nous présentons une nouvelle organisation destiner à réduire les pénalités dues aux instructions de branchement: l'organisation MIDEE, Multiple Instruction Decode Effective Execution. Cette organisation peut être mise en oeuvre pour toutes les profondeurs de pipeline. Elle est basée sur le chargement et décodage de deux instructions consécutives, sur le calcul en avance de l'adresse cible du branchement et sur l'utilisation de deux files d'instructions. Deux instructions sont décodées simultanément, mais aucun ajout matériel n'est nécessaire pour l'exécution; en effet seules les instructions valides entrent dans les étages d'exécution. Deux files d'instructions sont utilisées entre les étages de chargement et décodage et les étages d'exécution; ceci permet de réduire les penalités de branchement, mais aussi les penalités de défaut de cache instruction.

Les simulations réalisées montrent que les performances pour les pipelines profonds peuvent être sensiblement améliorées lorsque l'on utilise l'organisation MIDEE: les pénalités de branchement sont réduites et le nombre de gels du pipeline dus aux défauts de cache instruction diminue.

Mots-clé : pénalité de branchement, pipeline profond, organisation MIDEE, défaut de cache instruction

1 Introduction

As pipeline length increases, dependencies between instructions, particularly branch instructions, may affect computer performance. Around 20 % of all instructions are reported to be branches [5, 18], and one third of the branch instructions is conditional. These conditional branch instructions are becoming a major obstacle to the increasing of RISC processor performance, because they can break the smooth flow of instruction execution; the issuing of instructions after a branch instruction must often wait until the condition is resolved.

Since conditional branch instructions can severely degrade the performance, many methods have been proposed to address this problem. Techniques such as *branch prediction* strategies [18] have been used in order to reduce the penalty imposed by branch instructions. These strategies reduce delay by attempting to predict the direction that a branch instruction will take: they are divided in *static prediction* strategies [14, 18] (a privilege direction is statically chosen at compile time) and *dynamic prediction* strategies [13, 14, 18, 21] (previously taken directions are used in order to predict dynamically future directions). Such techniques can produce a high percentage of correct guesses, but their success is highly dependent on the application [5]. In order to avoid losing cycles on false predictions, *delayed branches* [1, 4] have been implemented in many architectures [11]. The compiler can exploit these *delayed branches* by reordering instructions. Unfortunately, filling one or few delay slots with useful instructions is not always possible (for the GCC, TEX and SPICE benchmarks in [5], a one delay slot is filled with a useful instruction in 45% to 50 % of the cases). Many other techniques [1, 3, 8, 4, 12, 20] have been proposed to reduce penalty due to conditional branch instructions.

A trend in recent RISC microprocessor implementations is to have deeper and deeper pipelines (8 stages in the MIPS R4000 instead of 5 stages in the MIPS R3000 [11]). We propose a new pipeline technique, the **MIDEE** (Multiple Instruction Decode Effective Execution), which reduces substantially the negative effect due to branch instructions and which may be implemented at a relatively low hardware cost for deep pipelines. In this paper, we restrict our study on single issue RISC processors.

The **MIDEE** organization is presented in section 2. It is based on the use of double fetch and decode as in [12], early computation of the target address and two instruction queues. The double fetch-decode, contrary to the technique presented in [12], concerns two consecutive address instructions; they are fetched and then decoded simultaneously. In this organization only useful instructions are effectively executed. Then no execution hardware (ALU, floating point ALU, register files ...) is duplicated, and no complex hardware mechanism is needed for nullifying unuseful instructions. The unuseful prefetched and decoded in advance, are discarded before entering the execution stage. A quite similar mechanism was first proposed in [7] by A. Gonzales, J.M. Llaberia and J. Cortadella, but it was only studied for 4 stages pipelines; the **MIDEE** organization

is proposed for all the pipeline depths. A pair of instruction queues are used between the fetch-decode stages and execution stage; this allows to hide the pipeline stalls during instruction cache misses since the instructions are fetched in instruction queues, while the cache miss is resolved.

Trace driven simulations presented in section 3 show that the performance of deep pipeline processors may dramatically be improved when the MIDEE organization is implemented: branch penalty is reduced and pipeline stall delays due to instruction cache misses are decreased.

2 MIDEE: Multiple Instruction Decode Effective Execution

The ultimate goal of this work is to allow branches take zero execution cycles and to decrease the impact of the instruction cache misses on performance. As our simulation tools [6] are linked to the Sparc instruction set [19], we shall consider this instruction set in the remaining paper[1]. First, we examine the branch formats for the instruction set of Sparc [19]. We then describe the design of the **MIDEE** organization: double instruction fetch and decode, early computation of the target address for branch instructions and a pair of instruction queues.

This organization has been proposed for deep pipelines, but in this section for more simplicity, we often illustrate our mechanism on a 5 stages pipeline.

2.1 Branch instructions

We distinguish two families of conditional branch instructions depending on whether or not a register value is needed for the target computation. A branch instruction for which no register value is needed, will be referred to a *no register branch instruction* and other branch instructions will be referred to the *register branch instructions*. When a *no register branch instruction* is issued, the target address can be computed very early in the pipeline, i.e before the preceding instructions are completed. This target address only depends on the program counter and a constant displacement coded in the instruction. But when a *register branch instruction* is issued, the accurate target address can only be computed after the computation of the requested register value is completed.

We consider the instruction set of Sparc [19]. In this instruction set, the two families of conditional branch instructions respectively include:

- *no register branch instruction*: Bicc (Branch Integer Condition Codes), CALL (call subroutine), CBccc (Branch on Coprocessor Condition Codes), FBfcc (Branch on Floating-Point Condition Codes).

[1] The mechanisms presented in the paper are also valid for other instruction sets.

