Branch Prediction and Simultaneous Multithreading

Sébastien Hily, André Seznec

Branch Prediction and Simultaneous Multithreading

Sébastien Hily & André Seznec

N˚ 2843

mars 1996

THÈME 1

ISSN 0249-6399

Branch Prediction and Simultaneous Multithreading

Sébastien Hily & André Seznec*

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n° 2843 — mars 1996 — 24 pages

Abstract: In this paper, we examined the behavior of three of the best performing branch prediction strategies while executing several threads of instructions simultaneously. We studied the impact of the addition of one Return Address Stack per hardware context. We showed that a 12-deep stack per thread is sufficient to enhance greatly the accuracy of branch prediction while adding a minimal implementation cost. We explored the behavior of the branch predictors when independant applications are running simultaneously and when the workload is constituted by a parallel program. Our simulations showed that in multiprogramming environment, if the sizes of the tables (PHT/BTB) are proportionnal to the number of active threads, there are very few interactions, be they destructive or constructive. With parallel workloads, we could have expected a beneficial sharing effect. In fact, it is very dependant of the branch predictors and in the best case, the gains stay very limited. Finally we showed that, for the three predictors, whether in multiprogramming or in parallel processing, if the sizes of the tables are kept small, there is a slight increase of the mispredictions, which is mostly due to an increase of the conflicts in the BTB.

Key-words: multithreading, branch prediction, superscalar

(Résumé : tsvp)

Les travaux de Sébastien Hily sont en partie financés par la région Bretagne

*hily@irisa.fr, seznec@irisa.fr

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

Prédiction de branchement et multiflot simultané

Résumé : Dans cette étude, nous examinons le comportement de trois stratégies de prédiction de branchement, parmi les plus performantes, lorsque plusieurs flots d'instructions sont exécutés simultanément.
Nous avons étudié l'intérêt de disposer d'une pile d'adresses de retour par contexte. Nous avons ainsi pu montrer qu'une pile de 12 entrées par flot est suffisante pour améliorer de façon significative la validité des prédictions de branchement tout en n'engendrant qu'un faible surcoût matériel.
Nous avons exploré le comportement des mécanismes de prédiction quand des applications indépendantes s'exécutent simultanément et quand les applications sont issues d'un même programme parallèle. Nos simulations ont montré que dans un environnement multiprogrammé, si les tailles des tables (PHT/BTB) sont proportionnelles au nombre de flots actifs, il y a très peu d'interactions, aussi bien constructives que destructives.
Pour un programme parallèle, nous pouvions attendre un effet de partage bénéfique. En fait, cela dépend du type de prédiction et, dans le meilleur des cas, les gains restent très limités.
Enfin, nous avons montré que pour les trois types de prédiction, que ce soit pour des applications indépendantes ou issues d'un programme parallèle, si les tables sont maintenues à des tailles faibles, il y a une légère augmentation du taux de mauvaises prédictions. Cette augmentation correspond principalement à une hausse du nombre de conflits dans la BTB.

Mots-clé : multiflot, prédiction de branchement, superscalaire

1 Introduction

The number of instructions that superscalar microprocessors can issue every cycle increases steadily. Actual processors are, or will soon be able to issue six instructions simultaneously. Nevertheless, a typical 6-issue superscalar architecture, on current applications, hardly sustains 1.5 instructions per cycle [JSL95]. Such a low hardware usage is mostly due to three constraints :

- dependencies between the instructions;
- breaks in the control flow, due to branches;
- wait states generated by cache misses.

To improve pipeline utilization, simultaneous multithreading (*SM*) [TEL95] is a promising technique. Several instructions issued from different threads are executed concurrently. These threads may be independant processes or processes issued by a single application. By this mean, dependencies tend to disappear and the long latency operations such as cache misses or divide operation, for a thread can be overlapped by useful execution of instructions from the other available threads [Aga92][Far91]. Moreover, the processor utilization is enhanced by the capacity of *SM* to schedule resources dynamically among the available threads. In [TEL95], Tullsen & al. showed that a simultaneous multithreaded architecture can achieve four times the instruction throughput of a singlethreaded wide superscalar with the same issue width.

To tackle with the branch problem on singlethreaded architectures, extensive research studies have been conducted on software and hardware mechanisms. Smith [Smi81] studied several hardware schemes for predicting branch directions, Lee and Smith [LS84] illustrated the interest of combining a good history prediction with BTB, Yeh and Patt [YP92a] imagined the two-level adaptive branch prediction while McFarling proposed combining branch predictors [McF93]. Branch prediction strategies for superscalar architectures now achieve more than 90% accuracy.

However, the effectiveness or even simply the usefulness of these prediction mechanisms in a multithreaded environment is not obvious. The purpose of this paper is to study the behavior of different kinds of branch prediction strategies when codes of several threads are executed simultaneously.

The interest of providing one private Return Address Stack per active thread will be examined. We will show that 12 entries stacks lead to an important decrease of the misprediction ratios while adding only a small implementation cost.

We will also explore the impact of the simultaneous use of prediction tables by several threads on the branch prediction accuracy. We will particularly try to characterize wether the threads take advantage of bigger tables or if the number of misprediction increases due to pollution. This will be done for multiprogramming processing, where completely independant processes execute, but also for parallel applications, in which execution of identical processes could induce sharing effects.

2 Branch Management

Most applications exhibit a ratio of 15% to 30% of branches; let's say one instruction out of five. The way these instructions will be handled is then a critical issue.

As illustrated by figure 1, in a pipelined microprocessor, the precise address of a branch is known only several cycles after the issue of the branch instruction. Waiting for the branch resolution for issuing new instructions would lead to poor performance.

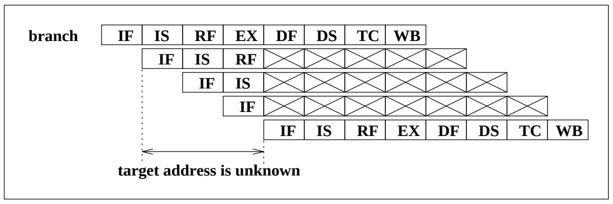

Figure 1: branch behavior on the MIPS R4000

Thus, many studies have been conducted in order to predict the future address for the issue to continue. When the prediction is correct, no cycle is lost. On a misprediction, mispredicted instructions feeded in the pipeline have to be discarded, resulting in pipeline bubbles, and so a loss of performance. Moreover, if the issue of wrong instructions has resulted in cache misses, the penalty may be more important. Therefore, processor performance depends highly on the branch prediction accuracy.

