SYRANT: SYmmetric Resource Allocation on Not-taken and Taken Paths

Nathanaël Prémillieu, André Seznec

INRIA

SYRANT: SY*mmetric Resource Allocation on Not-taken and Taken Paths*

Nathanaël Prémillieu — André Seznec

N° 7463

November 2010

Architecture and Compiling

R *apport de recherche*

SYRANT: SYmmetric Resource Allocation on Not-taken and Taken Paths

Nathanaël Prémillieu, André Seznec

Abstract: In the multicore era, achieving ultimate single process performance is still an issue e.g. for single process workload or for sequential sections in parallel applications. Unfortunately, despite tremendous research effort on branch prediction, substantial performance potential is still wasted due to branch mispredictions. On a branch misprediction resolution, instruction treatment on the wrong path is essentially thrown away. However, in most cases after a conditional branch, the taken and the not-taken paths of execution merge after a few instructions. Instructions that follow the reconvergence point are executed whatever the branch outcome is.

We present SYRANT (SYmmetric Resource Allocation on Not-taken and Taken paths), a new technique for exploiting control independence. SYRANT essentially uses the same pipeline structure as a conventional processor. SYRANT tries to enforce the allocation of the exact same resources on the out-of-order execution mechanism (physical register, load/store queue and reorder buffer) for both the taken and not-taken paths. Thus, on a branch misprediction, the result of an instruction already executed on the wrong path after the reconvergence point can be conserved in the same structure when it is data independent. Adding SYRANT on top of an aggressive superscalar execution core allows to improve performance for applications suffering a significant branch misprediction rate.

As a side but important extra contribution, we introduce ABL/SBL a simple and non-intrusive hardware reconvergence detection mechanism. ABL/SBL can be used in a conventional superscalar processor to improve branch prediction accuracy through exploiting the execution of branches along the wrong path.

Key-words: Architecture, sequential, multicore, manycore, control independence, control flow reconvergence, SYRANT

SYRANT : allocation de ressources fait de manière symétrique sur le chemin pris et non pris

Résumé : De plus en plus d'avancées dans le domaine de l'architecture des processeurs sont basées sur l'exécution spéculative des instructions. Dans le cas particulier de la spéculation liée aux branchements, le chemin d'exécution pour le branchement pris et le branchement non pris reconverge souvent après quelques instructions. Ce phénomène est appelé reconvergence du flot de contrôle. Il en résulte que ces instructions sont exécutées quelque soit la direction du branchement. On parle alors d'indépendance de contrôle. Plusieurs techniques ont été précédement proposées pour l'exploiter. Elles sont étudiées dans ce rapport.

Une nouvelle technique exploitant l'indépendance de contrôle est présentée dans ce rapport. Nommée SYRANT (SYmmetric Resource Allocation on Not-taken and Taken paths), elle consiste à forcer l'allocation des mêmes ressources sur le chemin pris et non pris. Ainsi, lors d'une mauvaise prédiction, le travail déjà effectué sur le mauvais chemin et situé après le point de reconvergence peut être conservé.

Mots-clés : Architecture, séquentiel, multicœurs, massivement multicœurs, indépendance de contrôle, reconvergence de flots, SYRANT

1   Introduction

Each core in a modern multicore is a superscalar processor [19], and while parallelism is the avenue to increase peak performance, poor parallelism in many applications and Amdahl's law [3] are pushing to continue the research on improving superscalar architectures [11]. In particular, while for parallel or multiprogrammed workloads energy consumption is a major issue, this constraint is much less important on sequential workloads since only one core is active. Therefore hardware mechanisms to improve single process performance particularly makes sense if they can be easily powered down when a multiprogrammed or parallel workload is executed. SYRANT, the major proposition presented in this paper, should be considered in this context.

Any gain on branch prediction accuracy results in a performance gain on a superscalar processor. Unfortunately since 2006 [18], the conditional branch prediction accuracy seems to have reached a plateau. Other techniques are needed to improve the superscalar processor performance, for instance exploiting control flow reconvergence [16, 8, 2, 5]. After a conditional branch, the taken and the not-taken paths of execution of a branch often merge after a few instructions (Figure 1).

In case of a branch misprediction, substantial work concerning the instructions subsequent to the reconvergence point might have been executed before the branch misprediction is resolved and execution resumes on the correct path.
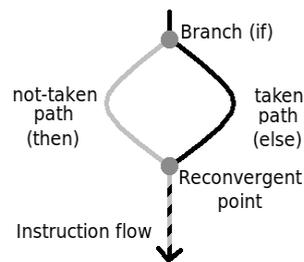
Figure 1: Illustration of the point of reconvergence and the merge of the taken and the not-taken path of a branch.

Instructions that follow the reconvergence point are executed whatever the branch outcome is. They are referred as control independent (CI) instructions [2]. If the operands of a CI instruction are independent on the executed path then its result is also independent on the path. These instructions are called Control Independent Data Independent (CIDI) instructions [2]. While the standard pipeline correction mechanism flushes all the instructions after a mispredicted branch, the objective of exploiting control

independence is to save the results of CIDI instructions and use them without reexecuting the instructions.

Control independence has already been considered in the literature. Several proposals are trying to exploit it to gain performance. A major study on control independence [16] has shown that it can be used to reduce the performance losses due to branch mispredictions. Exploiting control independence on a subset of mispredicted branches [8] is already sufficient to have performance gain. As exploiting control independence often means to change how the instructions are executed, quite complex logic and costly hardware are needed [2]. Modifying the instruction sequencing order to favor the execution of control independent instruction was also proposed [5].

The first contribution of this paper is SYRANT, a new technique for exploiting control flow reconvergence that respects the major pipeline flow of a superscalar processor. SYRANT, SYmmetric Resource Allocation on Not-taken and Taken paths, tries to enforce the allocation of the exact same resources on the out-of-order execution mechanisms (physical registers, Load/Store Queue (LSQ) and ReOrder Buffer (ROB)) in the execution core. Thus on a misprediction, the work already executed on the wrong path after the reconvergence point can be conserved in the out-of-order execution storage structures (registers, LSQ).

One of the issues that we had to address in the design of SYRANT was the design of a cost-effective solution to detect the reconvergence point. We propose ABL/SBL for Active Branch List/Shadow Branch List for this purpose. As a side contribution, we show that as a stand-alone add-on in the instruction fetch engine, ABL/SBL can be leveraged to improve the branch prediction accuracy in an otherwise conventional superscalar processor. ABL/SBL records the computed directions on the wrong paths. We show that these informations can be leveraged to improve the prediction accuracy of a state-of-the-art predictor such as TAGE [18].

The remainder of this article is organized as follows. Section 2 provides background on control independence. Related work is discussed in Section 3. Section 4 presents the fundamental principles of SYRANT. Section 5 details the whole mechanism of SYRANT including ABL/SBL, our proposal for detecting the reconvergence point. Section 6 points out that ABL/SBL can be used as simple mechanism to improve the prediction of the branches following the reconvergence point associated with a misprediction. Performance evaluation framework and results are presented in Section 8. Finally, Section 9 concludes this study.