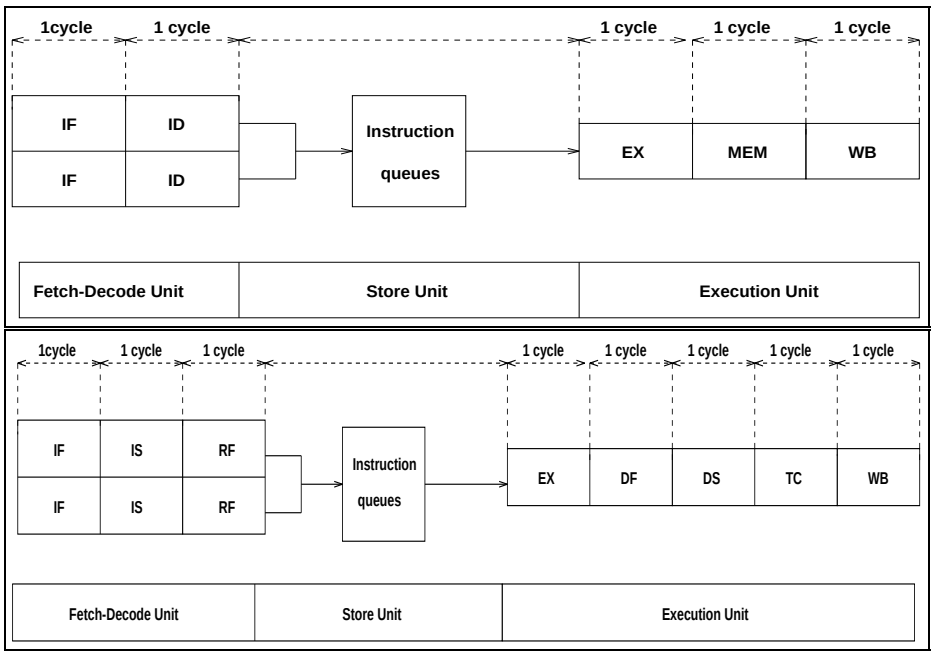

Figure 1: **MIDEE** organization for MIPS R3000 and MIPS R4000 pipeline structure

- *register branch instruction*: JMPL (Jump and Link) is used to return from a subroutine and can be used sometimes as a call subroutine.

Two distinct mechanisms will be used in order to treat these two families of branch instructions. Remark that only conditional branches are *no register branch instructions*.

2.2 MIDEE organization

In this section, we present our pipeline organization for a single issue pipeline processor; it can be used with an any number of stages pipeline. We distinguish three main units in the processor much the same way the architecture CRISP [2] and the mechanisms in [8, 9]: one fetch and decode unit, one store unit and one execution unit. Figure 1 illustrates our pipeline organization, corresponding to the MIPS R3000 pipeline organization and the MIPS R4000 deeper pipeline organization [11].

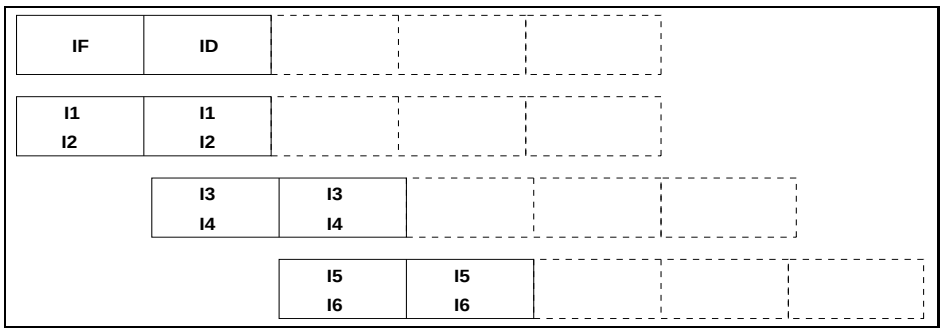

Figure 2: Fetch-decode unit with sequential instructions

2.2.1 Fetch-decode unit

The basis of our hardware mechanism is the double fetch-decode before the execution stage. Two instructions stored at consecutive addresses are simultaneously fetched, then decoded. Notice that the instruction cache has not to be dual-ported as in [12], but that the bus width has to be a double word. This is not a major hardware constraint since instruction caches are now on-chip. The mechanism is plotted in Figure 2 for 6 consecutive address instructions {I1, I2, I3, I4, I5, I6}. When a branch instruction is fetched, it is detected in the decode stage, then the target address can be computed in advance for a *no register branch instruction* (see Figure 3). This target address computation does not request an extra register port on the register files (no computation with register value). In Figure 3, we illustrate this fetch and decode mechanism; while the branch instruction B is not decoded, two consecutive address sequential instructions I2, I3 are fetched. Then when the branch instruction is detected in decode stage, the two consecutive address target instructions T1, T2 are fetched and so on until the branch instruction is executed; the choice is then made between sequential and target path. Figure 4 shows the alternative fetch-decode between the sequential and target path on a 8 stages pipeline.

2.2.2 Instruction queues and smoothing instruction cache miss penalties

A pair of instruction queues, the sequential instruction queue and the target instruction queue, are used in order to store decoded instructions before they are executed. The sequential queue is used for normal sequential fetching and the target queue for storing the target instructions in sequence. Unlike on the IBM RS6000 [10, 16], the sequential queue and the target queue can be swapped.

After a taken branch, the target queue becomes the sequential queue and the original sequential queue becomes free to be the target queue for the next branch. When the branch instruction is executed, the chosen queue is either the