Two kinds of predictions [LS84] can be distinguished. Static branch prediction uses opcode, profiling statistics or branch direction and is predetermined before execution. Dynamic prediction uses run-time history and predictions are not known until run-time.

Among branch prediction policies, one can find, by order of efficiency (as well as of complexity) [LS84] [YP92a]: not taken, taken, BTFN (backward branches taken, forward branches not taken), compiler directed, 1-bit history, 2-bit history and two-level algorithm. The first four policies, which are static schemes, have a relatively low efficiency (less than 75% for the compiler directed algorithm).The next three are dynamic.

Dynamic branch prediction [LS84] can be split in two different problems. The predicted information have to be the direction of the branch, and, for taken branches, the target instruction address.

The common way to predict the target address is to use a BTB (Branch Target Buffer), i.e a small cache memory which is read in parallel with the instruction cache. Each entry consists of the address of a branch, its target address and some state information. The target address is the one that was used at the previous occurence of the branch, so it is immediately available. A BTB will produce accurate information in all cases where the target address of a branch is unlikely to change frequently. If the target of a branch changes, the pipeline will have to be flushed when the true target address will be calculated.

Prediction of the direction is usually based on branch history. It consists of using the previous sequence of taken/not taken information for each branch to predict whether or not the branch will be taken next time it occurs. There is a trade-off between the size of history vector to retain, i.e the complexity, and the performance. J. E. Smith proposed utilizing a two-bit saturating up/down counter scheme to collect history [Smi81]. The state diagram of a different two-bit scheme is illustrated in figure 2. ST means strongly taken, WNT stands for weakly not taken. The label on the arrow is the result of the branch. This strategy

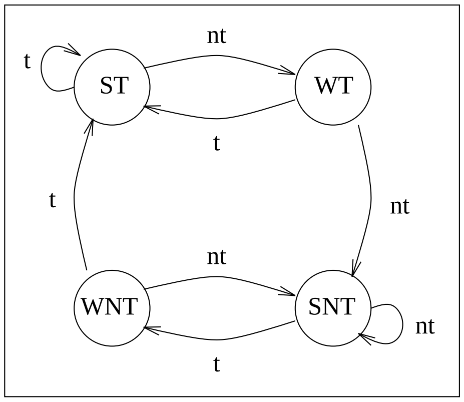

Figure 2: two-bit prediction state diagram

is slightly better than the saturating counters, mostly because if two wrong guesses are required to change the prediction, two are also required to return to the previous prediction. This dynamic scheme is more efficient than any static one, with 80 to 90 per cent of success [LS84].

Yeh and Patt have shown that substantially more accuracy can be achieved by utilizing more branch history [YP92a]. They proposed to use two levels of branch history information (two-level adaptive branch predictor) to make predictions (figure 3). The first level (history table) keeps the history of the last k branches encountered, the second (PHT, for Pattern History Table) is the branch behavior for the last s occurences of the specific pattern of these k branches. They studied several of their schemes, varying the size of the first level (global branch history register, per-set table, per-address table) or those of the second ones (global, per-set or per-address pattern history table). With these kinds of mechanisms, one can expect more than 95% of right guesses.

The less complex implementation, called GAg, uses only a global history vector as first level. The GAg model is interesting as it should have less delays and be easier to pipeline than local predictors (per-set or per-address). However, global history information is less efficient at identifying the current branch than simply using the branch address. History vector works well for workloads dominated by loops, like in scientific codes. This is not the case of integer workloads, where one branch depends generally on a recent one. We can

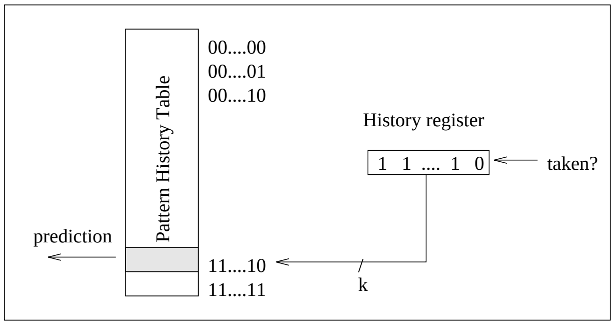

Figure 3: two-level adaptive branch prediction

say that they are correlated. A more efficient prediction can be made by hashing both the branch address and the global history, as proposed by Pan, So and Rahmeh [PSR92]. This kind of *bit selection* identifies the branches correctly. Another method is to do an exclusive OR of the history with the branch address; more bits from each of the two bit vectors are in use, that's why it is slightly better than *selection*.

superscalar architecture of degree **6**
20% of branches in the code
5% of bad predictions
8 cycles of penalty on a misprediction

there is a bad branch for every **100** instructions
it results in the issue of **48** additionnal instructions
say **148** issued instructions for **100** used
the maximum IPC is then

$$100 \times \frac{6}{148} = \mathbf{4.05}$$

If the architecture can execute **6** different threads
there is a bad branch for every **100** instructions
it results in the issue of **8** additionnal instructions
say **108** issued instructions for **100** used
the maximum IPC is then

$$100 \times \frac{6}{108} = \mathbf{5.56}$$

A still more accurate prediction, especially for small predictor sizes, can be obtained by combining branch predictors [McF93]. An algorithm combining 2-bit and XOR-selection predictors has largely higher prediction success (between 92 and 96 per cent) than two-level local prediction for predictor size lower than 512 bytes and equals it for larger predictors (near 97 per cent).

However, some remarks can be made about these figures.

First, a validity rate of 95% for the predictions is still not satisfying. Let us take the example of a superscalar architecture of degree 6, in which nothing can disturb the instruction issue. As illustrated in the insert, with a bad prediction penalty of 8 cycles, one cannot expect the number of executed intructions per cycle (IPC) to exceed 4.05 (although six is the ideal).

Next, the size of the tables required by the dynamic prediction can become very important. Thus, for the two-level mechanism of the Pentium-Pro architecture to achieve 90% to 95% of success on Specint-like programs, the control logic and the tables implemented require a number of transistors equivalent to those of a 8kbytes cache.

Finally, these numbers represent an optimistic estimation on the execution of a single thread. They especially do not take into account the context switches, which may invalidate tables, nor the badly predicted addresses due to the latency before the update of the tables.

3 Simultaneous Multithreading & Branch Prediction

If extensive research has been conducted on branch prediction for singlethreaded architecture, to our knowledge none has been done for multithreaded one. This study evaluates the new constraints of multithreaded execution on branch prediction and the potential performance improvements that we could expect.

Access to prediction tables in a singlethreaded environment, as for a classical cache memory, can result in different types of misses :

- initialization miss (or cold-start), the first time a branch is encountered;
- intrinsic miss, when branches are driving themselves out mutually due to the limited size of the tables.