2   Control independence

2.1   Forms of control independence

A program can be seen as a flow of instructions that the processor executes in the sequential order. This execution path is defined by branch instructions along it. Conditional branches offer two possible paths for the execution: the taken and the not-taken paths. These two paths merge after a few or a few tens of instructions for most of the conditional branches. This is called control flow reconvergence. In many cases, the reconvergence point of conditional branch can be uniquely determined. Compilers often exploit this property to perform some optimizations. Hence, after such a reconvergent branch, one can distinguish between control dependent (CD) instructions , which execution depends on the outcome of the branch and the control independent (CI) instructions that are executed whatever the branch outcome is.

Control independence after control flow reconvergence can be used to partially hide a branch misprediction penalty. As instructions are executed out-of-order, the instruction flow can reach the reconvergence point, starting to treat CI instructions before a misprediction resolution. Thus, instead of flushing all the pipeline to recover from a misprediction, one can try to save the work done by these instructions.
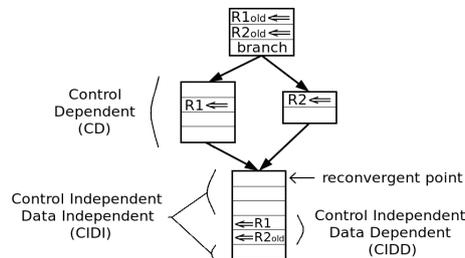
Figure 2: Illustration of control dependence, control independence, control independence data dependence and control and data independence [2].

Unfortunately the results of all CI instructions are not valid on both paths. Data dependencies between CD and CI instructions can arise. As illustrated in Figure 2, if a CI instruction has used a data operand produced by an incorrect CD instruction then its result computed on the wrong path is invalid (false data dependence). If a CI instruction has used an old data operand that is modified by a correct CD instruction then its result is also invalid (true data dependence). CI instructions can be divided in Control Independent but Data Dependent (CIDD) and Control Independent and Data Independent (CIDI). Only results of CIDI instructions are worth to be saved, the other instructions have to be re-executed.

2.2   Issues of control independence

To exploit control independence, one has first to be able to discriminate between CD and CI instructions, i.e., to identify the reconvergence point. Software (compiler or profiling) detection of the reconvergence point was proposed in previous studies [16], but it induces extensions of the ISA. In this study, we will rely on an hardware detection that preserves binary compatibility.

Once the reconvergent point is detected, preserving the results of the control independent instructions already executed is a major issue. In an out-of-order execution superscalar processor, the results, and also the dependency chain of the not committed instructions are stored in the out-of-order execution hardware resources: physical registers, LSQ and ROB. Entries in these structures are dynamically allocated by the front-end of the processor pipeline in the instruction fetch order. On a branch misprediction, these entries are simply deallocated and put back in the list of free entries. Therefore, in most cases, the dynamic allocation within these structures is completely different on the taken and on the not-taken paths. Exploiting control independence after a branch misprediction resolution necessitates to find some hardware mechanisms to save the contents of physical registers, LSQ entries and ROB entries for wrong path control independent instructions as well as some simple solutions to retrieve these data when executing the correct path.

However, saving the results of control independent instructions is not sufficient. One has also to discriminate CIDI and CIDD instructions. Only CIDI instruction results can be conserved: the result of a CI instruction executed on the wrong path can be conserved only if its dependency chain does not include any control dependent instruction on the wrong path as well as on the right path.

Therefore exploiting control independence necessitates 1) to discriminate between CD and CI instructions 2) to save CI results and dependency chains 3) to correctly determine CIDI instructions.

3   Related work

Exploiting control independence has been considered in several previous studies. Potential performance benefits could be drawn from this concept. Several hardware techniques have been proposed to exploit its potential.

Rotenberg, Jacobson and Smith have studied [16] the potential of control independence in details. This work highlights the fact that the major penalty for mispredictions come from "the wasted resources consumed by incorrect CD instructions". However they also showed that exploiting con-

trol independence can yield to a gain up to half of the gain brought by perfect branch prediction.

[16] also proposed an hardware implementation exploiting control independence. They dealt with reconvergence point detection through software analysis; adding some bits in the ISA to encode the necessary information. To address the data dependencies, the first steps of the execution of each instruction are replayed. If there is a difference in the source registers, the instruction must be re-executed. For memory access instructions, any change in the order of the memory accesses is detected. This leads to select the loads that have to be re-executed. The complete replay mechanism is derived from the trace processor described in a previous work [15].

Gandhi, Akkary and Srinivasan proposed a technique called Selective Branch Recovery (SBR) [8] that try to exploit control independence. In order to limit hardware complexity, they consider only the particular set of branches represented in Figure 3, i.e., the predicted not-taken *if-then* construct but without the *else* statement. On a misprediction, there is no extra CD instructions to be executed before the reconvergence point. The main issue remains to discriminate between CIDD and CIDI instructions.
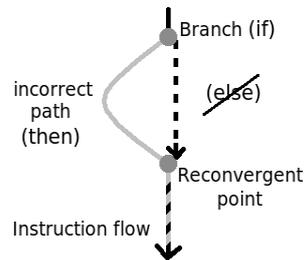

Figure 3: Exact convergence: no control dependent instructions on the correct path.

Cher and Vijaykumar [5] proposed an alternative approach to exploit control independence. They considered the main conclusion of the study of Rotenberg, Jacobson and Smith [16] as the starting point for their work. Their Skipper architecture simply skips the CD instructions until the branch is resolved, concentrating the execution on the CIDI instructions thus avoiding the waste of resources due to the execution of incorrect CD instructions and CIDD instructions. Once the branch is resolved, the correct CD instructions are fetched and executed. This ensures that only correct instructions are executed. However, Skipper induces important modifications of a superscalar core. Skipping over instructions means that instructions are fetched out-of-order. Skipper creates a gap in the instruction window, large enough to put the correct CD instructions once the branch is resolved. Moreover, as it means that essential resources are reserved, the process is

only used for branches difficult to predict in order to limit the amount of resources used. Difficult-to-predict branches are identified through the use of the JRS confidence predictor [13]. The different information needed by the architecture, like reconvergence point, resources consumption, but also CIDI information are gathered from previous dynamic executions of the branches. If these information are erroneous, Skipper simply squashes all the CI instructions and restarts the execution after the execution of the correct CD instructions.

Hilton and Roth proposed an new approach called Ginger [12]. Ginger proactively protects a branch by keeping room for the correct CD instructions if the branch has to be corrected. Instead of re-fetching and re-renaming the CI instructions after the correct path CD instructions have been fetched, Ginger performs a "search-and-replace" operation on the register tags by replacing the wrong path mapping checkpointed by the one read from the current mapping table. Thus, they change the renaming information of the in-flight CI instructions, updating the all the pipeline structures with correct data dependencies information. Ginger necessitates to halt the pipeline to perform the search and replace operation. When the execution restarts, all the instructions with a modified renamed form are re-executed as they are identified as CIDD.