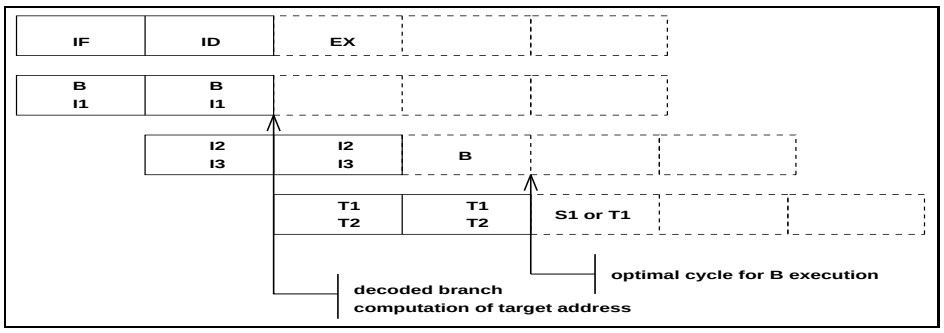

Figure 3: Fetch-decode unit with a branch instrution on a 5 stages pipeline

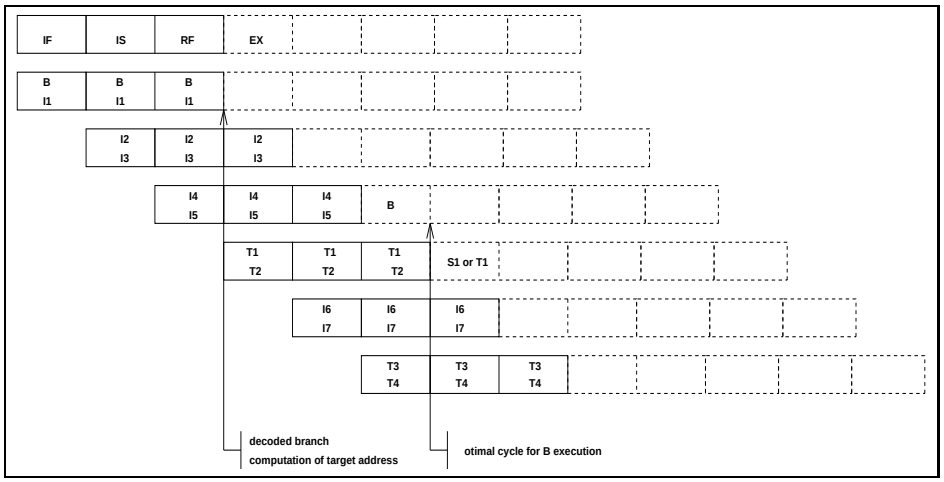

Figure 4: Fetch-decode unit with a branch instrution on a 8 stages pipeline

sequential queue if the branch is untaken or the target queue if the branch is taken: the content of the untaken queue is then discarded and no unuseful instruction enters the execution phase. At each cycle, only one queue is connected to execution stage. Notice that the write and read on the instruction queues does add any cycle in pipeline: when the instruction queue is empty, the instruction flowing out from the decode stage may bypass the instruction queue (as a classical bypass on register files).

The instruction queues also allow to hide the instruction cache miss penalty. Most of the delays due to instruction cache misses are reduced because the pipeline is not stalled as long as the instruction queues are not empty. Figure 5 shows the effect of instruction queues on the number of pipeline stalls due to instruction cache misses. If the target address for a branch instruction misses, the fetch and decode for the target path are stopped until branch is executed.

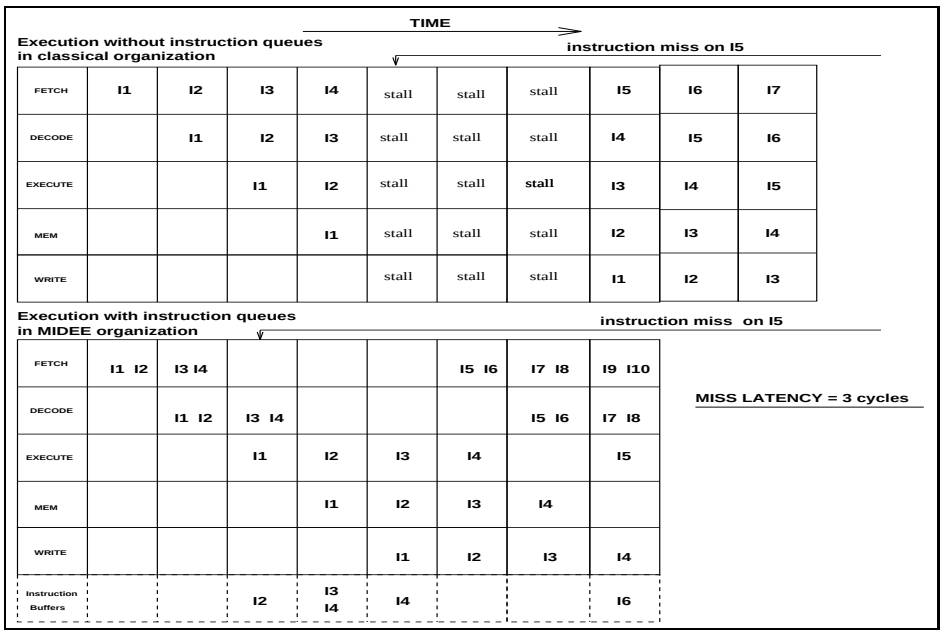

Figure 5: Instruction queues for smoothing miss penalty

Then only if the branch is taken, the target instructions are readed on secondary cache or main memory.

Instruction queues size has an effect on performance, this will be studied in section 3; it depends on the number of fetch-decode stages and the main memory or secondary cache latency.

2.2.3 Effective execution

Only useful instructions are executed; then no hardware associated with a pipeline stage below the decode stage has to be duplicated. In general, the result of the conditional branch instruction will be determined after that the sequential instruction and the target instruction have both been decoded, and of course before they reach the execution stage of the pipeline. When the branch condition is resolved, one of the two flows of instructions is chosen: in favorable cases, the instruction to be executed is already in the corresponding instruction queue and no branch delay penalty is paid.

2.2.4 Smoothing no register branch instruction penalties

In order to explain the general low penalty of a *no register branch instruction*, we introduce a new definition: the *inter branch delay*.

Definition 2.1 *An inter branch delay is the number of instructions between two consecutive branch instructions.*

The penalty on many branches can be reduced to zero, if between the decoding and execution of a *no register branch instruction*, there are instructions alive in both instruction queues. The instruction queues will be both not empty if the *inter branch delay* is greater or equal to the number of fetch-decode stages. On the other hand, if the *inter branch delay* is lower than the number of fetch-decode stages, a taken branch instruction will induce some penalty. When a branch instruction is pending, the decoding of a new branch instruction will stall the decode and prefetch stages until the previous branch is resolved; otherwise other instruction queues would be needed. More complex schemes for resolving these cases may be proposed, but more hardware would be needed (e.g. some other instruction queues).

Figure 6 illustrates the execution of a *no register branch instruction* using our mechanism for a 5 stages pipeline similar to the MIPS R3000 pipeline. In this example, we assume the *inter branch delay* is greater or equal to the number of fetch-decode stages: there will be a zero cycle branch penalty. Now let us consider that the *inter branch delay* is lower than the number of fetch-decode stages. Figure 7 shows that the penalty due to this configuration depends on the *inter branch delay* and the number of fetch-decode stages. In the illustrated example, when the two consecutive branch instructions are taken, there is a one cycle penalty.

2.3 The register branch instructions case

On our benchmark set, we have studied the dynamic ratio of register branch instructions. These probabilities show that barely 5 % of branch instructions are *register branch instructions*. As we are focusing on low cost hardware implementation (e.g. without requirement of extra access ports on register files and without complex control for nullifying instructions on false address computations), we have not considered any specific hardware for these *register branch instructions*.

Figure 8 shows the execution of a *register branch instruction*. If the execution stage is the nth stage of the pipeline, each *register branch instruction* will have *(n-x)* delay slots, where x is the number of stages before the execution stage. Thus if the *register branch instruction* is taken, the branch penalty is x cycles.

3 Performance evaluation

Trace driven simulations have been used to evaluate the performance benefit that may be expected from using a **MIDEE** organization in place of a classical organization.