Multithreading introduces a new one, the extrinsic default generated when branches of different threads are driving themselves out mutually [WG89]. The frequency of this kind of misses should vary a lot, depending on the simultaneously executed applications. Thus, it must be maximal when all processes are independant, like in multiprogramming processing. On the other hand, if the control of the tables allows it (for example by keeping application identifiers), it must diminish as soon as there is sharing (prefetch effect).

Having a prediction table and branch target buffer per thread is a solution to suppress the extrinsic misses. However, it should not be very cost effective, as tables need a lot of space. Moreover, the benefit of sharing totally disappears.

The question of sharing is an interesting one. When a branch target is computed, it is placed, if not already, in the Branch Target Buffer. Threads issued from a same application generally share the same instruction code. Their simultaneous execution should therefore induce a prefetch-like effect, one thread placing in the BTB addresses used later by other threads. However, for caches, Gupta [WG89] has shown that the number of intrinsic misses generated by different processes generally takes precedence over the prefetch effect.

As presented in the previous section, the currently obtained rates of success for predictions are still not satisfying. An advantage of multithreading is that it should put up with a smaller prediction accuracy as an important part of the pipeline is filled with usefull instructions from the other threads. However, prediction has to be sufficiently accurate so that the processor does not offer too much degraded performance when only a single thread is available for execution.

All these points are of great interest and should lead us to further studies. A preliminary step has been to evaluate the behavior of some of the prediction strategies which perform best for a single thread, in multithreaded environment. This work was done through simulations and will be the subject of the following sections.

4 Methodology for Performance Evaluation

We used the Spy program to conduct our simulations. Spy is part of SPA package, a set of tools written by Gordon Irlam to analyze the performance of SPARC binaries. We modified Spy to be able to support the simultaneous tracing of several programs. In order to handle parallelism (process creation, synchronization, locks, ...), we have also integrated supports for PARMACS, a set of portable PARallel MACroS defined at the Argone National Laboratory. In addition, we developed a fully configurable simulator program which read instruction streams generated by Spy and integrate feedback control necessary for handling parallelism. It can model superscalar execution pipelines, dynamic scheduling, memory hierarchy and branch prediction as well as multithreading and context switching.

4.1 Simulated Architecture

The simulated architecture is a multithreaded processor based on the SPARC7 instructions set [LOG90] and implementing a future-generation superscalar architecture. We assumed our processor was able to execute simultaneously 8 instructions in an out-of-order fashion.

Each thread is supported by a context, which includes a program counter, status registers and 32 registers (but 40 physical registers). In each cycle, up to 8 instructions are fetched from the available threads in a round-robin fashion. A single thread can eventually furnish all the issued instructions in that cycle. It is close to the Full Simultaneous Issue model in [TEL95], which is their most complex model to implement, but which offers the best insight into multithreaded potential.

Available threads are those which are resident on the processor in a running state. A thread can be resident but in a suspended state due to a synchronization (lock, barrier, ...), and only for parallel applications (we did not simulate here the impact of context switches).

Our study was focused on branch prediction, so we compared three different strategies : *2bit*, *gselect* and *gshare*. We were inspired by the GAg predictor in [YP92b]. It has the advantage of being simple, and should allow the prediction in one cycle. It should reduce the risk of bad prediction due to the use of non updated tables, which is important as we do not deal with the number of cycles normally necessary for the target address calculation. A schematic representation of the prediction mechanisms is given in figure 4. The *2bit* algorithm (illustrated figure 2) is a slightly modified version of the 2-bit saturating counter scheme proposed by Smith [LS84]. In *gselect*, the PHT is indexed with the concatenation of the lowest order bits of the address with the history register [PSR92] while in *gshare*, the index is the XOR of the history register with the lower order bits of the address [McF93]. *gselect* and *gshare* are names introduced by McFarling in [McF93].

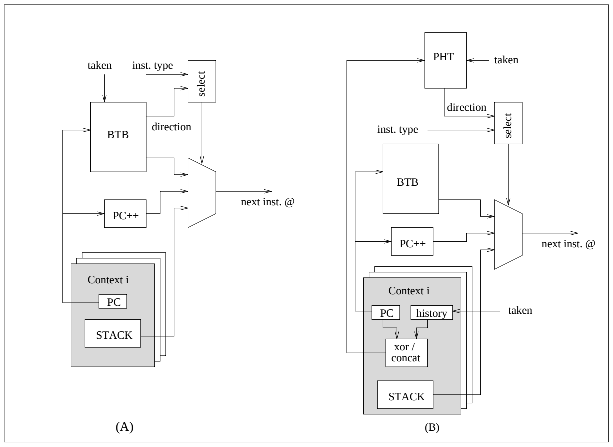

Figure 4: representation of the *2bit* (A) and *gshare/gselect* (B) prediction

When Return Address Stacks (RAS) are present, we used 12 entries stacks as in the DEC 21164 [Cor94], which should perform well and is a good compromise with complexity [YP92b]. A stack is implemented as a circular register file, which means that when the end of the file is reached, stack can continue at the beginning, with the older data discarded by the new ones. Thus, the last 12 return addresses are always available.

We supposed that at the prediction stage of the pipeline, we know which kind of branch we have (conditional, cond. always taken or never taken, jump, call or ret). It is possible if the instructions are predecoded during the fetch stage, as it is frequently done now (MIPS R8000, ...), or if the information is kept in the BTB after the first occurence of the branch (which induces a fault for the first occurence).

We fitted the size of the BTB according to the number of simultaneous threads, with a base of 512 entries per thread. The BTB is 4-way set associative, with a LRU replacement policy. The PHT, when present, has a size of 4096 entries multiplied by the number of threads and is direct-mapped.

4.2 Benchmarks

We used two different kinds of benchmark series depending on our goal. To simulate problems of sharing in multithreaded environment, we used applications of SPLASH2 [WOT+95] [SWG92]. SPLASH2 is a set of parallel applications for shared memory architectures written in Stanford University and using PARMACS. It covers a large field of currently encountered scientific applications.

For multiprogramming purpose, we used applications of SPLASH2 and of Spec92 benchmarks [Dix92]. This last set of applications was built by SPEC (Standard Performance Evaluation Corporation) as a standard performance evaluation platform for microprocessors. It includes a mix of integer and floating point programs. Table 1 gives a list of the benchmarks used in the simulations.

