Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading

Sébastien Hily, André Seznec

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading

Sébastien Hily, André Seznec

N° 3391

March 1998

THÈME 1

ISSN 0249-6399

RENNES

Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading

Sébastien Hily, André Seznec

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n3391 — March 1998 — 16 pages

Abstract: To achieve a high performance on a single process, superscalar processors now rely on very complex out-of-order execution. Using more and more speculative execution (e.g. value prediction) will be needed for further improvements.

On the other hand, most operating systems now offer time-shared multiprocess environments. For the moment most of the time is spent in a single thread, but this should change as the computer will perform more and more independent tasks. Moreover, desktop applications tend to be multithreaded. Most of the users should then be more concerned with the performance throughput on the workload than with the performance of the processor on a single process. Simultaneous multithreading (SMT) is a promising approach to deliver high throughput from superscalar pipelines. In this paper, we show that when executing 4 threads on an SMT processor, out-of-order execution induces small performance benefits over in-order execution. Then, for application domains where performance throughput is more important than ultimate performance on a single application, SMT combined with in-order execution may be a more cost-effective alternative than ultimate aggressive out-of-order superscalar processors or out-of-order execution SMT.

Key-words: Simultaneous Multithreading, Superscalar microprocessors, out-of-order execution.

(Résumé : tsvp)

This work done while Sébastien Hily was with IRISA. He was partially supported by a grant from regional council of Brittany. Sébastien Hily is now with Intel Corporation.

shily@ichips.intel.com, seznec@irisa.fr

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

A propos de l'exécution dans le désordre et des processeurs multiflots simultanés

Résumé : Dans cet article, nous montrons que l'association exécution dans l'ordre et multiflot simultané permet d'atteindre des niveaux de performance du même ordre que l'association exécution dans le désordre et multiflot simultané.

Mots-clé : Multiflot simultané, exécution dans le désordre

Abstract

To achieve a high performance on a single process, superscalar processors now rely on very complex out-of-order execution. Using more and more speculative execution (e.g. value prediction) will be needed for further improvements.

On the other hand, most operating systems now offer time-shared multiprocess environments. For the moment most of the time is spent in a single thread, but this should change as the computer will perform more and more independent tasks. Moreover, desktop applications tend to be multithreaded. Most of the users should then be more concerned with the performance throughput on the workload than with the performance of the processor on a single process. Simultaneous multithreading (SMT) is a promising approach to deliver high throughput from superscalar pipelines. In this paper, we show that when executing 4 threads on an SMT processor, out-of-order execution induces small performance benefits over in-order execution. Then, for application domains where performance throughput is more important than ultimate performance on a single application, SMT combined with in-order execution may be a more cost-effective alternative than ultimate aggressive out-of-order superscalar processors or out-of-order execution SMT.

keywords: Simultaneous Multithreading, Superscalar microprocessors, out-of-order execution.

1 Introduction

During the last few years, the design trend for general purpose microprocessors has been to track the ultimate single process performance. Current microprocessor architectures are relying on very aggressive hardware mechanisms to execute instructions out-of-order. Such mechanisms enhance an application execution by looking for independent instructions to issue. The ability of a processor to free itself from the sequential model imposed by the applications has thus become a key feature determining the level of performance. Further increase of the performance of singlethreaded superscalar architectures will depend on even more aggressive techniques, such as load and value predictions [2, 18, 17, 23], multiple basic block fetching [22] and possibly other forms of speculative execution. This will further increase the design complexity and may lead to longer and longer design and test cycle. Moreover, beside the high level of complexity reached by the implementation, the effective performance observed on general-purpose processors remains relatively low compared to their peak performance. This is mainly due to limited instruction-level-parallelism (ILP) offered by most applications [4].

On the other hand, most of the desktop computers or workstations run multitasked operating systems (e.g.: Windows95, WindowsNT, Unix). The operating system keeps switching the execution between all the active tasks, usually on a time-sharing basis. For an OS like Windows NT or Solaris, several tasks can even be executed simultaneously if several processors are available. While previously multithreaded workloads were limited to a scientific environment or to databases, now, a conventional PC running Windows95 has around 20 threads created before the user launches any application. Most of these threads are created by the 32-bit kernel, the task manager, the 16-bit applications manager and the Explorer user interface (see WinTop under Windows95 and TaskManager under WindowsNT) . The number of running tasks then quickly grows as the user executes new applications. Even if for the moment few tasks need and have high activity at the same time, this will change as the user will be willing to play a game while downloading a file or to print data and process a worksheet while writing a report. Moreover, desktop applications tend to be multithreaded [16]. Applications such as Powerpoint, Excel, Winword or Netscape create several threads. Most of these threads are dedicated to real-time activity such as spell-checking or graphics drawing while the user is working. Web browsers such as Netscape are especially prolific in creating threads and while downloading pages, the task manager shows often several threads working in parallel. Today, very few threads have intensive CPU activity, but desktop have more and more complex tasks running in background. While iconified, Netscape keeps performing CPU-intensive tasks with certain web pages. Moreover, a lot of applications, such as graphical applications or data processing could exhibit more very cpu-intensive threads. The absence of parallel support has prevented their algorithms to be parallelized but this can be easily done. In the near future more and more deskstop applications should then be able to take advantage of a hardware support for parallel execution.

Such a hardware support on a uniprocessor may be provided by Simultaneous Multithreading (SMT). SMT is a new concept of processor for achieving high performance throughput [24]. It relies on the avai-

lability of independent instructions, from several simultaneously active threads, to enhance ILP. The active threads can share all the processor resources, and thus, even when there is only one active thread, the performance can be high. The sharing gives also to an SMT architecture a significant performance advantage compared to a transistor budget equivalent on-chip multiprocessor [25]. Two alternatives exist for the design of the superscalar pipeline architecture: in-order execution and out-of-order execution. In-order execution has been rejected in most of the recent high end microprocessors because of poor performance compared to out-of-order execution. Nevertheless, the design complexity of out-of-order execution is very high, and recent examples have shown that such designs are often delayed.