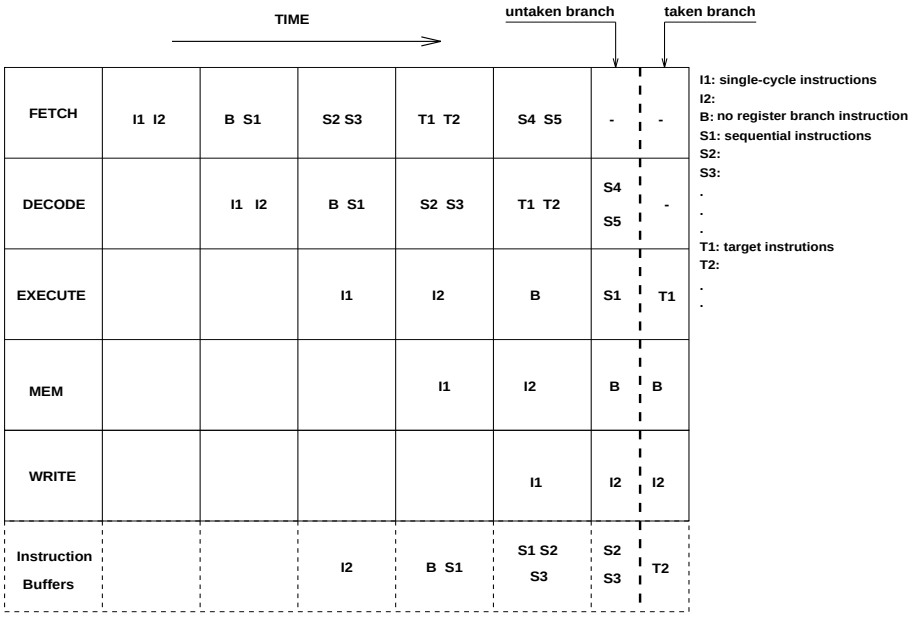

Figure 6: Optimal *no register branch instruction* execution

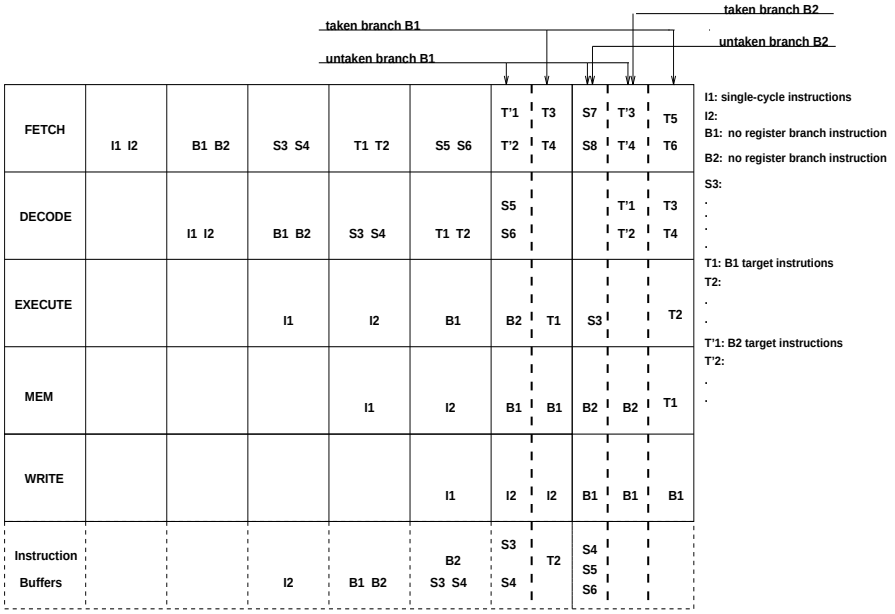

Figure 7: *No register branch instruction* execution with delay (*inter branch delay = 1*)

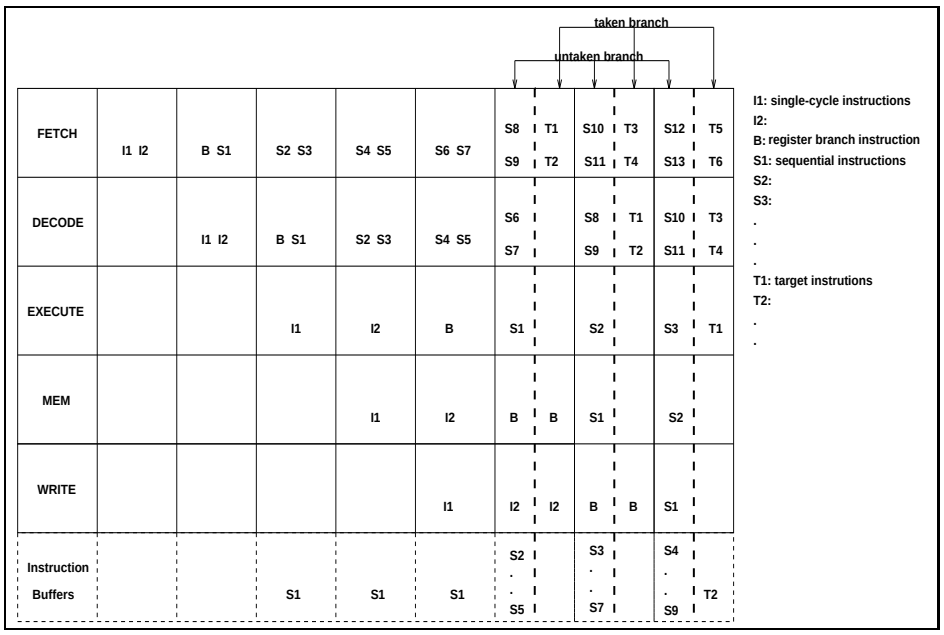

Figure 8: *Register branch instruction* execution with delayed branch strategy

3.1 Simulation model

The *Spa* package developed by Gordon Irlam [6] was used to generate address traces for programs executed on a SUN SparcStation2.

We consider that all instructions are executed in one pipeline cycle.

Classical organization

A classical organization has been implemented with a branch penalty equal to the number of fetch-decode stages minus 1 if the branch is taken, 0 else. It is adaptable for different pipeline depths.

Performance metric

As we only focus on instruction issuing and cache misses, we considered a perfect first-level cache which induces no penalty; data are accessed with 0 cycle. And the first-level instruction cache was considered to be direct-mapped and its size was 4 Kbytes. The cache line size was chosen to be 32 bytes. As a metric, we use the average number of clock Cycles Per Instruction (CPI) [5].

Benchmarks

The first set of benchmarks includes 3 typical C-like applications:

- GCC: the gcc compiler used on a large C input
- LATEX: the latex utility run on a 20 pages article.
- CPTC: a Pascal to C translator run on itself

A monoprocessor version of the NAS benchmark [15] was used as a Fortran benchmark:

- MG : multigrad
- FT : 3-d FFT PDE

And we have simulated two benchmarks from the SPLASH (benchmarks run on a monoprocessor [17]).