Al-Zawawi, Reddy, Rotenberg and Akkary proposed a technique called Transparent Control Independence (TCI) [2]. The key idea is to decouple the CIDI instructions from the CD and CIDD instructions during the execution of the CD instructions. TCI constructs a self-sufficient recovery program that is executed when the branch is mispredicted. The main structure of their design is a FIFO buffer called re-execution buffer (RXB) in which the CIDD instruction are stored with a copy of their source values if these values are supplied by CIDI instructions. When the processor has to recover from a mispredicted branch, the recovery program, constituted of the correct CD instructions followed by the CIDD instructions taken from the RXB, is executed. Therefore, the recovery is transparent for the processor, as it only executes instructions without having to cancel other instructions. TCI deals with all types of conditional branches. To detect reconvergence points, TCI uses a predictor proposed by Collins et al. [7]. Several modifications are made to the original predictor in order to gather the specific needed information. For example, an influenced register set (IRS) is collected for each branch. This IRS contains the registers that will be used as destination by the CD instructions. CIDI instructions are detected through the use of the IRS.

TCI exhibits a quite high degree of complexity, both in logic and storage structures of a conventional superscalar processor. For instance, a major modification of the pipeline execution core is needed with the two possible sources of instructions, the conventional instruction fetch pipeline and the re-execution buffer.

All these proposals have shown that exploiting control independence is promising to reduce branch misprediction penalty. However, for some of them, heavy hardware modifications and complex logic are needed. This strongly modifies the pipeline structure. In contrast, with the SYRANT proposal, we try to exploit control independence essentially respecting the main structures of an out-of-order execution pipeline.

4   SYRANT, SYmmetric Resource Allocation on Not-taken and Taken paths: resource allocation principle

In Section 2.2, we have pointed that on a misprediction for a given control independent instruction, two different sets of entries are successively allocated in the ROB, the register file and the LSQ. In order to exploit control independence, information (dependencies, register values, etc) must be preserved (e.g. copied) on misprediction detection and retrieved (on right path execution). This may lead to complex design inducing a lot of copying.

Our proposal SYRANT, SYmmetric Resource Allocation on Not-taken and Taken paths, turns around this difficulty through enforcing the allocation of the exact same entries in the main structures of the out-of-order execution pipeline on the taken and the not-taken paths: that is, on the taken and not-taken paths, a given CI instruction will be allocated the same physical register, the same ROB entry and if it is a load/store instruction the same LSQ entry.

To enforce this symmetric allocation, SYRANT inserts gaps in the structures to enforce both paths use the same number of physical registers, the same number of ROB entries and the same number of LSQ entries. Thus, at the reconvergence point, the pipeline has used exactly the same number of resources on both paths. After the reconvergence point, a CI instruction already renamed on the wrong path will be allocated the same physical register, the same ROB entry and the same LSQ entry for memory instructions. Then the information (dependencies, results, etc) associated with the CI instruction on the wrong path are available on the right path to be processed.

Figure 4 illustrates the gap mechanism for physical registers. Here, the taken path requires 2 registers ($P5$ and $P6$) and the not-taken path requires 5 registers ($P2$ to $P6$). Through "wasting" 3 registers ($P2$, $P3$ and $P4$) on the taken path, we ensure that the same physical registers will be used on both paths after the reconvergence point.

Enforcing the allocation of the same resources for CI instructions on both paths is only possible if the volumes of resource used on both paths are
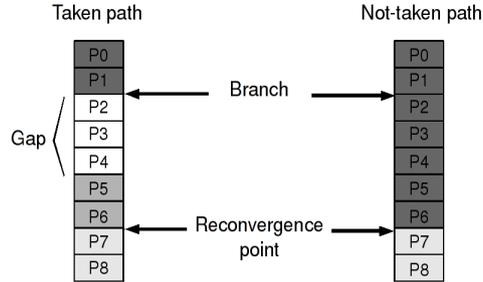
Figure 4: Modification of the registers renaming mechanism: a gap is introduced on the less demanding path in order to use the same registers on both taken and not-taken paths for the control independent instructions.

known. In Section 5.1, we introduce a simple reconvergence point detection mechanism and its use to monitor resource needs. Once the resource needs are known for both paths from the previous dynamic instances of a branch, they can be used on the next occurrence of this branch to create gaps of the appropriate sizes.

As already pointed out, only CIDI instruction results must be preserved when executing the right path. Therefore, data dependencies must be identified: register dependencies and memory induced dependencies. We detail this process in Section 5.2.

5   SYRANT: detailed description

In this section, we detail the principal mechanisms in SYRANT, first the reconvergence detection mechanism, then the dependency enforcing mechanisms.

5.1   Reconvergence point detection

In order to compute the resources needed on the taken and not-taken paths, the reconvergence point must be detected. However in practice, the knowledge of the effective resource needs is not required but rather the difference between the resource needs on the two paths, i.e., the sizes of the gaps. Therefore rather than detecting the precise reconvergence point, which would require to compare every instruction of the right path with every instruction on the wrong path we choose to detect the first branch after the reconvergence point as described below.

Detecting reconvergence   For the reconvergence point detection, three hardware structures are used. The Active Branch List (ABL) is used to

record the branches on the path currently fetched (Figure 5.a). On a mispre-diction resolution, all the branches on the wrong path in the Active Branch List are copied in the Shadow Branch List (SBL) (Figure 5.b). Branch stor-age in the ABL is resumed after the branch misprediction resolution. Each new fetched branch is compared against the content of the SBL. The first match indicates the reconvergence point (Figure 5.c).
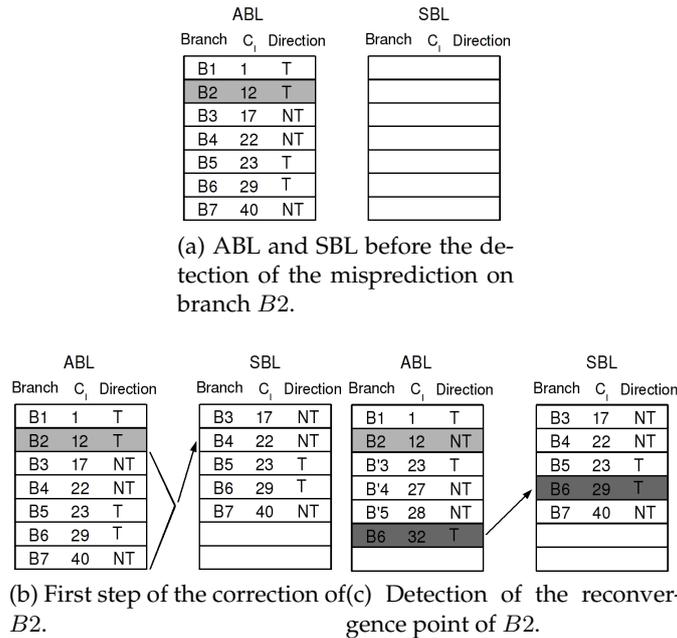
	ABL				SBL	
Branch	C_i	Direction		Branch	C_i	Direction
B1	1	T				
B2	12	T				
B3	17	NT				
B4	22	NT				
B5	23	T				
B6	29	T				
B7	40	NT				

(a) ABL and SBL before the de-tection of the misprediction on branch $B2$.