SPEC codes	type	description
compress	int	compression (Lempel-Ziv) of a 1MB file
xlisp	int	lisp interpreter on the 9 queens problem
espresso	int	minimizes boolean functions
ear	fp	simulates propagation of sounds in an inner ear model
tomcatv	fp	vectorized mesh generation program
wave5	fp	two dimensional simulation of plasma phenomena
SPLASH codes	type	description
fft	kernel (fp)	complex 1-D version of the radix-$\sqrt{n}$ six step FFT
lu	kernel (fp)	LU factorization
radix	kernel (int)	integer radix sort program
barnes	appli. (fp)	3-D simulation of bodies interactions using Barnes-Hut method
cholesky	appli (int)	blocked sparse Cholesky matrice factorization
volrend	appli. (fp)	3-D volume rendering using a ray casting technique
raytrace	appli. (fp)	3-D scene rendering using a ray tracing technique
watern	appli. (fp)	evaluates forces and potentials occuring over time in a system of water molecules

Table 1: Benchmarks of Spec92 and SPLASH2 used in the simulation

The programs were compiled on Sparcstations (2, 10 or 20) using gcc or f77, with the standard optimization -o. We used, as often as possible, the standard inputs recommended for the Spec92 and the SPLASH2 series. It was especially not the case, in the parallel study, for *barnes* for which we simulated only 8192 bodies (instead of standard 16384 used in the multiprogrammed part) and for *volrend* and *raytrace* for which we used *head-scaleddown2* and *teapot* respectively as inputs.

5 Multiprogrammed Workload

We first examine the behavior of the three branch prediction strategies on a workload corresponding to a multiprogramming processing. We used several applications issued from the Spec92 (compress, ear, espresso, tomcatv, wave5 and xlisp) and SPLASH2 (barnes, cholesky, fft and lu) packages. The SPLASH2 applications were running for a one processor machine, i.e. not parallel. For each application, the first 50 million instructions were ignored and we simulated the following 10 million instructions.

Figure 5 shows the static branches found in the traces. For a given program, this number is fixed and can be counted by looking at the program. The total number is given between brackets after the name of the benchmark. *catnt* represents the conditionnal branches always or never taken.

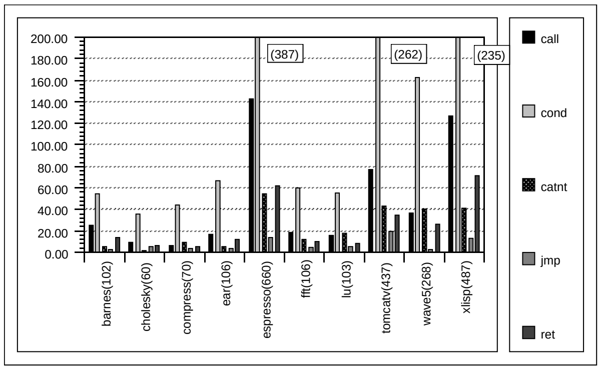

Figure 5: distribution of static branch instructions

We simulated a classical singlethreaded (but superscalar) architecture (figure 6), a two-degree multithreaded architecture (figure 7) and a four-degree multithreaded architecture (figure 8). For each application, we expressed the percentage of bad branch predictions corresponding to the different prediction strategies. The sizes of the prediction tables are kept proportionnal to the number of threads, with a base per thread of 512 entries for the BTB and 4096 entries for the PHT. Each entry in the PHT corresponds to a 2 bits counter (figure 2). In the *2bit* branch prediction algorithm, the PHT is part of the BTB, and thus has the same number of entries and is 4-way set associative.

Figure 6 shows, for each application executing alone, the ratio of bad branch predictions corresponding to the three prediction strategies, with and without a Return Address Stack (RAS). In the keys, *xxx-s* refers to a simple branch prediction scheme without stack and *xxx-12* to the use of a 12-deep RAS.

As expected, for a singlethreaded execution, the *gshare* strategy is the best performing, with *gselect* following closely. In most of the cases, the best *gshare* achieves less than 5% of bad predictions. The simplest *2bit* scheme is outperformed, with more varying accuracy

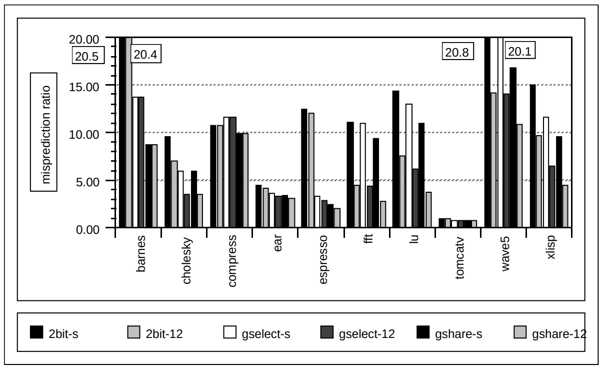

Figure 6: 1 threads, BTB of 512 entries, PHT of 4096 entries

than the two previous strategies, and for several benchmarks, has misprediction rates greater than 10%.

Why a RAS per context The RAS has a variable impact on misprediction ratios, depending on the running application. There is no gain for *barnes* or *compress* but 30% to 50% of misprediction reductions are obtained for *xlisp*. For *fft* or *lu*, the results are even more impressive, with misprediction ratios more than halved.

The usefulness of the stack depends greatly on the dynamic interlacing of the *call* and *ret* instructions. A private stack per context is needed since a stack is a typical resource that cannot be shared between several threads. Indeed, in a stack, data manipulations are reduced to push and pop operations on the top. The coherence of the stack is only given by the order of *call* and *ret* instructions in the program code. By mixing such instructions from several threads, one can no longer maintain the consistency of the data order in the stack and the instructions order in each individual thread. Other simulations have shown that the gain in performance when shifting to a 32-deep stack is very small. 12-deep Return Address Stack works already quite well and is cheap to implement. Thus, we will later assume a 12-deep RAS per context.

Multiprogramming processing We now examine results obtained when several threads are simultaneously executed. In the following figures, the name of a workload is the concatenation of the first two letters of the involved applications (for example, *bach* corresponds to the execution of *barnes* with *cholesky*).

Figures 7 and 8 show that the simultaneous execution of 2 and even 4 threads exhibit the same behavior as with one thread. There is no global increase nor decrease of the misprediction ratios for the three prediction schemes and the *gshare* scheme keeps performing the best.