- PTHOR: a parallel, distributed time, event driven simulator
- LOCUS: a VLSI standard cell router

3.2 Simulation results

3.2.1 Inter branch delay and number of fetch-decode stages

Table 1 illustrates the *inter branch delays* for several benchmarks. For example, with the LATEX benchmark, one instruction *inter branch delays* represent 0.55 % of the branches.

As an example, if you consider a 4 fetch-decode stages pipeline and the results average, for the **MIDEE** organization around 30 % of branch instructions take penalty: 0.11 % take 3 cycles, 11.09 % take 2 cycles and 18.42 % take 1 cycle. Since around 50 % of branch instructions are taken [18], for the classical organization 50 % of branch instructions take 3 cycles penalty (number of fetch-decode stages minus 1).

Figures 9 and 10 illustrate CPI for each benchmark as a function of number of fetch-decode stages for the **MIDEE** organization and the classical organization. When instruction cache misses are not considered, the **MIDEE** organization performs better than the classical organization; particularly when the number of fetch-decode stages is high. For the LOCUS benchmark with a 5 fetch-decode stages pipeline, the CPI improvement for **MIDEE** is 18.38 %. These results are obtained because the branch penalty is reduced for the **MIDEE** organization.

Inter branch delay	1	2	3	4	5
LATEX	0.55	6.45	12.6	9.53	17.43
GCC	0.25	8.42	34.51	11.25	6.4
CPTC	0.001	17.89	17.69	7.18	14.56
FT	0.002	6.19	11.21	2.74	0.37
MG	0.006	5.11	7.68	0.79	0.65
PTHOR	0.003	16.25	26.12	22.11	13.22
LOCUS	0.002	17.34	19.16	33.55	10.01
AVERAGE	0.11	11.09	18.42	12.45	8.94

Inter branch delay	6	7	8	9	10	11+
LATEX	10.78	7.81	12.22	3.38	3.23	16.02
GCC	20.84	5.08	1.46	3.77	1.24	6.78
CPTC	15.34	5.67	5.73	10.50	2.55	2.89
FT	0.98	7.77	0.52	12.11	2.15	55.96
MG	0.89	5.83	4.88	14.71	0.15	59.31
PTHOR	8.28	5.33	2.21	1.34	0.25	4.89
LOCUS	6.69	4.85	2.50	1.80	0.007	4.10
AVERAGE	9.11	6.04	4.21	6.80	1.36	21.47

Table 1: Dynamic statistics of inter branch delays (%)

3.2.2 Instruction queue size

Now we study the optimal value of instruction queue size. In the **MIDEE** organization, the two instructions queues are not statically affected to the sequential or target instruction sequences, but their usage are dynamically determined; then they will have the same size S. The optimal size S depends on the number of fetch-decode stages and on the main memory or secondary cache latency.

- **Number of fetch-decode stages:** the instruction queue size S must be not less than x words (x is the number of fetch-decode stages). Between the decoding and execution of a *no register branch instruction*, there are instructions alive in instruction queue. x is the minimal instruction queue size.

- **Miss latency:** this miss latency can concern a second-level cache access or the main memory access. The instruction queues must allow to hide most of the instruction cache misses. Most of the delays due to instruction cache misses can be reduced because the pipeline is not stalled as long as the instruction queues are not empty. Figures 11 and 12 show the CPI as function of instruction queue size S for each benchmark. The CPI decreases as the instruction queue size increases. But when the instruction queue size is greater than the miss latency, this decreasing is low. And since

the on-chip space is high cost, the optimal instruction queue size must be approximatively equal to miss latency.

Figures 9, 10, 11, 12, 13 and 14 show that the **MIDEE** organization may dramatically enhance performance when instruction cache misses are considered. In this case, the difference between the **MIDEE** organization and the classical organization is higher. For the PTHOR benchmark with a 4 fetch-decode stages pipeline, a 8 cycles miss penalty and a 10 words instruction size, the CPI improvement for **MIDEE** is 24.77 %. The remaining benchmarks show considerable improvements in CPI for the **MIDEE** organization.

4 Conclusion

The **MIDEE** organization proposed in this paper smoothes the branch and instruction cache miss penalties on deep pipelines. This organization consists of a double fetch and decode, early computation of the target address for branch instructions and the using of a pair of instruction queues. The double fetch-decode concerns a pair of instructions stored at consecutive addresses: no complex dual-ported instruction cache is required. These instructions are then decoded simultaneously, but no execution hardware is duplicated; only useful instructions are effectively executed. Thus branch penalty is reduced. A pair of instruction queues are used between the fetch-decode stages and execution stages. As the **MIDEE** organization induces advanced decoding, instruction cache misses occur in advance, but the instructions already decoded may be executed while servicing the instruction miss. This allows to hide most of the instruction cache miss penalties.

Simulation results are encouraging. They indicate that the **MIDEE** organization compares favorably with the classical organization and show that it provides a quite significant performance improvement at a reasonable hardware cost (no duplications of the execution stages, no hardware for nullifying instructions, etc). Applications of the **MIDEE** organization may be envisaged in both low-end and high-end microprocessor designs. The simplicity of the mechanisms and particularly the absence of the need for nullifying instruction executions allows to envisage boosting low-end microprocessor performance at a reasonable transistor count, while on high-end microprocessor, they would not lengthen the critical paths for enabling very high clock frequency.

Since the pipeline depth and particularly cache pipeline access depth, continue to increase, using the **MIDEE** organization is becoming very attractive: branch penalty may be reduced and pipeline stalls due to instruction cache misses may be decreased. Notice that the **MIDEE** organization may be adapted to a superscalar pipeline. In this case, the number of instruction addresses simultaneously fetched and decoded should be twice the number of instructions issued in parallel.