In this article, we investigate the respective performance of both types of implementation in an SMT processor. Our study shows that, while ultimate single process performance requires out-of-order execution, high-throughput microprocessors may rely on in-order execution and SMT. When executing 4 threads, in-order execution is shown to be nearly as effective as out-of-order execution. We then look at the opportunity of using static instruction reordering to enhance the performance when executing several threads simultaneously. We show that the impact of this reordering is quite low for in-order execution as well as for out-of-order execution.

The remainder of this paper is organized as follows. After a short discussion of instruction pipeline in Section 2, we describe in detail in Section 3 the simulated architecture as well as the benchmarks used for the tests. In Section 4, we compare the respective performance of the two types of pipelines for optimized codes. Section 5 investigates the impact of instruction reordering on SMT. Finally, Section 6 presents a summary of this study.

2 Instruction pipeline

The core of modern microprocessors is constituted of an instruction pipeline [14]. The execution of instructions is divided into different successive phases. Generally these phases are, the instruction fetch from memory, a decoding phase, the execution of the operations, possibly an access to memory and finally the update of registers. This traditional decomposition has known an evolution to adapt itself to new architectural constraints and the increase of the clock frequencies. Processors have also evolved in two major categories, the first based on in-order execution of the instructions, the second allowing out-of-order execution.

Out-of-order execution improves performance by maximizing the number of instructions issued on every cycle [13]. Nevertheless, the implementation implies a very high cost. When executing instructions in-order, pipeline hazards prevent issuing further instructions, yet a lighter control authorizes higher clock frequencies. These two design philosophies have been at the origin of the classification of the processors into two categories: the "Brainiacs" running after a greater number of instructions to be executed in each cycle and the "Speed Demons" looking for higher clock frequencies [8].

Since several years, the performances showed by the "Brainiacs" and the "Speed Demons" cover a large but relatively equal spectrum. The Alpha processors (in-order model) have nevertheless kept the lead in the matter of peak performances. However, as the integration capacity and clock frequencies increase, complex designs appear increasingly attractive. Out-of-order execution allows singlethreaded superscalar processors to reach a significantly higher degree of instruction level parallelism (ILP) than in-order execution. Most of the high level microprocessor designers have thus rallied around the "Brainiacs" philosophy. The new DEC processor, the Alpha 21264, or the latest Hewlett-Packard processor, the PA8000, execute instructions out-of-order. Nevertheless, the race for complexity requires huge development teams (200-400 persons) and leads to the extension of the development time (3 to 5 years).

There has been a sustained research effort put on singlethreaded superscalar architectures and new techniques are emerging to further increase the performances of microprocessors : load and value predictions [2, 18, 17, 23], multiple branch prediction or trace cache [22] ... However, this effort focuses on increasingly complex mechanisms with more and more hardware for control and less for true execution.Moreover, the effective performance observed remains relatively low compared to the peak performance. This phenomenom is mainly due to the limited ILP offered by the applications [4].

Achieving performance on a processor may be obtained by using alternative architectures and especially by relying on more parallelism. Being able to execute several processes simultaneously could offer a significant gain in performance. Simultaneous multithreading (SMT) is a new architectural concept offering high performance through a better hardware utilization [25, 24]. In an SMT processor several threads can be executed simultaneously. Thus, many

independent instructions are available for execution on each cycle. When one thread has no fireable instructions, other active threads are likely to have instructions ready to be sent to the functional units. Issuing opportunities offered by the threads should preserve high ILP, even when instruction selection from one thread is limited to the sequential order of the program. Compared to an on-chip multiprocessor, in an SMT microprocessor the active threads share all the hardware resources, in particular, instruction and data caches may be shared. This gives SMT a significant performance advantage [25], especially when the number of threads decreases as when only one thread is available, it can use the complete processor to execute. An SMT single-chip processor can be built adapting an existing superscalar architecture, thus limiting the design and test cycle. In [24], Tullsen et al. chose an out-of-order pipeline in their SMT architecture model. On the other hand, the model proposed by Goossens and Vu [7] aiming at a short cycle is limited to in-order execution. To our knowledge, no studies comparing performances of in-order and out-of-order executions have been undertaken when several threads are available simultaneously. In the remainder of this article, we study the respective performances of different pipelines in a simultaneously multithreaded environment as well as the impact of static instruction reordering on the performance of synchronous multithreaded processors.

3 Experimental framework

A trace-driven simulation model has been used in this study. Our simulator is based on the Sparc V7 instruction set [20] and has been build from Spy, the tracing application for Sparcstation written by Gordon Irlam [12].

3.1 Simulated architectures

For the simulations, a 7-stage integer pipeline has been used (ref. Figure 1). First, instructions are fetched from the memory into an instruction buffer. There is one buffer per thread. The instructions available in the buffers can then be fed into the pipeline according to the priorities given to the active threads. The highest priority is granted to the thread having the least instructions in the static part of the pipeline[24]. The lower this number, the higher the priority. Intuitively, a low number of instructions present for a thread gives a good opportunity of executing subsequent fetched instructions.