ABL			SBL			ABL			SBL		
Branch	C_i	Direction	Branch	C_i	Direction	Branch	C_i	Direction	Branch	C_i	Direction
B1	1	T	B3	17	NT	B1	1	T	B3	17	NT
B2	12	T	B4	22	NT	B2	12	NT	B4	22	NT
B3	17	NT	B5	23	T	B'3	23	T	B5	23	T
B4	22	NT	B6	29	T	B'4	27	NT	B6	29	T
B5	23	T	B7	40	NT	B'5	28	NT	B7	40	NT
B6	29	T				B6	32	T			
B7	40	NT									

(b) First step of the correction of $B2$.   (c) Detection of the reconver-gence point of $B2$.

Figure 5: Monitoring process during the correction of a branch.

ABL entries and SBL entries are identical (Figure 6.a). An entry al-lows to identify a branch and to record the amount of resources needed on the path. It consists of the PC of the branch, the number of registers that have been used before the branch, the number of instructions fetched before the branch, the number of LSQ entries used and the direction of the branch (taken or not-taken). Therefore upon the detection of the reconver-gence point, one can determine the resource gaps by simply computing the difference between the different fields for the current ABL entry and the matching SBL entry. However, note that the computed resource gaps are including the gaps consumed by inner reconvergent branches (see Figure 7).

Using reconvergence   When detecting a reconvergence, the resource gaps associated with the reconvergent branch are computed. This information is stored in the Resource Allocation on Not-taken and Taken paths (RANT)

table. A RANT entry (Figure 6.b) consists of the branch PC and the signed value of the gaps for physical registers, ROB entries and LSQ entries.

At instruction fetch, the RANT table is checked; on a hit, the gap insertion mechanism can be activated. In Section 7, we will show that gap insertion is not always beneficial, but it can be conditionally activated.
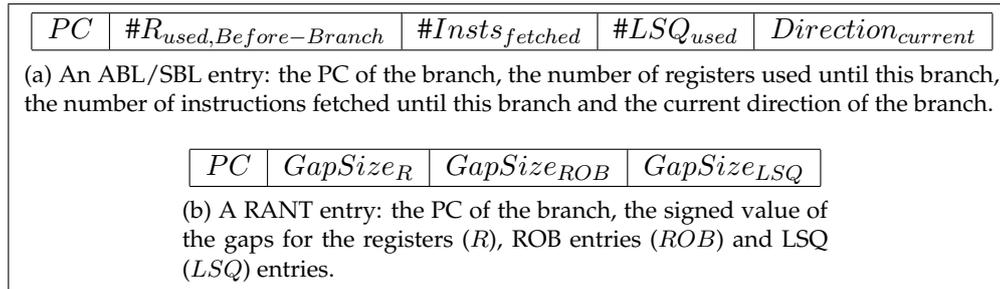
PC	$\#R_{used,Before-Branch}$	$\#Insts_{fetched}$	$\#LSQ_{used}$	$Direction_{current}$

(a) An ABL/SBL entry: the PC of the branch, the number of registers used until this branch, the number of instructions fetched until this branch and the current direction of the branch.

PC	$GapSize_R$	$GapSize_{ROB}$	$GapSize_{LSQ}$

(b) A RANT entry: the PC of the branch, the signed value of the gaps for the registers (R), ROB entries (ROB) and LSQ (LSQ) entries.

Figure 6: ABL/SBL entry, RANT entry

(a) ROB allocation after the correction branch $B1$, first time misprediction.

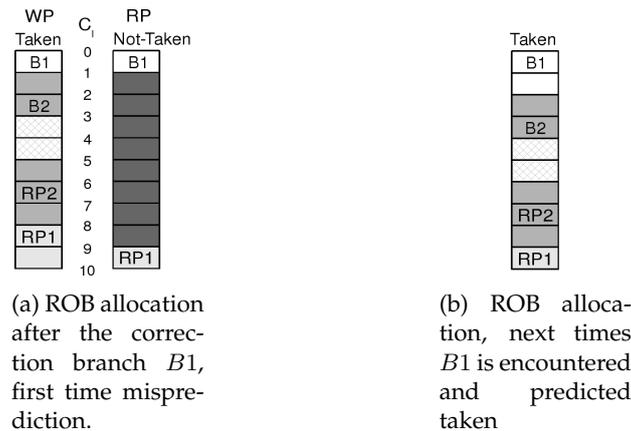(b) ROB allocation, next times $B1$ is encountered and predicted taken

Figure 7: Monitoring process of the same branch at different times

5.2 Identifying Control Independent Instructions and Enforcing Data Dependencies

Only CIDI results must be preserved. The result of a CI instruction can be data dependent through two distinct channels: register dependency, i.e, one register operand is computed differently on the correct path and memory induced dependency. Thus, CI instructions have to be register independent (RI) but also memory independent (MI).

5.2.1   Identifying Control Independent Instructions

The reconvergence point detection presented above associated with gap insertion should enforce that after a misprediction control independent instructions are allocated to the same ROB entry it was already allocated on the wrong path. Therefore detecting a CI instruction is straightforward. The instruction is checked against the occupant of its assigned ROB entry. If it is found that the instruction already present in the ROB entry is the same as the one to be pushed in, then it is a control independent instruction.

Although the real reconvergence point of a branch can be before the one that is detected, SYRANT is still able to identify the instructions between these two points as CI instructions. As these instructions are present on both paths (because they are CI instructions), the resources they consume are counted on both paths. Therefore, the size of the gap that is computed is the exact difference of resources consumed only by CD instructions on both paths, not the consumption between the branch and its detected reconvergence point. As a result, if a gap is made, all the CI instructions will have the same allocated resources, even the one between the real and the detected reconvergence point. Thus all CI instruction can be identified, as long as the size of the gap is correct.

Note that even when gaps are inserted the lengths of the wrong path and the correct paths may not match. In that case, the identification of the CI instructions just fail. The pipeline just continue to act as usual potentially missing some performance gain opportunities, but missing them does not bring performance degradation.

5.2.2   Identifying and Propagating Register Dependencies

The renaming process is in charge of preserving the results of already executed CIDI instructions, but invalidating the results of CIDD instructions and CD instructions.

After a misprediction, the instruction fetch is resumed and fetched instructions are checked against the wrong path instructions occupying their entries in the register file, the ROB and the LSQ. To assess the validity of the data already present in these structures, different rules are applied. For CI instructions other than load instructions, the validity of the result of the instruction must be conserved if the operands of the instruction remain valid on the correct path.

A difficulty is that different versions of the data operands can be successively available in the same physical register and for the same successive instances of the instruction: register P1 can have been valid on the wrong path allowing to execute instruction I2, then discovered as invalid on the right path thus the operand for I2 is invalid, however if I1 is executed,

register P1 becomes valid again. To ensure the use of the correct operand version, we propose the tagging process for identify the correct version of the data described below.

We refer to a sequence of instructions that are fetched, decoded and renamed without any interruption by the correction of a branch misprediction or a load/store dependency as a rename sequence. A unique RS-tag (Rename Sequence tag) is associated with any rename sequence. Basically, this tag is used to determine when the information associated with the instruction has been computed.