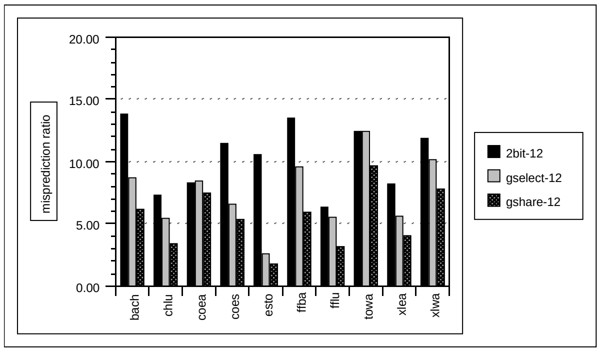

Figure 7: 2 threads, BTB of 1024 entries, PHT of 8192 entries

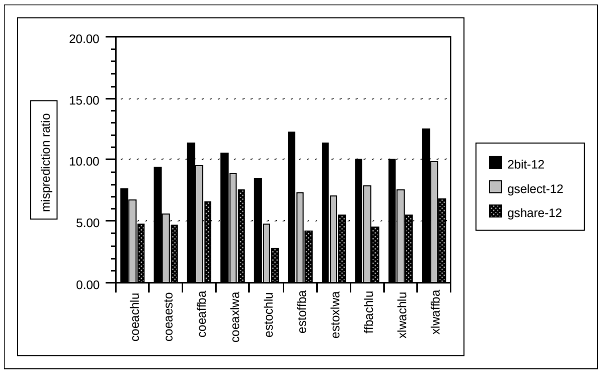

Figure 8: 4 threads, BTB of 2048 entries, PHT of 16k entries

One has to be careful not to compare directly the results obtained for one thread and for two or four threads. Indeed, the mispredictions of the different applications do not have the same weight in terms of number of branch instructions. In order to evaluate the impact of simultaneous execution, either constructive or destructive, we have computed the misprediction ratios which would have been obtained when the benchmarks of the different workloads are executed sequentially. In figures 9 and 10, $A{+}B/AB$ represents respectively the sequential/simultaneous execution. There appear to be few differences between the two types of execution, which means that there are few interactions of the different threads in the tables (PHT, BTB) of the branch prediction unit.

The *2bit* strategy has a remarkable steady behavior. From the figure 10, one can observe that, although very minor, *gselect* has the more fluctuating results, sometimes positives, other times negatives. Finally, simultaneous execution is never negative for *gshare*, and can sometimes be positive.

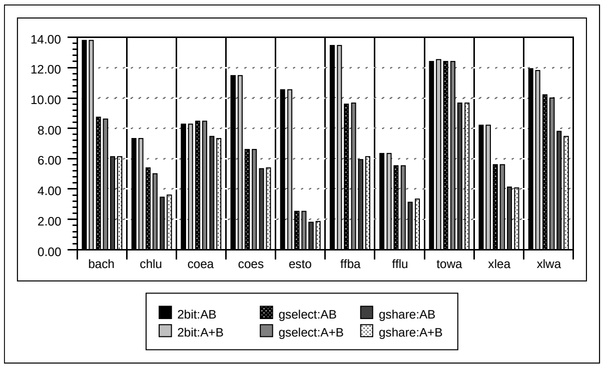

Figure 9: comparison between A+B and AB predictions for 2 threads

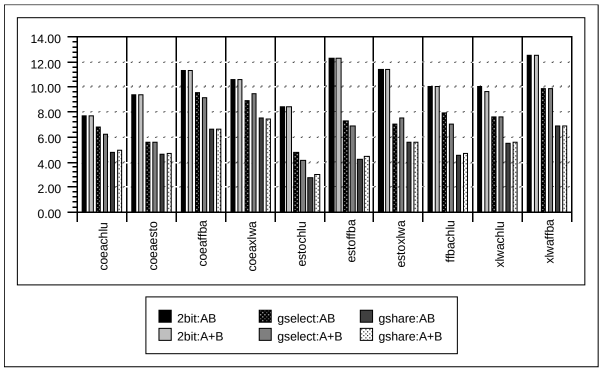

Figure 10: comparison between A+B and AB predictions for 4 threads

RAS and Multithreading Assuming the usefulness of the Return Address Stack from results obtained with singlethread execution, the previous simulations used one 12-deep RAS per context. Figure 11 shows, for a four-degree multithreaded architecture, the decrease of the misprediction ratios obtained over prediction without stack, with the addition of these 12-deep stacks. Except for one workload, the decreases are always greater than 10%. More the global prediction becomes accurate, more the advantage taken from the stacks grows. Thus, the gain for the best performing *gshare* scheme is as high as 30% to 50%. All of this confirms the necessity of implementing one stack per context.

To evaluate the impact of the size of the stack, we have compared the misprediction ratios obtained with 12-deep stacks versus 32-deep one (figure 12).

For all the workloads, there is no gain increasing the RAS size to 32. However, it is essential to understand if the 12-deep stacks are already sufficient or if 32 entries are still

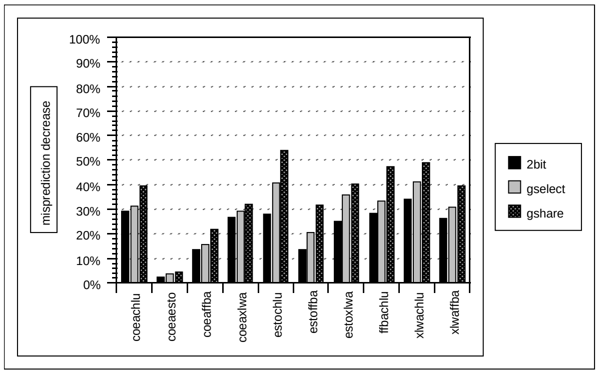

Figure 11: 4 threads, misprediction reductions with RAS

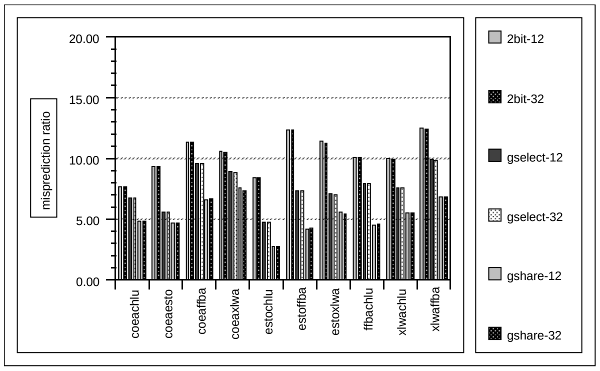

Figure 12: misprediction ratios with 12 and 32 entries RAS

bad. Figure 13 shows the distribution of the mispredictions for the *gshare* prediction strategy (but the results are independant of the strategy).

Capital and small letters stand respectively for the true program behavior and the predicted branch direction. *T/NT* means the branch is taken/not taken, *hit/miss* refers to the BTB and *bad @* means a misfetch occured. *bad return @* signifies that the RAS didn't give the good address.