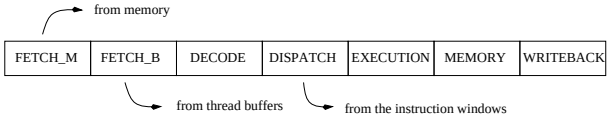

Figure 1: Integer pipeline

Each instruction buffer has a capacity of two cache blocks (i.e 16 instructions). On every cycle, up to 8 instructions are read from the buffers. Following the computed priorities, one fireable instruction is selected for each active thread. If all the issue slots are not filled and there are instructions left in the buffers, new selections are carried out. Thus, if only one thread is available, it should be able to fill all slots in the execution pipeline (i.e. 8 instructions may be selected from the same thread). When the number of instructions remaining in a buffer is smaller than the block size, a prefetch can be made. Figure 2 illustrates the instruction selection mechanism for four threads and a 8-wide instruction pipeline. Such a powerful (but complex) fetch mechanism should not be unrealistic. Other studies, such as in [5], have shown that complex instruction fetch mechanisms are feasible.

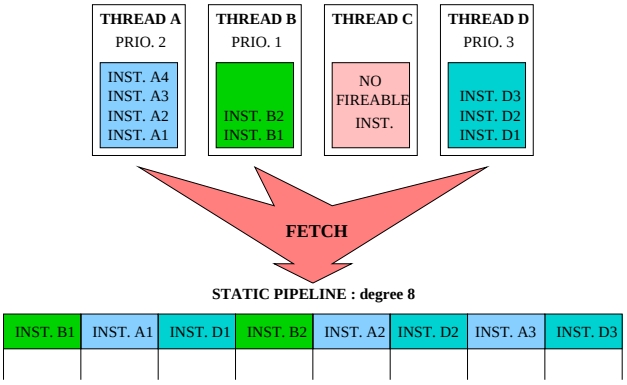

Figure 2: Instruction selection

In Figure 3, we illustrate the two simulated architectures, the first, featuring in-order execution, the second, out-of-order execution.

In-order execution For the in-order execution (Figure 3A), the instructions are placed in instruction queues after the decoding phase. One 32-entry instruction queue is associated with each thread. On every cycle, queues are scanned to find instructions to issue to the functional units. The same priority rule as described for instruction fetch is used for instruction selection in the queues. Instructions from a queue are issued in-order until there is a resource

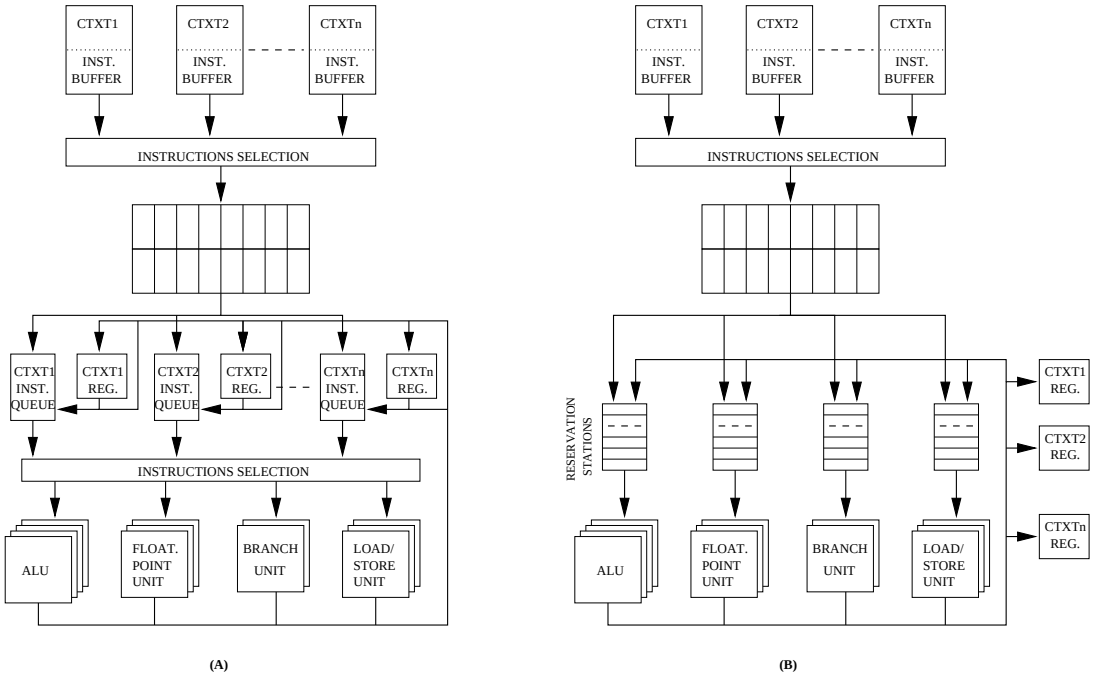

Figure 3: Architectures featuring in-order execution (A) and out-of-order execution (B)

conflict, a data hazard or a branch. The following queue in the priority order is then selected. When all the queues have been searched, although functional units are still available, a new round of selections is attempted on queues whose selections were stopped on a branch. This process goes on until all the queues are empty or blocked because of data hazards or resource conflicts or until all the functional units are busy. Each thread has access to 32 integer registers and 32 floating-point registers.

Out-of-order execution For out-of-order execution (Figure 3B), after the decoding phase, the instructions are placed in reservation stations [9]. The instructions wait there for the availability of their operands and functional units. On every cycle, any fireable instruction can be executed irrespective of the program order. Out-of-order execution relies on register renaming with 64 physical integer registers and 64 physical floating-point registers per thread [13]. Each reservation station has 16 entries.

Common features These two architectures can rely on a shared branch prediction mechanism and a shared memory hierarchy whose characteristics are summarized in Table 1. In [10], Hily and Seznec showed that when using a gshare branch predictor, all prediction structures (except the return address stack) might be shared without any significant loss of accuracy (size of prediction structures should be set up according to the number of threads). Prediction tables used in the presented simulations are then shared (except return stacks). The relatively small structures that we used is explained by the lack of kernel instructions in our traces. For real implementations, larger structures would be needed [6]. Two branch predictions on two different threads can be made on each cycle. The two levels of caches are non-blocking, meaning that several misses on the same level can be pending at the same time [15]. The execution phase of an integer instruction takes one cycle. Floating point instruction execution phase takes three cycles. The execution phase can be followed by a memory access which is done in one cycle when hitting in the cache. The write-back phase assumes that there are no conflicts on the result buses. For the out-of-order architecture, we do not assume a precise interrupt mechanism, therefore no in-order retirement was simulated.