The register renaming process acts as follows to preserve CIDI work: after a misprediction, the current RS-tag is changed (incremented for instance). At renaming, a RS-tag is associated with each instruction in the ROB and with its destination register in the map table. For instructions other than load instructions, the following rules are applied:

1. **if** the PC of the new instruction is different from the PC of the old instruction in the same ROB entry then store the new RS-tag both in the ROB entry and the register map table and mark the register as invalid and the instruction as unexecuted.

2. **else**:

   - **if** the instruction does not read any register operand but produces a result then keep the old RS-tag and preserve the register validity and execution status.

   - **if** the instruction reads operands which names after renaming, including the RS-tags are identical to the ones from the wrong path then conserve the old RS-tag, the register valid bit and the execution status **else**
     store the new RS-tag both in the ROB entry and the register map table and mark the register as invalid and the instruction as unexecuted.

This process for all instructions except loads is illustrated on Figure 8. At first decode, Tag T is associated with the result of an instruction. After misprediction, Tag N is associated to CD instruction results as well as CIDD instruction results.

In order to propagate memory dependencies, load/store instructions require special treatments involving the LSQ. In the LSQ, the entry associated with a store will be marked as invalid, i.e considered as storing an invalid data, if the store does not match its associated LSQ entry or its ROB entry. In case of matches, the entry will also be marked invalid if either its load address operand or its write operand is invalid otherwise the validity of the wrong path execution will be preserved.
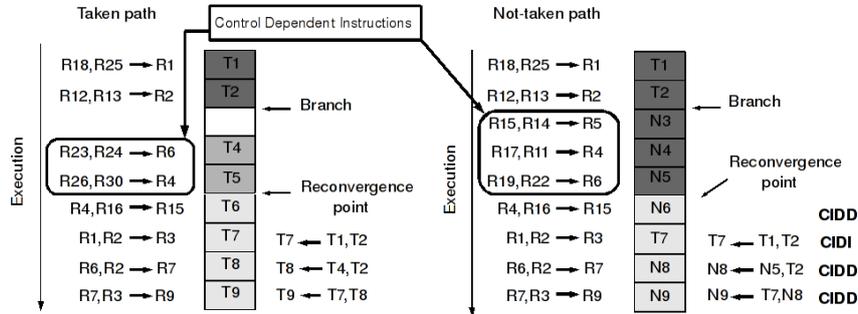
Figure 8: Illustration of the modified rename process and the mechanism to identify data dependent instructions. Ti means physical register number i with tag T. Ni means physical register number i with tag N.

To preserve the validity of the result of a CI load instruction, its address computation must be valid, i.e., the register operands must be valid. However, the validity of the load data depends also of the effective validity of data read on the memory: a load instruction can get either its data from the memory or from a non-committed store, i.e., data that is present in a LSQ entry. That is the load data can have been forwarded on the wrong path to the load by a store that is invalid on the right path. In order to handle this case, we implement an extra feature on the LSQ. When a data for a non committed store S is forwarded to a subsequent load L, the index of the entry associated with S in the LSQ is associated with L. When on the correct path, L passes the rename stage, validity of store S is checked in the LSQ. If the data associated with S is invalid then L is marked invalid (register and LSQ entry).

Important remark on the LSQ in SYRANT   On the execution of a store S, all the speculatively executed loads that follow the store S must be checked in order to verify that no memory dependency violation was done. As SYRANT is preserving wrong path results of CI loads that can be posterior to S, the results of these loads must also be invalidated in case of a memory dependence violation with S.

Memory dependence prediction   RAW hazards are costly in terms of performance. Thus, predictors are used to try to avoid them. Several predictors have been proposed in the literature: the synonym predictor [14], the store sets predictor [6] and the store barrier predictor [10]. These predictors try to identify loads that are dependent on some stores to issue them after the stores they depend on have been executed.

Our SYRANT implementation is compatible with these predictors and we use the store sets predictor in our simulator.

5.3 Continuing Wrong Path Execution After Branch Misprediction Resolution

On a conventional superscalar processor, there is no interest to continue the execution past the branch misprediction point. If one tries to exploit Control Independence, it becomes interesting to continue execution of the instructions, particularly CIDI instructions. Instructions that are on the wrong path are not totally flushed upon a misprediction detection. We refer to these instructions as phantom instructions. Phantom instructions continue their execution in the pipeline as the valid instructions, with a lesser priority than normal instructions. A phantom instruction is invalidated if one of its resources is reclaimed by the pipeline front-end for a valid instruction, hopefully its valid instance on correct path.

5.4 Artificially matching path lengths

When a branch is fetched, it is searched in the RANT table. Upon a hit, the corresponding gap size information are retrieved. Using these information, if needed, gaps are inserted on the less demanding path.

5.4.1 Gaps insertion

Gaps are inserted after a branch either after its initial fetch or after its misprediction resolution After the branch, the fetch and rename process continue as usual. In practice inserting a gap in the ROB, the free list physical registers or the LSQ is simply moving a pointer and leaving some entries free.

5.4.2 Recycling the resources

When a gap is inserted after a branch, the resources are reserved in a different way than by normal instructions. The associated resource needs to be recycled to avoid resource starvation. ROB entries and LSQ entries are very simple to recycle. These structures are circular buffers, the entries are freed the same order as they are allocated. Freeing entries is simply incrementing a pointer. For registers, all the gap registers must be recycled in the free list, when committing the branch instruction.

5.5 Using SYRANT for selective instruction invalidation

When a RAW memory dependency is violated, i.e., a load is executed prematurely and loads a wrong value, the complete chain of dependent in-

structions may have been executed or issued before the RAW violation is detected. All these instructions must be invalidated. Selective invalidation is a complex mechanism to implement in a pipeline and most processors simply flush the pipeline and rely on dependence prediction to avoid as many flushes as possible. SYRANT offers an intermediate implementation between ad-hoc selective invalidation preserving all the executed instructions and complete flush of the pipeline.

5.6 Hardware complexity considerations

SYRANT induces some modifications in the pipeline of a superscalar processor, but essentially the information flow of a conventional superscalar processor is respected. The major structures of the out-of-order execution pipeline are only marginally modified (RS-tag added to the register name in the ROB and index to retrieve the forwarding store in the LSQ). The monitoring process to compute the gap is the major cost with the introduction of the ABL, the SBL and the RANT table and of a few comparators in the front-end of the processor.

6 Using wrong path computed branches to improve branch prediction

The ABL/SBL structure proposed in Section 5.1 to detect the reconvergence point after a branch can also be used to keep the directions of the branches on the wrong path. This will obvious help in the context of the SYRANT proposal since it allows to directly exploit the computed CIDI branches for fetching on the corrected path. Interestingly the ABL/SBL structure can be useful per se even if the remainder of the SYRANT mechanisms are not implemented.

When a branch B has been computed on the wrong path, its computed direction is present in the SBL. If the ABL/SBL mechanism detects that the branch B is posterior to the reconvergence point then on re-fetch after branch correction, the pre-computed direction of branch B can be used for branch prediction instead of the usual branch prediction. It should be noted the ABL/SBL mechanism per itself is not able to discriminate between CIDD branches and CIDI branches. However, we found that, in many applications the quality of ABL/SBL prediction is better than the quality of the state-of-the-art TAGE branch prediction we use in our simulations. Moreover we found that this property can be monitored globally on the whole application with a single 4-bit counters.