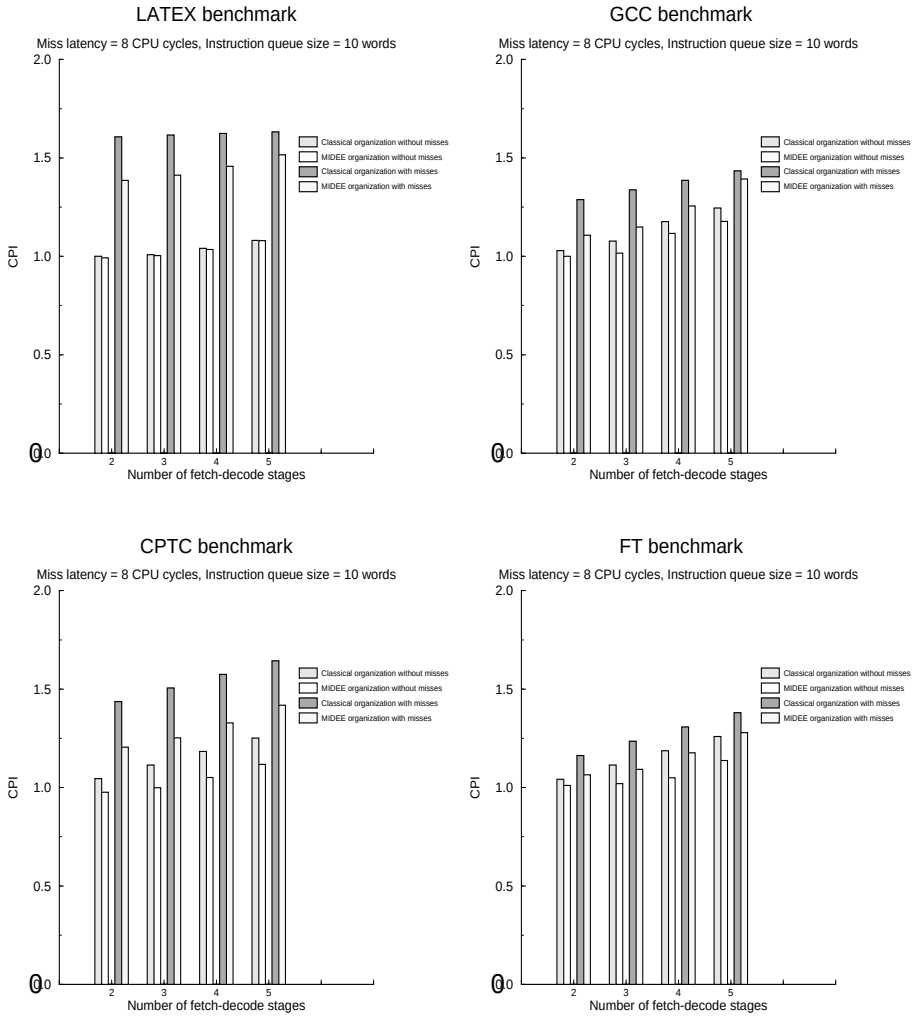

Figure 9: CPI as function of number of fetch-decode stages

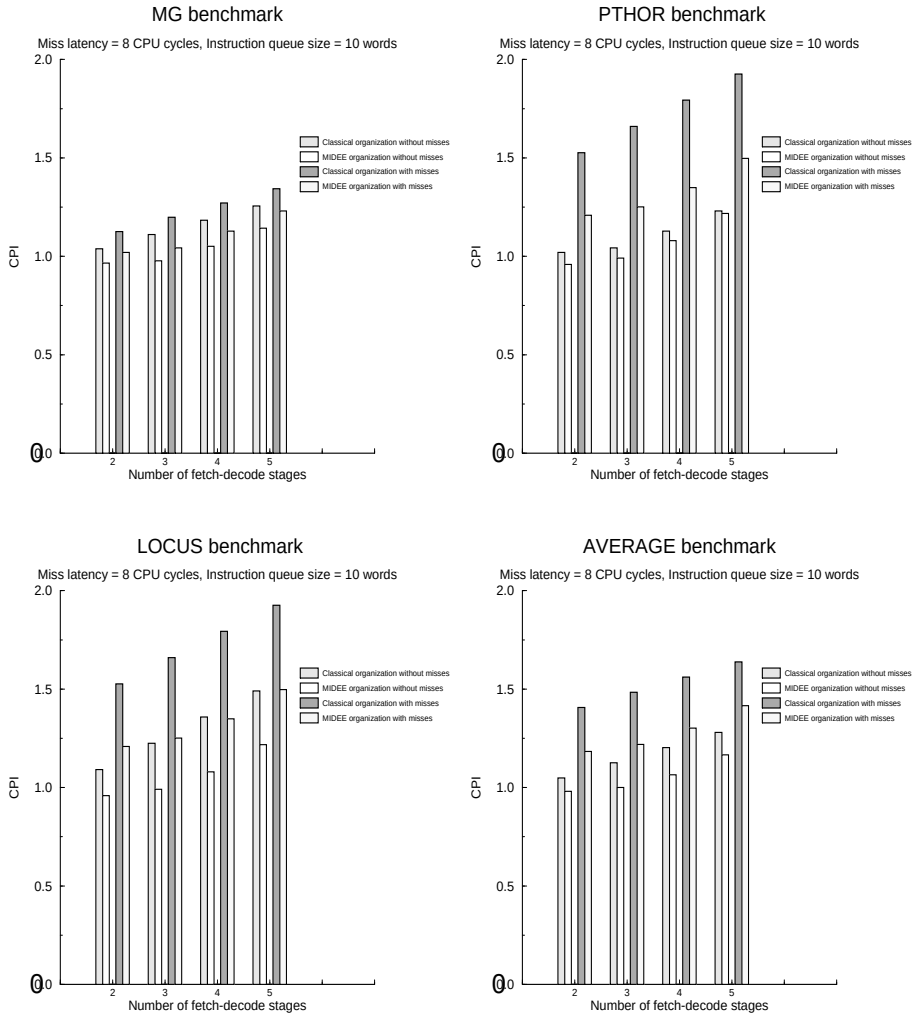

Figure 10: CPI as function of number of fetch-decode stages

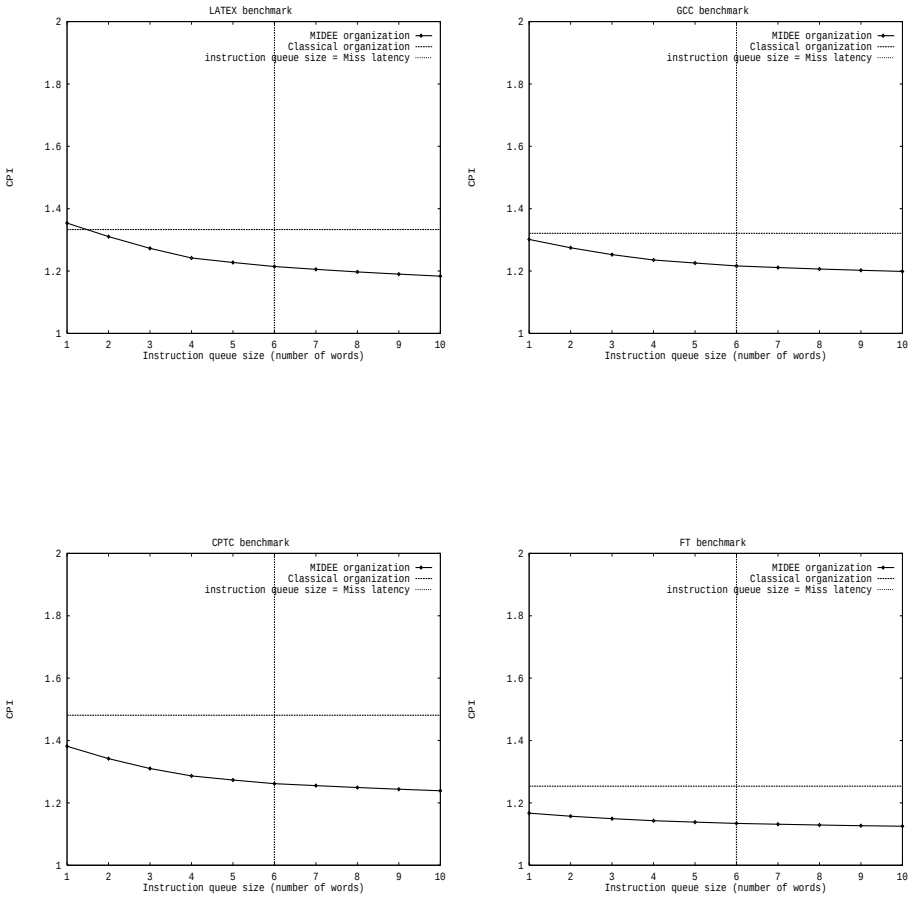