Both architectures have 4 ALU, 3 load/store units, 2 branch units and 3 undifferentiated floating-point units. All functional units are fully pipelined.

False branches A specific difficulty on trace-driven simulations is to correctly model the impact of false branches on the execution, i.e. the code executed following a mispredicted branch instruction. In our simulations, we have implemented the following

Memory	latency	30 cycles
	throughput	16 bytes every 4 cycles
	bus size	128 bits
L2 Cache	Size	1 Mbyte
	block	64 bytes
	associativity	4
	latency	5 cycles
	throughput	16 bytes every 2 cycles
L1 Caches	Size	32 Kbyte
	block	32 bytes
	associativity	2
	latency	1 cycle
	Nb. of instruction-cache access	2/cycle
	Nb. of data-cache access	3/cycle
	bus size	256 bits
Prediction	scheme	gshare
	BTB	512 entries
	PHT	4096 entries
	stack	32 entries/thread

Table 1: General characteristics

mechanism to deal with this difficulty: Each time the prediction is wrong, a false branch is created. This false branch consists of fake instructions reflecting the statistical distribution of the instruction previously executed by the faulting thread. Our simulations take into account that, these instructions fill slots in the static pipeline and instruction windows. Therefore, our model does not favor in-order or out-of-order execution in the static stages of the pipeline.

However, our model does not reflect the real behavior of the dynamic stages of the pipeline: instructions are not executed then they do not compete for accessing the functional units and do not have any impact on caches. This clearly favor out-of-order execution. First, for out-of-order execution, instructions on the wrong-path might lead to (unneccessary) data cache misses which may waste L2-cache and memory bandwidth, we did not simulate this phenomenom at all. On the other hand, for in-order execution, such a situation can not occur since a branch is resolved before (or at the same time) any memory access on the wrong path is presented to the data cache. Second, on a out-of-order execution, many more speculative instructions than on an in-order execution processor are presented to the functional units. Those speculative instructions from the wrong path may delay instructions from the rigth path of an other thread.

3.2 Benchmark selection

Our benchmark set is composed of Spec95 applications described in Table 2 (the names of the programs normally begin with a number that we omitted for a better readability).

All the benchmarks were compiled using gcc 2.7.2 or f77 SC3.0.1 compilers with the standard -O2 optimization, under SunOS 4.1.4 Operating System and on a Sparcstation 20. The standard inputs recommended for Spec95 benchmarks were used.

In the remainder of this article, the names of the composite workloads appearing in the illustrations are the concatenation of the first two letters of the executed applications (except for *apsi* which is *ai*). For example, *pevo* corresponds to the simultaneous execution of *perl* and *vortex*.

3.3 Simulation protocol

We have undertaken simulations for various workloads constituted of 1, 2 or 4 distinct applications of Spec95. Our study is limited to a multiprogrammed environment. Workload applications correspond to the simultaneously executed threads. It was shown in [11] that with a conventional memory hierarchy, more than 4 threads would not be cost-effective, even when using an ultimately aggressive out-of-order execution. For each of the simulations, the memory hierarchy is initialized by executing 10 millions of instructions per active thread in order to avoid the impact of cold

type	program name	description
integer	go	Plays the game Go against itself
	m88ksim	Simulates the Motorola 88100 processor running Dhrystone and a memory test program
	compress	Compresses large text files (about 16MB) using adaptive Limpel-Ziv coding
	lisp	Lisp interpreter
	ijpeg	Performs jpeg image compression with various parameters
	perl	Performs text and numeric manipulations (anagrams/prime number factoring)
	vortex	Builds and manipulates three interrelated databases
floating-point	tomcatv	Generation of a two-dimensional boundary-fitted coordinate system around general geometric domains
	swim	Solves shallow water equations using finite difference approximations
	su2cor	Masses of elementary particles are computed in the Quark-Gluon theory
	hydro2d	Hydrodynamical Navier Stokes equations are used to compute galactic jets
	mgrid	Calculation of a 3D potential field
	applu	Solves matrix system with pivoting
	turb3d	Simulates turbulence in a cubic area
	apsi	Calculates statistics on temperature and polluants in a grid
	fpppp	Performs multi-electron derivatives
	wave5	Solves Maxwell's equations on a cartesian mesh

Table 2: Spec95 programs

start misses. We then simulate up to 20 millions of instructions per thread (the simulation stops as soon as one of the threads has executed 20 millions of instructions). Notice that in no way our study tries to exactly characterize SMT behavior for these particular applications, but that our goal is to compare, for given traces, the behavior of different architectures.

Throughout this study, performance will be reported in Instructions Per Cycle (IPC).

4 In-order versus out-of-order executions

Our aim is to evaluate the respective performance of architectures issuing instructions in-order or out-of-order when several threads are being executed simultaneously. Preliminary simulations have been made with optimized applications running alone to characterize their behavior (peak version of Spec95 applications).

4.1 Singlethread performance

Figure 4 illustrates the performance obtained for the singlethread simulation of the applications constituting our benchmark. Dark bars refer to in-order execution, light bars to out-of-order execution. The ratio of in-order performance versus out-of-order performance is illustrated by the dots. Dots are connected to help the reading, but no particular significance should be attached to this connection.