In the remainder of the paper, we will refer to a prediction made using the information recorded in the SBL as a SBL prediction.

We would like to point out that the introduction of the SBL prediction in the pipeline is very local, since it does mot modify the global structure of any component of the superscalar processor, apart the branch predictor.

7 Limiting the size of gaps

Preliminary experiments showed that, for most applications, applying systematically SYRANT would lead to waste a huge amount of resources in the gaps, thus generally leading to performance losses. In our simulation framework, all benchmarks apart one were suffering performance losses.

Therefore we explored several techniques for limiting the number of gap insertions as well as their size based on their anticipated utility, on their anticipated moderate impact on performance if inserted on the correct path. The most useful filters of gap insertion are described below.

In order to limit the possible performance loss on the correct path, a first possibility is to insert the gaps if the branch was mispredicted. At decode time, it can not be determined that the branch will be mispredicted. By inserting gaps only upon the correction of a mispredicted branch, we only insert gaps when there is a chance to recover some useful work. However through this technique, gaps are only inserted if the mispredicted path was the most demanding path. We will refer to this gap insertion filter as *On Correction Only* filter. This strategy targets approximately the same branches that Selective Branch Recovery (SBR) [8] .

While gap insertion on the corrected path appears as natural, one can also use several indicators to assess the usefulness of gap insertion on the predicted path. Confidence on the branch prediction is a natural indicator. As a confidence estimator for the TAGE predictor [18], we use the provider component and the value of the prediction counter. The TAGE predictor was also modified as suggested in [17] in order to ensure a high misprediction coverage for low confidence predictions and a very low misprediction rate for the high confidence predictions. We will refer to this filter as the *Confidence* filter.

The quality of the reconvergence information is also important to assess if the gap insertion will be useful. For instance, one would like to insert gaps only if the information on the resource usage on taken and not-taken paths is stable enough, i.e., the reconvergence has been detected several times and the sizes of the gaps remained constant. It can be implemented as a stability counter associated with each RANT entry counting the number of times the branch has reconverged. If the size of the gap changes between two reconvergences, the counter is reset. Gaps are only inserted if the stability counter reach a threshold. We will refer to this filter as the *Stability* filter.

Limiting the size of each inserted gap is also a way to decrease the resource waste generated by the gaps. When the size of the gap is large, there is a high probability that control independent instructions will be data dependent. The gap is inserted only when its size is inferior to a threshold. We will refer to this filter as the *Size* filter.

Of course these filters can be also combined in order to further select the gap insertions that are the most likely to be useful.

8   Performance evaluation

A simulation study has been carried out for evaluating the SYRANT proposal. We derive our out-of-order simulator from the SimpleScalar framework [4].

8.1   Characteristics of the simulator

Unless otherwise noted, the simulator models a very aggressive 8-way superscalar processor with a 1024-entry ROB, a 512-entry LSQ and 2048 physical integer and floating point registers. We have chosen very large structures in order to maximize the number of in-flight instructions. The width of the different stages is set accordingly to fetch enough instructions before the detection of a misprediction in order to reach the reconvergence point of a maximum of mispredicted branches. For SYRANT, we use 256 entries on ABL and SBL, and 4K entries on the RANT table.

The processor also features a state-of-the-art conditional branch predictor, the TAGE predictor described in [18]. We model fetching up to two basic blocks per cycle with a maximum of 8 instructions. We use the store sets predictor [6] to predict memory dependencies. The minimum misprediction penalty is 20 cycles. The other characteristics are summarized in Table 1.

We will refer to this configuration as the base configuration (BASE).

8.2   Benchmarks

The benchmarks are part of the Spec 2006 benchmarks set [1]. As we have targeted the Alpha instruction set, we were only able to compile 18 of them. There are 11 integer benchmarks and 7 floating point benchmarks. The integer benchmarks are: *astar, bzip2, gcc, go, h264, hmmer, mcf, omnetpp, perl, quantum* and *sjeng*. The floating point benchmarks are: *lbm, leslie3d, milc, namd* and *povray*. To reduce the amount of simulation time, we use the Simpoint methodology [9] to summarize each benchmark in a set of 100 millions instructions slices. Each slice represents a certain part of the benchmark execution with a weight corresponding to the importance of this part

fetch/decode/issue commit width	8/8/8/8
ROB entries	1024
Rename registers	
LSQ entries	512
Cache data L1	64Kb 4-way set-associative
Cache instruction L1	64Kb 4-way set-associative
Cache shared L2	4Mb 8-way set-associative
Data TLB	4096-entry fully-associative
Memory latency	100 cycles
Integer ALU	6
Integer Multipliers	2
Floating Point ALU	4
Floating Point multipliers	4
Memory ports	4
Branch predictor	256Kbits TAGE
Memory dependence predictor	Store sets
Minimum misprediction penalty	20 cycles

Table 1: Characteristics of the simulated architecture

Benchmark	Number of Simpoints	IPC (BASE)	Miss rate (MPKI) (BASE)	Miss rate (MPKI) (with SBL prediction)
Floating Point Benchmarks				
bwaves	20	5.01	0.01	0.01
gromacs	19	4.84	1.14	1.06
lbm	13	0.97	0.03	0.03
leslie3d	22	2.56	0.01	0.01
milc	19	1.39	0.001	0.001
namd	20	4.18	1.56	1.26
povray	19	3.95	0.35	0.35
Integer Benchmarks				
astar	18	2.96	11.14	9.26
bzip2	17	4.26	2.91	2.67
gcc	19	3.65	0.78	0.78
go	17	2.60	6.93	6.60
h264	18	4.87	0.60	0.54
hmmer	9	2.57	9.05	7.8
mcf	24	0.60	8.32	6.98
omnetpp	13	1.80	2.56	2.51
perl	8	2.89	0.33	0.32
quantum	20	1.74	0.05	0.04
sjeng	26	2.67	4.01	4.01

Table 2: Characteristics of the benchmarks set

among the total execution. For each, the results shown are the weighted mean of the set results. Table 2 shows the number of simpoints taken for each benchmark.

8.3   Benchmark misprediction rates

Table 2 also lists the branch misprediction rates for each of the used benchmarks. As shown in Table 2, some benchmarks have a really low misprediction rate. On these benchmarks, it can be expected that a mechanism exploiting control independence will not increase performance. These benchmarks are *bwaves*, *lbm*, *leslie3d*, *milc*, *povray*, *gcc*, *h264*, *perl* and *quantum*. For the class of applications that encounters very small misprediction rate dynamic activation / deactivation of SYRANT could be considered to optimize energy consumption. Such a mechanism is out of the scope of this paper and will be considered in future work.

On the other hand, the remaining benchmarks exhibit a significant miss rate, especially *astar*, *go*, *hmmer* and *mcf*.

8.4   Reconvergence detection

Detecting the reconvergence of the mispredicted branches is a sine-qua-non condition for achieving some performance gain. The ABL/SBL mechanism

is able to detect most of the reconvergence cases on the benchmarks exhibiting significant misprediction rates (Figure 9).