For the workloads *coeaesto*, the 12-deep RAS has achieved already 100% of good predictions. Workloads as *coeaxlwa*, *estoxlwa* or *xlwaffba* take a small advantage of the increase of the stack size. However, as illustrated by figure 12 it results only in an insignificant gain in the overall misprediction ratio. On the other hand, workloads like *coeachlu*, *estochlu* or *ffbachlu* keep a high rate of bad return address mispredictions without taking any advantage of a shift to 32 entries.

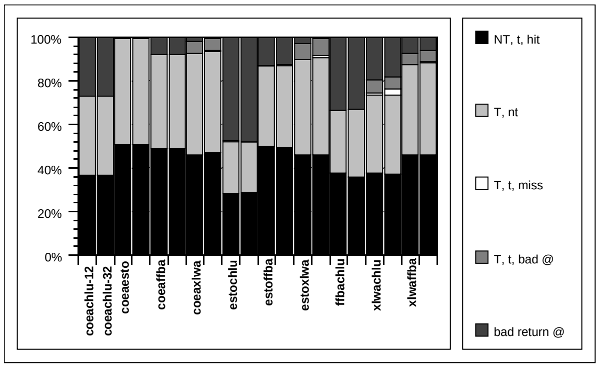

Figure 13: distribution of the mispredictions for gshare scheme

The variety of these results illustrates the difficulty to predict the behavior of a stack. The usefulness of the stack depends more of the dynamic interlacing of the *call* and *ret* instructions than of their static number. That's why even *call*-intensive individual benchmarks like *xlisp*, *ear* or *lu* have very distinct behaviors. One badly behaving benchmark, like *cholesky* (for which bad return addresses with a 32-deep stack still represent 80% of the mispredictions), can pull down the whole prediction accuracy. However, in most cases, increasing the stack size (keeping a reasonable limit) offers very small enhancement. Simulations with 64-deep stacks still showed no gain over the previous misprediction ratios.

Prediction tables sizes For all the previous simulations, we fitted the size of the tables according to the number of threads the architecture simultaneously supports. To evaluate the capacity of the branch prediction schemes to stand pollution, we executed the benchmarks with fixed sizes for the BTB and the PHT. Figure 14 shows the branch misprediction ratios obtained for the simultaneous execution of four threads with the BTB and the PHT having respectively, 512 and 4096 entries. Compared to figure 8, the observed evolution is small. The previously observed global behavior is preserved, and the resizing of the prediction tables results only in a small increase of the misprediction ratios.

Summary Interprating the different figures is difficult and a representation of average behaviors can help synthetize results. In figure 15, the average ratios are given for the three branch prediction strategies, according to the number of simultaneous threads and the sizes of the tables. Each average ratio is calculated as the misprediction which could have been obtained if the corresponding ten workloads were executed sequentially.

These average mispredictions corroborate the previously deduced results. The *gshare* branch strategy is the best one, showing nearly 25 per cent and 50 per cent increase in accuracy over *gselect* and *2bit*. As the number of threads increases, the *2bit* and *gselect* strategies suffer very little pollution while the *gshare* scheme appears to gain a little bit.

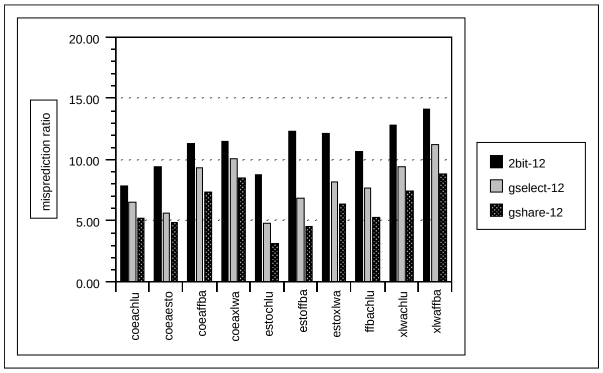

Figure 14: 4 threads, BTB of 512 entries, PHT of 4k entries

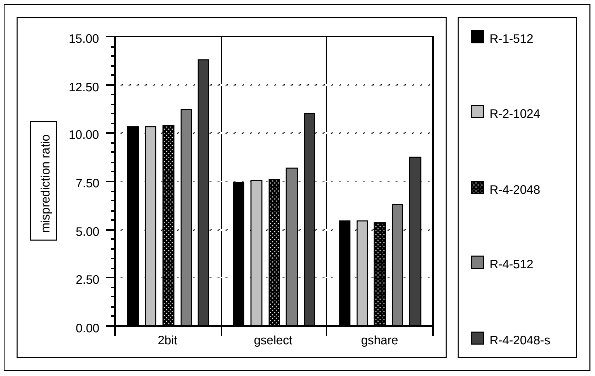

Figure 15: average misprediction ratios

The key *R-4-2048-s* in figure 15 refers to the misprediction ratio exhibited by the execution of four threads without RAS. The lack of stacks causes a strong decrease of the three branch prediction accuracies. For the *gshare* scheme, the penalty attained 60%, which confirms the need of a RAS per context.

If the sizes of the tables are maintained to those of a 1 thread predictor, while running several threads, a small increase of the misprediction ratios is noticeable. The distribution of the mispredictions resulting from the simultaneous execution of four threads and corresponding to the *gshare* scheme in figures 8 and 14, is given in figure 16. These distributions are quite similar for the two other prediction schemes. From this figure, it appears clearly that the greater number of mispredictions comes mostly from an increase of the number of BTB misses (represented in white on the bars).

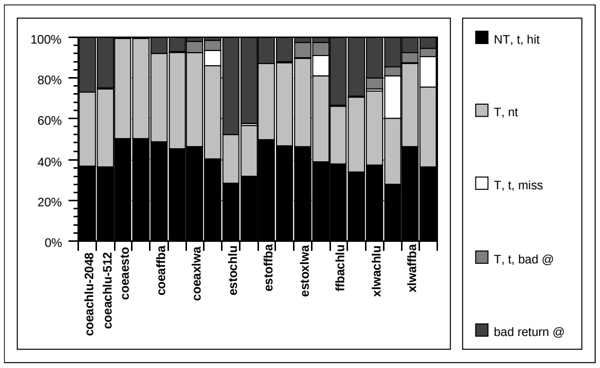

Figure 16: mispredictions distribution according to the PHT/BTB sizes

6 Parallel Workload

In the previous section, we have examined the behavior of completely independent processes. However, a multithreaded processor is also expected to execute in parallel threads from a single application. Thus, it is essential to evaluate the impact of sharing on the predictions.