Despite our aggressive design, the performances exhibited by in-order execution are relatively weak. The IPC varies between only 1 and 2 whereas the superscalar degree is 8 (with 12 functional units). Such a high number of functional units is not particularly useful for in-order execution. Out-of-order execution allows a significant performance gain. For 6 of the applications, the IPC varies between 3 and 4. For *fpppp*, the IPC is hardly above 1.5, which is mainly due to two reasons: the simulated code sequence shows a very low instruction cache hit ratio and execution is slowed by the numerous dependencies on the floating-

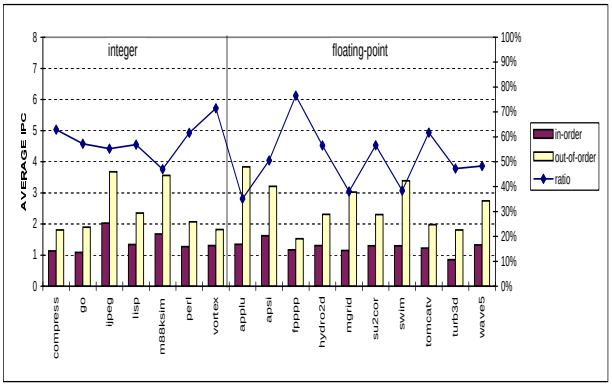

Figure 4: 1 thread, simulation of the pipeline, the branch prediction and the memory hierarchy

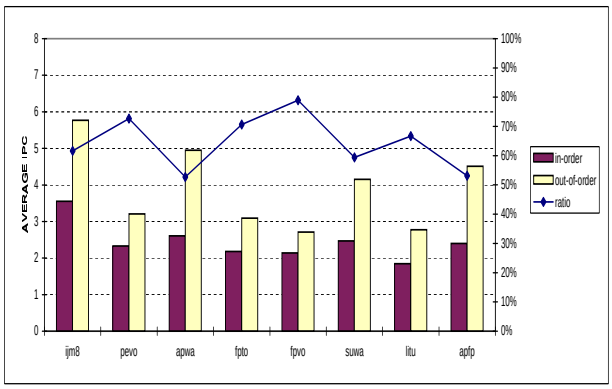

Figure 5: 2 threads, simulation of the pipeline, the branch prediction and the memory hierarchy

point instructions. For this particular application, the ratio of in-order to out-of-order performances is very high (77%). Depending on applications, these ratios vary between 35% and 77%. On average, ratios are however lower than 60%, which is quite low and explains why out-of-order execution has been chosen in most current high-end microprocessor designs. This performance gap would have been even larger if more aggressive instruction fetching (eg. trace cache [22]) had been considered for out-of-order execution.

4.2 Executing multiple threads

Workloads selection 17 applications have been tested for single application tests. Mixing four threads would lead to 2380 (C_{17}^4) different thread combinations. This would have consumed too many computing resources and would have lead to a huge volume of simulation results (hardly exploitable). Therefore we prefered to select a few combinations of benchmarks based on the behavior of stand-alone applications (Ref. table 3). In-order performance or in-order/out-of-order ratio were taken into consideration in this selection. For instance, in *ijm8*, we put together the best two integer applications. In *apfp*, we combined two applications, *fpppp* showing the best in-order/out-of-order ratio, *applu* exhibiting the worst ratio.

Global performance Figure 5 illustrates simulation results for 2 threads. A sharp increase of performance may be noticed. For in-order execution, performance varies between 1.8 and 3.5 instructions executed per cycle. A speedup varying between 1.64 and 1.95 is observed (when comparing the IPC corresponding to the sequential execution of the applications constituting the workloads to the IPC obtained with the SMT execution). The speedup of 1.95 is obtained for *apwa*, a workload mixing two of the floating-point applications showing the best performances with the in-order pipeline. When executing out-of-order, performance varies between 2.7 and 5.7 IPC. The speedup factor ranges from 1.4 to 1.76. Factor 1.76 is obtained for *apfp*, a workload mixing the best performing (*applu*) and the worst performing (*fpppp*) applications for the out-of-order pipeline.

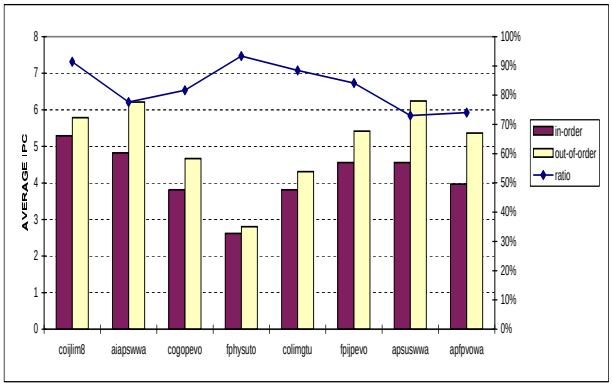

Figure 6: 4 threads, simulation of the pipeline, the branch prediction and the memory hierarchy

The increase of the IPC is even more substantial in the case of 4 threads running simultaneously (ref. Figure 6). For in-order execution, performance varies between 2.6 and 5.3 IPC, while for out-of-order, it varies between 2.8 and 6.2 IPC. The lowest performance is observed for *fphysuto*, i.e. 4 floating-point applications among those having the worst singlethread performance. The gain remains however important compared to the sequential execution of the 4 applications. For workloads made of applications performing well in singlethreaded execution, performance remains very high for multithreaded execution (*coij-*

2 threads	description	4 threads	description
apfp	best and worst ratios	coijlim8	good int. IPCs
apwa	good floating IPCs	cogopevo	bad int. IPCs
fpto	bad floating IPCs	aiapswwa	good flt. IPCs
fpvo	best ratios	fphysuto	bad flt. IPCs
ijm8	best integer IPCs	colimgtu	2 int.and 2 flt. with average IPCs
litu	average integer IPC, bad floating IPC	fpijpevo	good ratios
pevo	worst integer IPCs	apsuswwa	bad ratios
suwa	average ratios	apfpvowa	2 good ratios, 2 bad ratios

Table 3: Mixes of applications, for 2 and 4 threads

lim8, aiapswwa). For a floating-point workload such as *aiapswwa*, out-of-order execution results in an appreciable performance gain over in-order execution (the ratio is "only" 78%).