We fail to detect reconvergence when misprediction is detected before reconvergence branch is fetched.
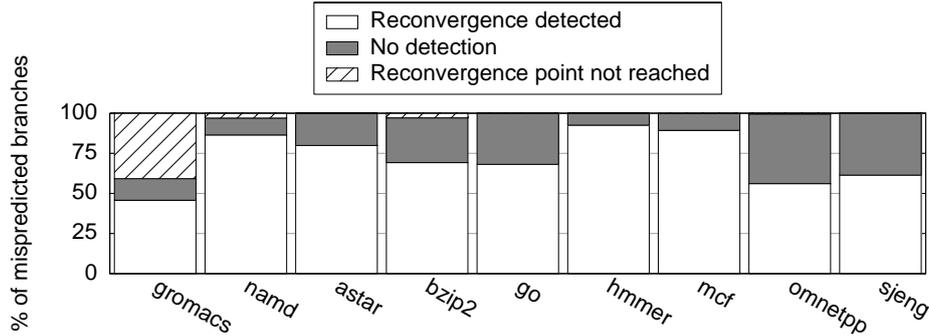

Figure 9: Breakdown of the mispredicted branches: reconvergence is detected for some of them.
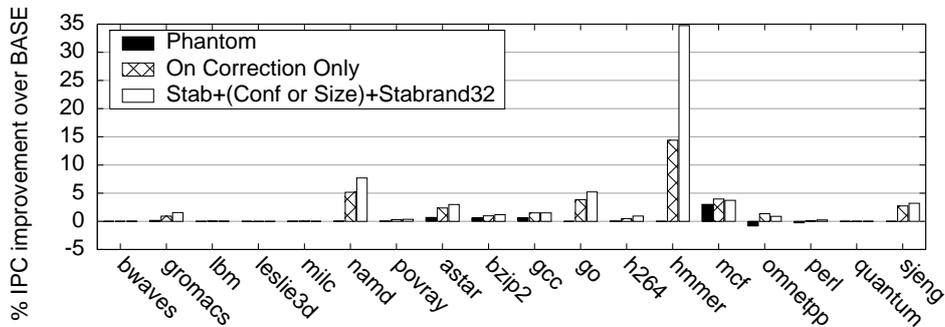
8.5 Gap insertion filters

Figure 10: IPC improvement with SYRANT over BASE with different gap insertion filtering mechanisms.

Figure 10 illustrates our experiments assuming the very aggressive 8-way issue configuration with 1024 ROB entries, 1024 integer registers, 1024 floating point registers and 512 Load Store Queue entries. Performances are illustrated as speed-up over the base configuration without SYRANT.

The *Phantom* configuration does not use SYRANT, but continues execution on the wrong path till resources are claimed back by the main execution path (see Section 5.3). Phantom execution, by itself is beneficial

for some benchmarks as it leads to often prefetch data for the correct path. This phenomenon is visible on *mcf* and *omnetpp*. For a few applications, phantom execution results in marginal wasting of memory bandwidth and some cache pollution and therefore results in a small performance loss.

With *On Correction Only* filter, gaps are inserted only if the correct path was the less demanding path is the correct path. This strategy eliminates a large amount of useless gaps since most branches are correctly predicted. Therefore it results in performance gains on nearly all the benchmarks. However the performance gains are relatively minor except for *hmmer*. Statistically only half of the control independence situations are candidate for exploitation.

Inserting gaps at prediction time can be useful for performance but their insertion has to be filtered by some mechanism.

Several gap insertion filtering policy have been proposed in Section 7. Due to space limitation, we only report the best combination we explored, *Stab+(Conf or Size)+Stabrand32*).

The *Stab+(Conf or Size)+Stabrand32* filter systematically inserts gap on branch correction but uses several filters for the insertion at prediction time:

- *Stab* (stand for *Stability*): this filter inserts a gap only if the reconvergence has been detected several times and with the same gap values. Using *Stability* filter with various thresholds was tested and using thresholds higher or equal than 2 were found to results in approximately equivalent performances.

- *Conf or Size* (stand for *Confidence or Size*): when using the *Confidence* filter, a gap is inserted on low confidence branches and if the predicted path is the less demanding path. The *Size* filter inserts gap only if its size is less than a threshold. Thus with the *Conf or Size* filter, a gap is only inserted on low confidence branches or if its size is less than a certain threshold.

- *Stabrand32* (stand for *Stability randomly decreased with a $\frac{1}{32}$ probability*: it is not a filter but an additional operation on the *Stability* counters. Each time a gap is inserted at decode time, the *Stability* counter associated to the branch is decremented with a $\frac{1}{32}$ probability. Thus, if creating gap at decode time is not useful anymore, the *Stability* counter will go under the threshold after some time, reducing the number of unnecessary gaps.

With this filter, a gap is inserted at decode time if it is stable (*Stability* filter) **and** (if the branch is low confidence *or* the insertion gap size is under some threshold). In the illustrated experiments, the threshold is 4 ROB entries, 4 registers and 2 LSQ entries. In practice, on our benchmark set, *Stab+(Conf or Size)+Stabrand32* performs at the same performance range as the best of the *Confidence, Stability* and *Stab+Conf* filters.

This configuration performs better than the *On Correction Only* config- uration for nearly all the benchmarks, except for *mcf* and *omnetpp* where the number of additional gaps inserted at decode time is still a little bit too large.

Compared with the BASE configuration the *Stab+(Conf or Size)+Stabrand32* filter leads to performance gain for all of our benchmarks.

As it could be expected, there is a strong correlation between the TAGE misprediction rate and the ability of SYRANT to increase the performance.

8.6 SBL prediction
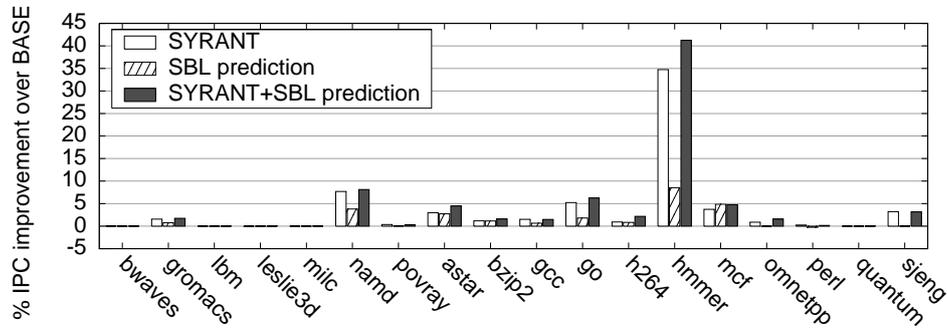
Figure 11: IPC improvement of SYRANT, SBL prediction and SYRANT+SBL Prediction over BASE.

As pointed out in Section 6, the ABL/SBL hardware mechanism can be used to improve branch prediction per itself by exploiting the executed branches after the reconvergence branch.