Figure 11: CPI as function of instruction queue size (miss latency = 6 CPU cycles, number of fetch-decode stages = 4)

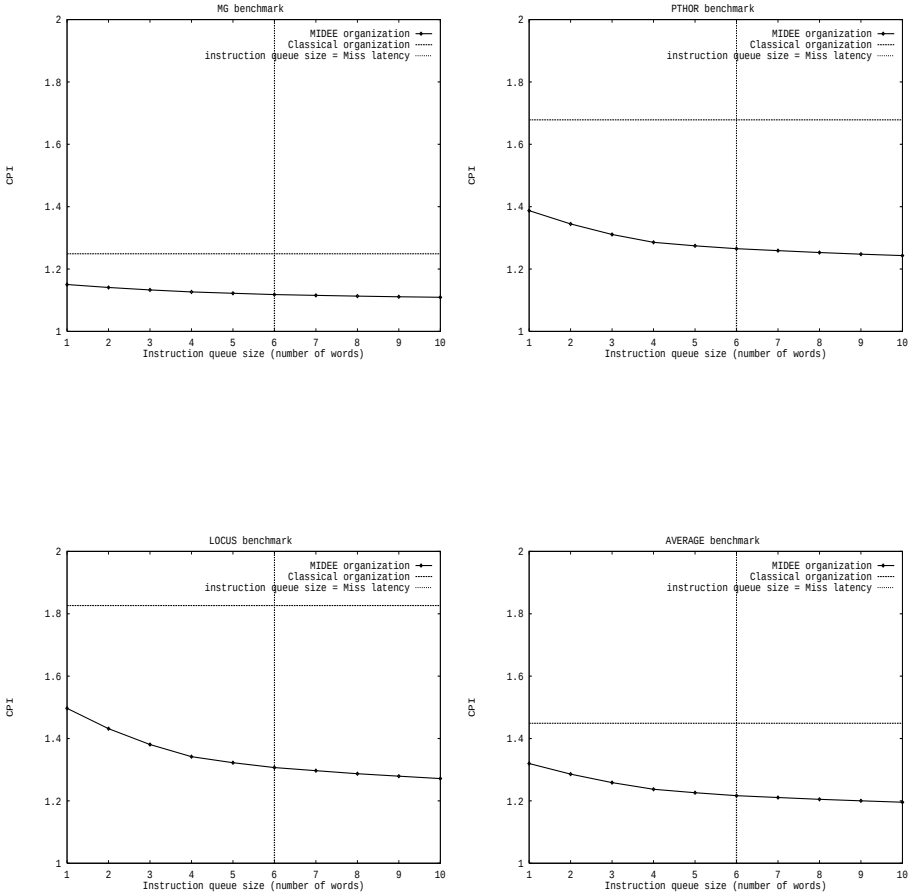

Figure 12: CPI as function of instruction queue size (miss latency = 6 CPU cycles, number of fetch-decode stages = 4)