For this purpose, we only used applications from SPLASH2 series (barnes, cholesky, fft, lu, radix, raytrace, volrend and watern). We ran the programs with 1, 2 or 4 threads. To obtain balanced threads, the parallel activity of each application has to be studied as it is dependant of the configuration (especially for the length of the initialization phase, which is generally done by a single thread). The pattern of branch accesses are different for the different configurations of a same application and the distribution of static branches for the execution of two threads is not exactly the same as for 4 threads.

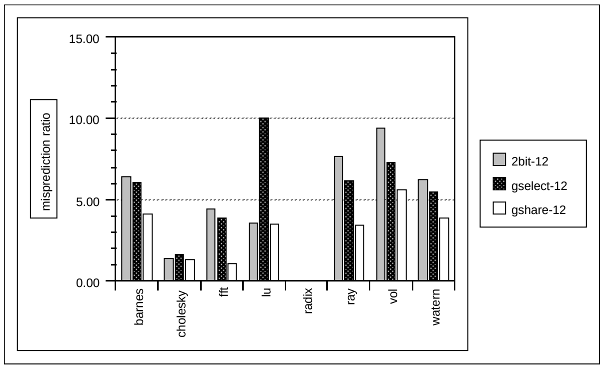

Figure 17: 1 thread, BTB of 512 entries, PHT of 4k entries

A look at figure 17 confirms what ones know about the behavior of the predictors on a single thread. However, a few remarks are necessary here.
First, the *gselect* scheme has a very bad prediction accuracy with the *lu* benchmark. This comes from the inability of this scheme here, to create and use regular history patterns, which results in a very poor branch direction prediction accuracy (only 86% when the two other schemes reach 96%).

For the *radix* benchmark, the prediction accuracy with any of the three prediction strategies is nearly hundred per cent. Although its number of static branches is not the smallest of the different benchmarks, the number of dynamic branches is very low (less than 5%). 100% of the branches are taken, which allows nearly 100% of good branch direction prediction. Moreover, nearly 90% of the branches are *call* and *ret* instructions. Without a stack, the misprediction ratio would have jumped to more than 40%.

Finally, the results, for *barnes* for example, are different from those given by the figure 6 because we are not in the same section of the program. It is not really important as what interests us is to compare the three predictors and the global evolution of their behavior when the number of threads vary.

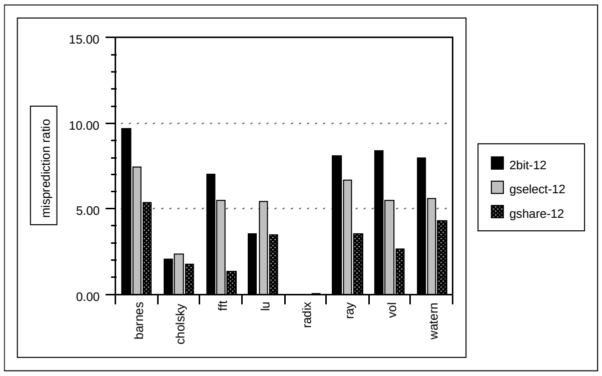

Figure 18: 2 threads, BTB of 1024 entries, PHT of 8k entries

Parallel processing Results from figures 17, 18 and 19 are more shaded than those obtained with a multiprogramming workload. As we have seen previously, one can't observe a general increase or decrease of the misprediction ratios. When the number of threads increases, branch misprediction ratios evolve very differently according to the benchmarks. Thus, for the benchmark *barnes*, the prediction accuracy decreases with two simultaneous threads and increases with four. For the benchmark *ray*, the misprediction ratio increases as the number of threads increases. For *lu*, with two threads, the misprediction ratio is divided by two for *gselect*, but stays unchanged for the two other strategies. When there are four threads to be executed, *gselect* does not evolve any more but the two other strategies

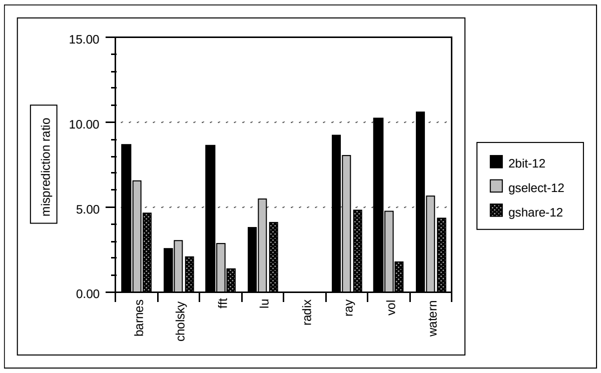

Figure 19: 4 threads, BTB of 2048 entries, PHT of 16k entries

accuracy decreases. Finally, *volren*, *gshare* and *gselect* strategies give better results when the number of threads increases, but not *2bit*.

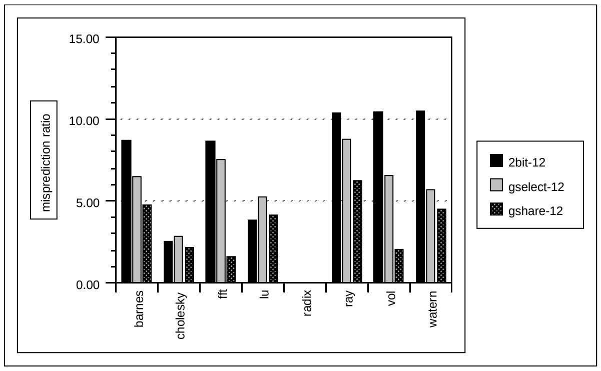

Figure 20: 4 threads, BTB of 512 entries, PHT of 4k entries

Contrary to the multiprogramming study, here, one cannot evaluate the misprediction ratios that could have been obtained by a sequential execution of the threads composing a benchmark. This increases the difficulty to compare results. The threads issued by the parallel execution of an application are expected to produce similar branch patterns. To evaluate a possible capacity of the predictors to take advantage of this potential sharing, one can reduce the sizes of the prediction tables. Figure 22 shows the misprediction ratios when four threads are executing simultaneously and the PHT/BTB have respectively only 512/2048 entries. Here, as expected, the results are either identical, or worse. The case of *lu* is interesting. The misprediction ratio of *gselect* is the same as when the BTB/PHT had bigger sizes and is half of the one obtained for one thread. A look at the causes of this better accuracy shows that it is mostly due to a better branch direction prediction. From figure 22, one can deduce that this doesn't come from an increase of the PHT size, but only from the change in the dynamic interlacing of the branch instructions.