These results illustrate perfectly the constructive behavior of SMT. Effectively, none of our workloads observed a performance loss due to the simultaneous execution of 4 threads. We did not investigate more than 4 threads as it was shown in [11] that L2 and memory bandwidth would be saturated when trying to execute 6 or more threads and that supporting more than 4 independent threads simultaneously would not be effective.

Performance convergence The most noticeable point is that the gap between out-of-order and in-order executions decreases as the number of threads increases. This gap seems to decrease quite independently from the chosen workload. For 2 threads (respectively 4 threads), in-order execution allows to reach between 53% and 79% (resp. 73% and 93%) of the IPC observed for out-of-order execution. The large performance advantage offered by out-of-order execution on singlethreaded architecture shrinks when several threads are executed on an architecture featuring SMT. This can be explained by several converging reasons: in the in-order pipeline, the opportunity to find independent instructions to fire increases when several threads are executing in parallel. In the out-of-order pipeline, the speculative execution of instructions which will be later canceled consumes resources potentially useful for the execution of valid instructions from the other threads.

Real performance results should even be more close for in-order and out-of-order execution: on false branches, our simulation model ignores the impact of memory bandwidth wasting and contention to access the functional units.

4.3 Impact of memory hierarchy and branch prediction

In order to measure the respective impact of the memory hierarchy and the branch prediction on performance, we run three complementary sets of simulations assuming either perfect or effective models of caches and branch predictor.

Figure 7 presents a synthesis of the average performance for architectures featuring 1, 2 or 4 simultaneous threads and executing instructions either in-order or out-of-order. In the first graphic, only the pipeline is simulated; branch predictions and the first-level caches were assumed perfect. In the second graphic, pipeline and branch prediction are simulated but the accesses to caches remain ideal. In the third graphic, a real memory hierarchy is simulated with the pipeline, but the branch prediction is perfect. In the fourth graphic, the complete processor is simulated, i.e. the pipeline, the branch prediction and the memory hierarchy. Results for 1, 2 or 4 threads are not strictly comparable as the workloads do not correspond exactly to the same mix of applications. However, the performances are kept in their respective orders.

These four graphics show that the saturation of caches leading to conflict and capacity misses and the limited accuracy of branch prediction clearly impairs performance :

- Branch mispredictions, despite a quite accurate prediction mechanism, still results in a high loss of performance, especially in out-of-order execution: out-of-order execution favors the speculative execution of “less likely to be useful” instructions. The impact of a misprediction however weakens as the number of threads increases. First, due to the mix of instructions from several threads in the static pipeline, the other threads continue to use the execution units. Second, less advance is made on execution by the running threads and less instructions are execu-

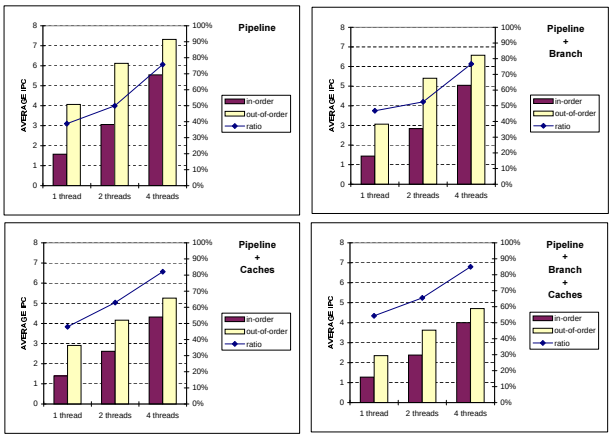

Figure 7: Average performance impact of memory hierarchy and branch prediction for 1, 2 or 4 threads with in-order or out-of-order execution

ted speculatively. The pipeline is then very rarely empty.

- The impact of the memory hierarchy is far more important. Unlike in branch mispredictions, the penalty of a cache miss will be potentially "paid" by all the threads because it consumes memory bandwidth and may delay any subsequent cache miss either from its thread or from another. For example for 4 threads, the average drop in performance induced by the cache misses is higher than 25%.

When the entire architecture is simulated, one can observe a significant degradation of the performances. The combined negative impact of the bad predictions and the cache misses is however very dependent on the type of the pipeline and on the number of threads. Thus, when instructions execute in-order, this negative impact increases as the number of threads grows; the loss in the IPC is 19% for one thread and reaches 28% for four threads. For out-of-order execution, this is the opposite with a decreasing impact when the number of threads increases; the loss in the IPC is 42% for one thread and falls to 36% for four threads, mainly due to a lower impact of mispredictions on the performance. Despite this, simultaneous multithreading offers a strong gain in performance, as the IPC reaches an average of 4 for in-order execution, and 4.7 for out-of-order execution. Moreover, when looking at the first graphic in Figure 7 (perfect branch prediction and ideal cache accesses), one can see that for 4 threads, poor benefits can be expected from using more aggressive speculative techniques (such as load value prediction, multiple branch prediction, etc).

4.4 Summary

In spite of a quite aggressive and in a certain way, optimistic architectural model, the average performance observed for a singlethreaded architecture is relatively low. If the main objective was single-thread performance, the weakness of in-order execution would justify the implementation of out-of-order mechanisms. With four threads, the average performance (measured in IPC) achieved by in-order execution reaches 85% of the one offered by out-of-order execution.