Table 2 illustrates the misprediction accuracy improvement obtained through using SBL prediction on top of the TAGE prediction. This accu- racy improvement is significant for a few benchmarks and results in some performance improvement (Figure 11). For instance *hmmer*, *mcf*, *astar* and *namd* have very significantly reduced mis prediction rate and experience a visible performance improvement. On the other hand, *sjeng* and *omnetpp* do not benefit from the SBL prediction, but do not lose any prediction ac- curacy.

Figure 11 also illutrates the combination of the *Stab+(Conf or Size)+Stabrand32* SYRANT filter with SBL prediction (column *SYRANT+SBL prediction*). For some benchmarks (*astar*, *go*, *h264* and *hmmer*), benefits of SYRANT and SBL prediction appear as nearly cumulative while on a few others, (*gromacs*, *namd*, *bzip2* and *mcf*) the performance gains do not cumulate.

8.7 Moderate issue width

We run experiments using SYRANT on a 4-way superscalar processor using half of the execution resources of the aggressive configuration. On Figure 12, we only illustrate the results for the *SYRANT+SBL prediction* configuration (called only SYRANT). The same benchmarks as for the aggressive configuration are exhibiting speed-ups.
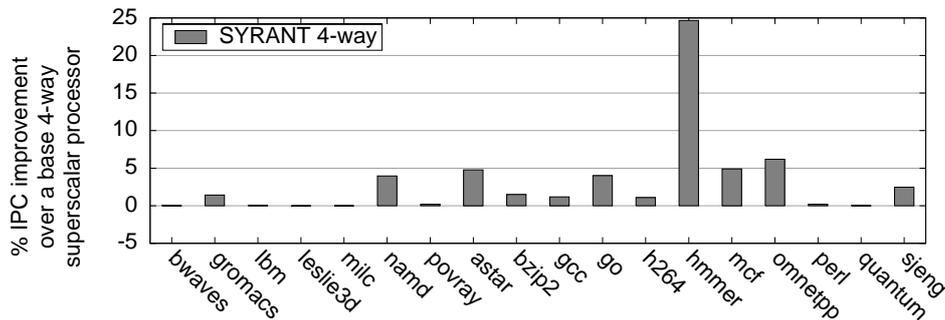

Figure 12: IPC improvement with SYRANT over BASE on a 4-way superscalar configuration

8.8 Varying the instruction window size

Figure 13 illustrates simulations using instruction windows of respectively 256, 512 and 1024 instructions using the *SYRANT+SBL prediction*. At the exception of *Quantum* the performance benefit from using SYRANT increases with the size of the instruction window. Moreover, a window of 512 instructions seems to be sufficient to observe significant results using SYRANT. More aggressive filters would be required for small instruction windows.

9 Conclusion

For achieving ultimate performance on sequential codes, exploiting control flow reconvergence is appealing since it allows to reuse already executed instructions. However, the prior proposals relied on complex hardware mechanisms [8, 5, 12, 2, 20] necessitating complex modification in the execution pipeline of superscalar processor

This hardware complexity may prevent processor designers to implement control flow reconvergence. We have elaborated a new proposal called SYRANT, SYmmetric Resource Allocation on Not-taken and Taken paths. SYRANT does not imply major modifications of the execution core on the
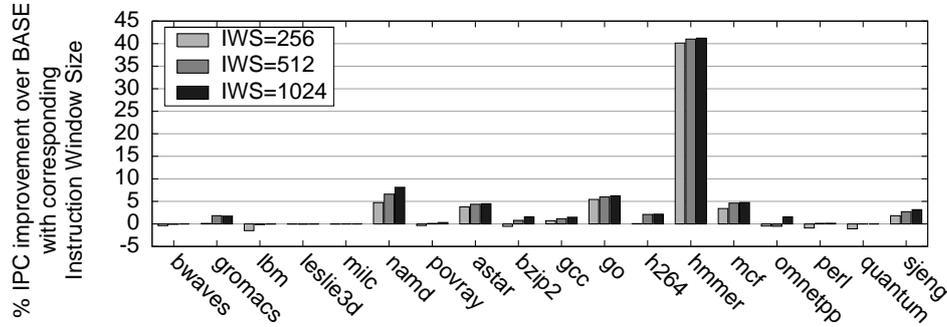
Figure 13: Variations of the results with respect to the Instruction Widow Size (IWS).

superscalar execution core. SYRANT is designed to allocate the same resources of the out-of-order execution core to the same instructions after the reconvergence point on the taken and the not-taken paths. Thus complex data movements are no longer needed to exploit control independence. Re-associating the result of a Control Independent instruction I already executed on the wrong path with the new instance of the same instruction I on the correct path is trivial.

The symmetric resource allocation is enforced through gap insertions in the out-of-order execution structures (register free list, ROB, LSQ). This allows to ensure that the same resources are used on both paths. We have presented simple mechanisms to detect the reconvergence and to enforce data dependencies while preserving already executed control independent data independent instructions.

The simulation presented in this paper indicates that provided a correct filtering of the gap insertion, SYRANT is able to bring a small speed-up on most of the applications exhibiting significant branch misprediction ratios.

In the process of defining SYRANT, we had to invent a new and effective mechanism for detecting reconvergence points. The definition of our ABL/SBL mechanism appears as an important contribution for improving superscalar processor performance with very limited intrusion in the processor structure. ABL/SBL allows to monitor branch reconvergence and to keep the results of executed branches on the wrong path. The addition to ABL/SBL to a conventional pipeline is not intrusive, but would allow to significantly improve branch prediction accuracy on some hard-to-predict benchmarks.

While SYRANT preliminary results might not justify the hardware implementation of SYRANT, we intend to pursue the research using the SYRANT framework in several directions to improve ultimate sequential performance. Continuing the exploration of new insertion gap filters seem necessary, in

particular for medium size instruction windows. The speculative execution of the branches on the wrong path could be exploited to enhance branch prediction after branch misprediction recovery. SYRANT also appears as a possible framework to implement dual-path execution at a reasonable cost.

References

[1] SPEC CPU2006. *http://www.spec.org/cpu2006/*.

[2] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. Akkary. Transparent control independence (tci). In *ISCA*, pages 448–459, 2007.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[5] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO*, pages 4–15, 2001.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[7] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *MICRO*, pages 129–140, 2004.

[8] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. In *HPCA*, pages 254–264, 2004.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0 : Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, vol. 7, September 2005.

[10] J. H. Hesson, J. LeBlanc, and S. J. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load/store instructions in a processor capable of dispatching, issuing and executing multiple instructions in a single processor cycle. US Patent 5,615,350, March 1997.

[11] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[12] A. D. Hilton and A. Roth. Ginger: control independence using tag rewriting. In *ISCA*, pages 436–447, 2007.

[13] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 142–152, Washington, DC, USA, 1996. IEEE Computer Society.

[14] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 235–245, Washington, DC, USA, 1997. IEEE Computer Society.

[15] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *MICRO*, pages 138–148, 1997.

[16] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in super-scalar processors. In *HPCA*, pages 115–124, 1999.

[17] A. Seznec. Storage free confidence estimation for the tage branch predictor. Research Report RR-7331, INRIA, August 2010.

[18] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism*, February 2006.

[19] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, volume 82, pages 1609–1624, 1995.

[20] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *ISCA*, pages 194–205, 1997.