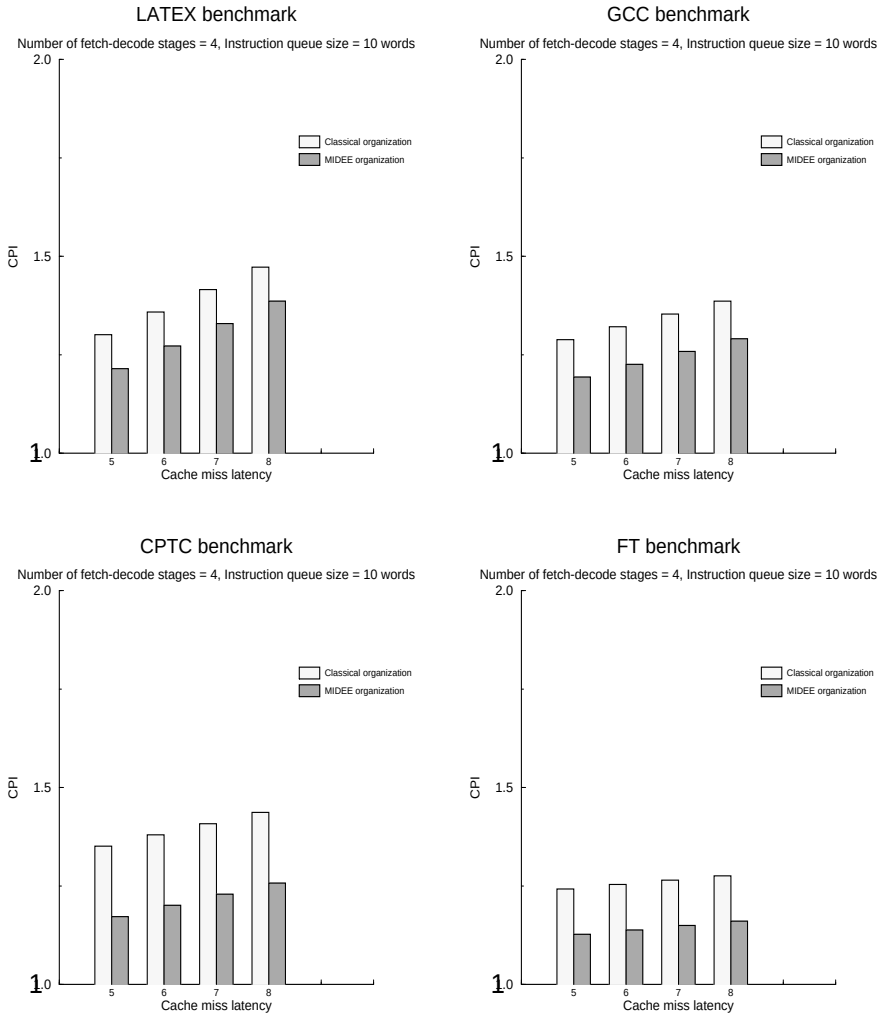

Figure 13: CPI as function of cache miss latency

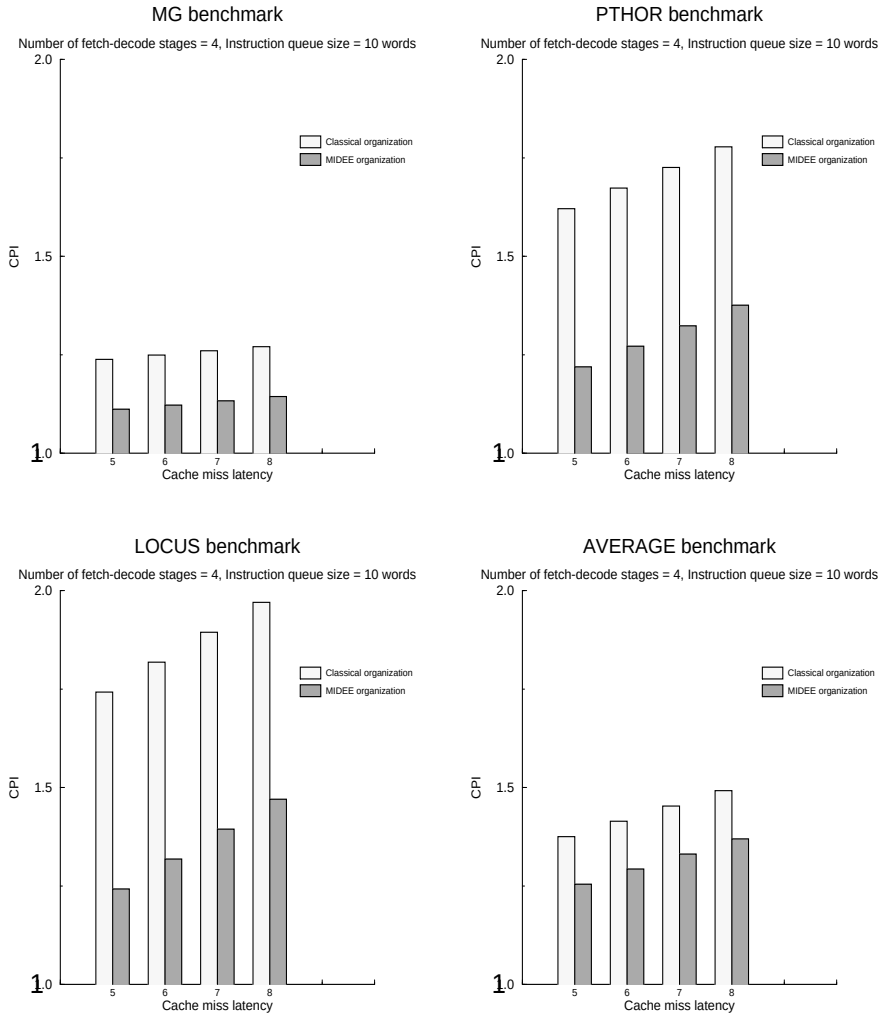

Figure 14: CPI as function of cache miss latency

References

[1] J.A. DeRosa, H.M. Levy, " An Evaluation of Branch Architectures", Proceedings of the 14^{th} International Symposium on Computer Architecture, June 1987

[2] D.R. Ditzel, H.R. McLellan, A.D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor", Proceedings of the 14^{th} International Symposium on Computer Architecture, June 1987

[3] D.R. Ditzel, H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of the 14^{th} International Symposium on Computer Architecture, June 1987

[4] T.R Gross, J. Hennessy, "Optimizing Delayed Branches", Proceedings of the 15^{th} Annual Workshop on Microprogramming, Oct. 1982

[5] J.L. Hennessy, D.A. Patterson *Computer Architecture a Quantitative Approach* Morgan Kaufmann Publishers, Inc. 1990

[6] G. Irlam, *SPA package*, 1991

[7] A. Gonzales, J.M. Llaberia, J. Cortadella, "A Mechanism for Reducing the Cost of Branches in RISC Architectures", Microprocessing and Microprogramming 24, 1988

[8] A.M. Gonzalez, J.M. Llaberia, "Instruction Fetch Unit for Parallel Execution of Branch Instructions", Proceedings of the 3^{rd} Supercomputing, June 1989

[9] A.M. Gonzalez, J.M. Llaberia, "Reducing Branch Delay to Zero In Pipelined Processors", IEEE Transactions on Computers, March 1993

[10] "IBM RISC system/6000 technology", IBM Corporation, 1990

[11] G. Kane, J. Heinrich *MIPS RISC Architecture* Prentice-Hall, 1992

[12] M.J. Knieser, C.A. Papachristou, "Y-Pipe: A Conditionnal Branching Scheme Without Pipeline Delays", Proceedings of the 25^{th} Annual International Symposium of Microarchitecture, Dec. 1992

[13] J. Lee, A.J. Smith, " Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer,January 1984

[14] S. McFarling, J. Hennessy, "Reducing the Cost of Branches", Proceedings of the 13^{th} International Symposium on Computer Architecture, 1986

[15] D.H. Bailey, E. Barszcz, L. Dagum, H.D. Simon "NAS Parallel Benchmarks Results", Proceedings of Supercomputing '92

[16] R.R Oehler, R.D Groves, "IBM RISC System/6000 Processor Architecture", IBM J. Res. Develop., 1990

[17] J.P. Singh, W. Weber, A. Gupta "SPLASH : Stanford Parallel Applications for Shared-Memory", Technical Report CSL-TR-91-469, Stanford University, 1991

[18] J.E Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8^{th} International osium on Computer Architecture, May 1981

[19] *SPARC Architecture Manual (Version 7)*, Sun Microsystems, Inc., 1990

[20] K.D. Wilken, D.W. Goodwin, "Two Zero-Cost Branches Using Instruction Registers", Proceedings of the 25^{th} Annual International Symposium of Microarchitecture, Dec. 1992

[21] T-Y. Yeh, Y.N. Patt, "Alternative Implementation of Two-Level Adaptative Branch Prediction", Proceedings of the 19^{th} International Symposium on Computer Architecture, May 1992

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399