It should even be pointed out that our simulations did not take into account various phenomena that may even reduce this gap. First, the impact of memory accesses performed on a mispredicted branch. As less instructions are executed on the mispredicted branch in in-order execution, the loss of performance due to memory bandwidth wasting on the false branch would be higher on out-of-order execution than with in-order execution. Second, we did not simulate in-order retirement of instructions. In-order retirement would further limit out-of-order execution performance. Third, we assume the same minimum instruction pipeline length for in-order and out-of-order execution. A longer pipeline for out-of-order execution should further reduce the performance gap between out-of-order execution and in-order execution. For instance, when assuming the longer pipeline structure illustrated in Figure 8 for out-of-order execution (an extra cycle for renaming registers and a second one for the selection and issuing phase), our simulations showed that the performance gap between in-order and out-of-order execution falls to 9 % in average with 4 threads. Finaly, achieving the same

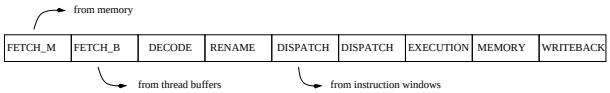

Figure 8: Integer instruction pipeline for out-of-order execution

processor cycle on the two designs is not straightforward.

Therefore, the effective performance for the two execution models should be very close for multithreaded workload (closer than just reported by our simulations). We believe that, even if there remains a small difference in performance between in-order and out-of-order executions, this will not be worth the added complexity of out-of-order execution. This extra complexity will lengthen the design and test of the processor.

5 Impact of instruction reordering

Applications are generally optimized to allow reasonably efficient execution on the different hardware platforms from a specific manufacturer. However, the performance could be enhanced if the applications were specifically optimized for a given architecture. Static reordering of instructions [3] could especially bring a better utilization of the instruction pipeline. The reordering should benefit far more an in-order pipeline than an out-of-order one. In this section, we investigate such optimization. It is shown that unlike for single applications, instruction rescheduling has a poor impact on the performance of in-order as well as out-of-order SMT execution.

Methodology We conducted a new set of simulations on SPEC95 codes which were reordered using SALTO [21]. SALTO allows to implement assembly-level code transformations using a description of the resources and latencies of an architecture. We implemented a simple instruction reordering algorithm. The rescheduling was limited to a single basic block, and no attempt was made to get more aggressive anticipation of instructions. Moreover, we did not rename registers. In this algorithm, an instruction is "executed" as late as possible, regarding the dependencies with the following instructions until the end of the basic block. Figures 9 and 10 illustrate averages IPCs for respectively 1 thread and 4 threads, in-order or out-of-order pipeline, with (*-reorder*) or without (*-order*) instruction reordering. For out-of-order execution, the simulated pipeline is the one represented in Figure 8. Results presented in this section can differ a little from those presented in the previous sections as we did not trace exactly the same sequences of code: as we did not get the library sources for this experiment, we chose sections of code where these libraries are not significantly used.

Singlethread execution and reordering As expected, for in-order execution of one thread, reordering exhibits a positive impact on the performance. This enhancement is particularly significant for *swim* (27%), *turb3d* (15%) and for *applu* (14%), three floating-point applications where long latencies (3 cycles) are encountered. However, the benefit of our reordering algorithm varies a lot with the applications and on average, the gain in performance is limited to 6%. With the out-of-order execution, the instruction reordering brings a maximum of 4% increase in performance, for *hydro2d* or *mgrid*, and the average increase is limited to 1%.

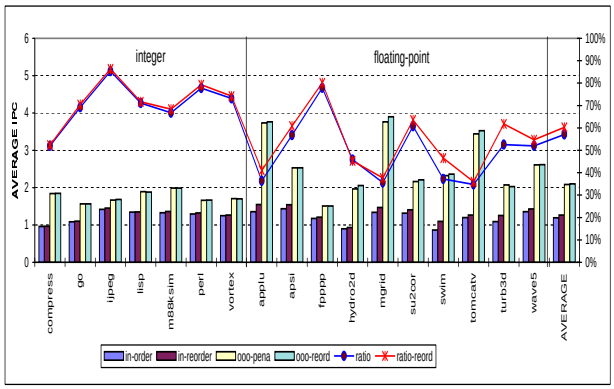

Figure 9: Average performances for 1 thread, in-order and out-of-order execution, with and without instruction reordering

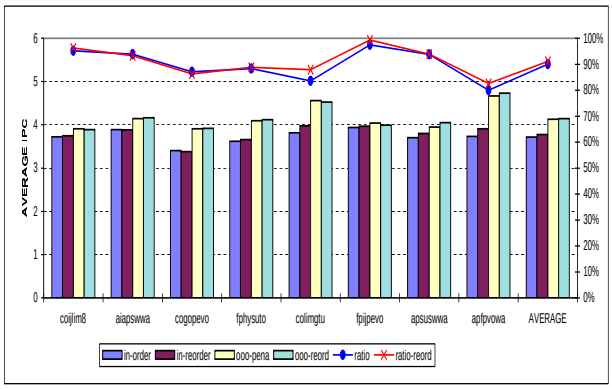

Figure 10: Average performances for 4 threads, in-order and out-of-order execution, with and without instruction reordering

Multithreading results With the simultaneous execution of several threads, the impact of reordering appears unsignificant. The difference in performance spans from 5% to -1% for in-order execution and from 3% to -1% for out-of-order execution. Moreover, for out-of-order execution, for 3 of the workloads the impact of reordering is negative. SMT execution does not appear to take advantage of instruction reordering, and this, even when the workload is composed of applications for which reordering can bring a significant increase of the IPCs. Thus, reordering offers no gain of performance for *aiapswwa* but allows to increase performance by 7%, 14%, 27% and 5 % on respectively *apsi*, *applu*, *swim* and *wave5*.