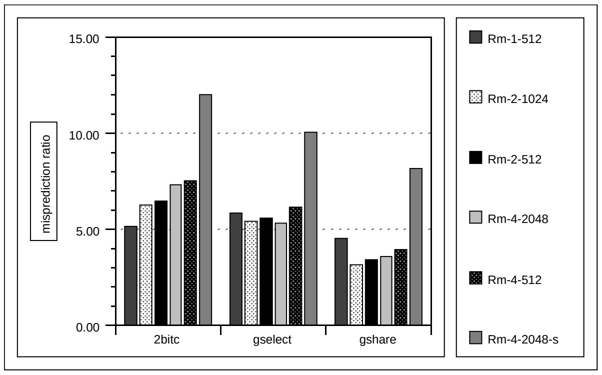

Figure 21: average misprediction ratios

Summary Figure 21 gives a synthetic view of all the previous results. The keys refer to the number of simultaneous threads and the size of the BTB. In a general manner, the *gshare* scheme still performs better than *gselect* which is better than *2bit*. Moreover, it is clear here that the three prediction strategies don't have the same capacity to take advantage of the execution of parallel applications.

The *2bit* scheme reveals significantly decreasing prediction accuracy when the number of threads increases. The misprediction ratio for four simultaneous threads is worse by 25% than for one thread. On the other hand, it suffers less from pollution when it has small tables.

gselect seems to be able to benefit from parallelism as there is a small decrease of mispredictions when the number of threads increases. However, degradation of accuracy is more important here if the tables are not well sized.

gshare is the prediction strategy which appears to take the most advantage of parallelism. There is a great decrease of the average misprediction rate when two threads of a same application are executing simultaneously. However, this gain is smaller with four threads and when the sizes of the tables are not proportionnal to the number of threads.

Figure 22 gives the distribution of the mispredictions when the sizes of the PHT/BTB are either proportionnal or not to the four threads executing simultaneously. This shows that the misprediction increase comes mostly from BTB misses (represented in white on the bars).

In figure 21, the key *Rm-4-2048-s* corresponds to the misprediction ratios obtained while executing four threads without having one Return Address Stack per thread. The misprediction ratios appear to be very bad, and for *gshare*, the accuracy is more than twofold worse

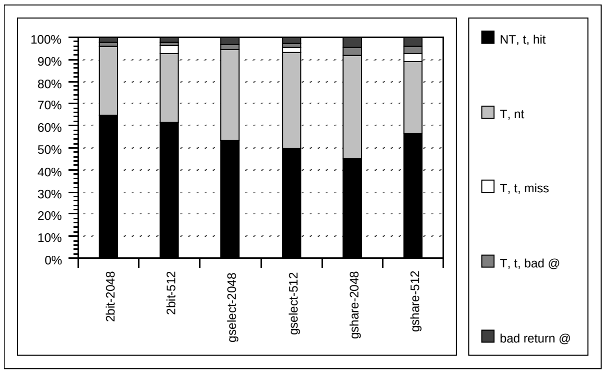

Figure 22: average mispredictions distribution

than the one obtained with RAS. This confirms the necessity of having one RAS per context in a simultaneous multithreaded microprocessor.

7 Concluding Remarks

In this paper, we examined the behavior of different branch prediction strategies while executing several threads of instructions simultaneously. We chose to simulate a simple mechanism based on 2-bit counters associated with entries of a BTB and two more complex strategies based on the 2-level adaptive branch prediction mechanism proposed by Yeh & Patt. These last two strategies use a PHT index constructed as either the concatenation or the XOR of an history register and the branch address.

We studied the impact of the addition of one Return Address Stack per hardware context. We showed that a 12-deep stack per thread is sufficient to enhance greatly the accuracy of branch prediction while adding a minimal implementation cost.

We also explored the behavior of the branch predictors when independant applications are running simultaneously and when the workload is constituted by a parallel program. Our simulations showed that in multiprogramming environment, if the sizes of the tables (PHT/BTB) are proportionnal to the number of active threads, there are very few interactions, be they destructive or constructive. If the sizes of the tables are kept small, there is a slight increase of the mispredictions, which is mostly due to an increase of the conflicts in the BTB.

With parallel workloads, we could have expected a beneficial sharing effect. In fact, it is very dependant of the branch predictors. The simple *2-bit* predictor behaves very badly when the number of threads increases. *gselect* and *gshare* seem to take a small advantage of the execution of threads from the same application. This is particularly true for *gshare* with 2 threads. For the three predictors, the decrease of the size of the tables has a bad effect on prediction accuracy by increasing the number of BTB misses.

Finally, the respective behavior of the three branch prediction strategies appear to be the same whatever may be the number of simultaneously executed threads and the type of workload. *gshare*, by far, has the better accuracy, with *gselect* following and *2-bit* performing the worst.

The next phase of our study will consist in evaluating the real branch penalty induced by mispredictions. More than branch predictor accuracy, it would be interesting to know what is the cost of bad branches, in terms of number of wasted issue slots between a branch instruction and the next valid instruction. This penalty, which is very important on superscalar architectures, could be very reduced when instructions from several threads are available.

References

[Aga92] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE transactions on parallel and distributed systems*, 3(5):525–539, September 1992.

[Cor94] Digital Equipment Corporation, editor. *Alpha 21164 Microprocessor Hardware Reference Manual*. September 1994. preliminary edition.

[Dix92] K.M. Dixit. New cpu benchmark suites from spec. *COMPCON*, pages 305–310, spring 1992.

[Far91] M. K. Farrens. Strategies for achieving improved processor throughput. In *18th International Symposium on Computer Architecture*, pages 362–369, May 1991.

[JSL95] Stephan Jourdan, Pascal Sainrat, and Daniel Litaize. Exploring configurations of functional units in an out-of-order superscalar processor. In *22nd International Symposium on Computer Architecture*, pages 117–124, June 1995.

[LOG90] LSI LOGIC, editor. *SPARC Architecture Manual (Version 7)*. 1990.

[LS84] Johnny K.F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[McF93] Scott McFarling. Combining branch predictors. TN 36, Digital Western Research Lab., June 1993.

[PSR92] S.T. Pan, K. So, and J.T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *proceedings of ASPLOS V*, pages 76–84, October 1992.

[Smi81] James E. Smith. A study of branch prediction strategies. In *8th International Symposium on Computer Architecture*, pages 135–147, June 1981.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20:5–44, March 1992.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximising on-chip parallelism. In ACM, editor, *22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[WG89] W.D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *16th International Symposium on Computer Architecture*, pages 273–280, 1989.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash2 programs: Characterization and methodological considerations. *22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[YP92a] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. *19th International Symposium on Computer Architecture*, pages 124–134, May 1992.

[YP92b] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *25th International Symposium on Microarchitecture*, pages 129–139, December 1992.

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