These results show clearly that optimizing application codes individually gives no guarantee of obtaining better performance under SMT. Interactions of the applications in the pipeline and at the resource level are not predictable and can lead to an increase of the conflicts. Our results concur here with the results obtained by Lo et al. [19] on software speculative execution. In this way, instruction reordering is a poor way to increase the performance of in-order SMT architectures. These results suggest that static optimizations of codes aimed at enhancing single thread ILP on in-order execution processors (function inlining, loop unrolling, software pipelining, ..) will not bring the same benefit for a multithreaded workload.

On the other hand, the poor 2% enhancement of performance for 4 threads for in-order execution can be viewed positively. It shows that an in-order SMT architecture does not need specific instruction rescheduling for achieving high performance, but achieves performance improvement without requiring specific recompilation, as have done out-of-order execution microprocessors on single-threaded workloads.

This reinforces the interest of SMT as a way of implementing low-cost high performance processors targeted at high throughput and not towards ultimate performance on a single application.

6 Summary

Today's general purpose high-performance microprocessors are based on a superscalar pipeline issuing instructions out-of-order. The dynamic extraction of instruction level parallelism permits the processor to overcome the numerous constraints faced by compilers when scheduling instructions. Such microprocessors are often used in a time-sharing multi-process environment. In such environments, time slices are successively allocated to different processes. Nevertheless, these processors achieve high performance on a single sequential application.

In an architecture featuring simultaneous multithreading, instructions from different threads share the pipeline at the same moment. Opportunities to find independent instructions for execution are therefore higher on an SMT processor, which naturally increases the effective ILP and decreases the resource wasting due to data and control hazards.

In this paper, we have investigated the respective performances offered by two types of architectures both featuring up to 4 threads simultaneously but one issuing instructions in-order and the other out-of-order.

First, our simulations have established that despite a quite aggressive out-of-order execution model, in-order execution may achieve roughly the same performance provided that 4 threads are available for execution. On a single thread, in-order execution suffers from a 46 % performance gap compared with out-of-order execution. However, the performance gap is only around 15% in average when the same number of pipeline stages and the same clock cycle are assumed for in-order and out-of-order execution. Moreover, many of our assumptions for simulations were optimistic when considering out-of-order execution: same minimum pipeline length for in-order and out-of-order execution, same clock speed, .. Then on real implementations, the small performance advantage of out-of-order execution will be lower, and may even not exist if higher clock speed is achieved with in-order execution.

We have also shown that imperfect branch prediction and memory hierarchy have a high impact on the performance. The use of multithreading decreases the impact of the branch prediction: less instructions are involved by a single misprediction. On the other hand, use of multithreading increases the stress on the memory hierarchy generating more cache misses and more memory traffic.

We further pointed out that even for in-order execution on SMT, the performance for multiple threads do not rely on a careful instruction scheduling. On the contrary, such instruction scheduling was shown to have a negligible impact (if any) on the performance of 4 threads.

For users concerned by ultimate performance on a single process, out-of-order execution will be needed; further performance improvements will rely on more and more complex mechanisms such as value and address prediction, multiple branch prediction, etc. On the other hand, if operating systems and applications continue to evolve to more and more thread and process level parallelism, then more and more users will be more concerned with throughput rather than with ultimate single process performance. Then for those users, our study has established that processors featuring SMT and in-order execution will be sufficient. Even when only 2 threads are available, an SMT architecture featuring in-order execution will outperform a singlethreaded superscalar architecture relying on out-of-order execution.

References

[1] *Alpha 21164 Microprocessor Hardware Reference Manual*, September 1994. preliminary edition.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture Support for Reducing Load Latency. In *28th Annual International Symposium on Microarchitecture*, pages 82–92, November 1995.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single Instruction Stream Parallelism is greater than two. In *18th Interernational Symposium on Computer Architecture*, pages 276–286, May 1991.

[5] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction Fetch Mechanisms for High Issue Rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[6] N. Gloy, C. Young, J. Bradley Chen, and M. D. Smith. An Analysis of Dynamic Branch Prediction Schemes on System Workloads. In *23rd Annual International Symposium on Computer Architecture*, pages 12–21, May 1996.

[7] B. Goossens and D. T. Vu. On-chip multiprocessing. In *Lecture Notes in Computer Science. Euro-Par'96*, volume 2, pages 789–796, August 1996.

[8] L. Gwennap. Speed Kills ? Not for RISC Processors. *Microprocessor Report*, page 3, March 1993.

[9] J. Hennessy and D. Patterson. *Computer Architecture : a quantitative approach.* Morgan Kaufmann publishers, 1990.

[10] S. Hily and A. Seznec. Branch Prediction and Simultaneous Multithreading. In *International Conference on Parallel Architecture and Compilation Techniques*, 1996.

[11] S. Hily and A. Seznec. Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading. In *Workshop on MultiThreaded Execution, Architecture and Compilation*, held in conjunction with HPCA-4, Colorado State Univ. Technical Report CS-98-102, January 1998.

[12] G. Irlam. Spa package. http://www.base.com/gordoni/spa.html, 1994.

[13] M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[14] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[15] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organisation. In *8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[16] D. Lee, P. Crowley, J.L. Baer, T. Anderson, B. Bershad. Execution Characteristics of Desktop Applications on Windows NT. To appear in *25th Annual International Symposium on Computer Architecture*, June 1998.

[17] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit Via Value Prediction. In *29th Annual International Symposium on Microarchitecture*, pages 226–237, December 1996.

[18] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[19] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning Compiler Optimizations for Simultaneous Multithreading. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[20] LSI LOGIC. *SPARC Architecture Manual (version 7).*

[21] E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, and F. Raimbault. SALTO: System for Assembly-Language Transformation and Optimization. Technical Report PI-1032, IRISA, June 1996.

[22] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.

[23] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. In *29th Annual International Symposium on Microarchitecture*, pages 238–247, December 1996.

[24] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23th Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[25] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximising On-Chip Parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399