Paired ROBs: A Cost-Effective Reorder Buffer Sharing Strategy for SMT Processors

R. Ubal, J. Sahuquillo, S. Petit, and P. López

Department of Computing Engineering (DISCA)
Universidad Politécnica de Valencia, Valencia, Spain
raurte@gap.upv.es, {jsahuqui,spetit,plopez}@disca.upv.es

Abstract. An important design issue of SMT processors is to find proper sharing strategies of resources among threads. This paper proposes a ROB sharing strategy, called *paired ROB*, that considers the fact that task parallelism is not always available to fully utilize resources of multithreaded processors. To this aim, an evaluation methodology is proposed and used for the experiments, which analyzes performance under different degrees of parallelism. Results show that paired ROBs are a cost-effective strategy that provides better performance than private ROBs for low task parallelism, whereas it incurs slight performance losses for high task parallelism.

1 Introduction

As a single software task is far from exploiting peak performance of current superscalar processors, simultaneous multithreading (SMT) [1] was proposed as a way of increasing the utilization of the processor functional units. One of the main research challenges of SMT processors is the design and optimization of proper resource allocation policies that decide how processor resources are assigned to each thread. Related proposals focus on the distribution of bandwidth resources, such as fetch or issue width [2][3], as well as storage resources, such as instruction queue, load-store queue or physical register file [4][5][6]. Most storage resources can be easily distributed in a dynamic way among threads, since any of their free entries can be allocated/deallocated at any time. However, the reorder buffer (ROB) manages instructions from threads in a FIFO order, which lowers the flexibility of dynamically assigning ROB entries to different threads.

There are two basic strategies to share the ROB among threads. The first one is referred in this paper to as *private ROB*, and consists in statically partitioning the ROB among threads. This approach is simple, has small hardware cost, and distributes ROB resources fairly among threads. The second strategy is referred to as *one-queue ROB*, and consists in a shared queue where instructions from different threads are inserted in local FIFO order, but in any global order. This approach has, depending on the implementation, several drawbacks in terms of hardware cost and performance: a) the dispatch, commit, and recover logic increases, b) instructions from different threads can block each other at commit, c) ROB *holes* (empty intermingled slots) appear when one single thread squashes its instructions after a mispeculation, and d) global performance can be reduced when a stalled thread occupies too many entries of the shared ROB.

All these arguments against the shared approach seem enough to discard this design. This conclusion has been already drawn in several research works [4][7][8]. However, all these studies evaluate ROB partition strategies in heavily loaded systems, that is, systems running a number of computation intensive software tasks equals to the number of hardware threads (e.g., a 4-benchmark mix on a 4-threaded 8-way processor). But it is not uncommon to find lightly loaded multithreaded systems, that is, systems using only a small portion of their hardware threads for computation intensive workloads.

Let us consider a typical user executing a single sequential (i.e., non parallel) computation intensive application on a 2-threaded SMT processor. This user does not experience any performance improvement at all with multithreading. Moreover, if the processor implements a private ROB partitioning, only half of the ROB entries are available to the computation intensive application, so a performance loss can be noticed for the same total ROB size. This effect has been evaluated by running the SPEC2000 benchmark suite on a single-threaded and a 2-threaded processor, both of them with the baseline parameters shown in Section 4, and a total number of 128 ROB entries. Results (not shown here) point out a performance loss of 9.2% and 20.2% for integer and floating-point benchmarks, respectively, when using the multithreaded machine with private ROBs.

Of course, multithreaded processors should not be evaluated only in lightly loaded systems, which would be unfair. On the other hand, to assume that the system is always running a heavy workload is neither a realistic way to measure performance. We claim that this is an issue that needs to be considered for a fair evaluation of multithreaded processors.

In this work, we propose a ROB sharing strategy, called *paired ROB*, that takes into consideration the fact that task parallelism is not always available to fully utilize resources of multithreaded processors. First, an evaluation methodology is proposed based on a formal definition of the available task parallelism. Then, paired ROBs are shown to perform similarly to shared ROBs for low task parallelism (with a lower hardware cost), and close to private ROBs for high task parallelism.

The rest of this paper is structured as follows. Section 2 analyzes advantages and disadvantages of private and shared ROBs, and describes the paired ROB proposal. Section 3 defines in a formal way the available workload parallelism, and details the simulation methodology. Section 4 shows simulation results, and finally, some related work and concluding remarks are presented.

2 ROB Sharing Strategies

In this section, three different ROB sharing strategies are analyzed, namely private, one-queue (shared) and paired ROBs. The first two sharing strategies represent basic design options, which suffer from some disadvantages regarding either complexity or performance. These drawbacks are smoothed by paired ROBs, by trying to gather the best properties of both approaches.

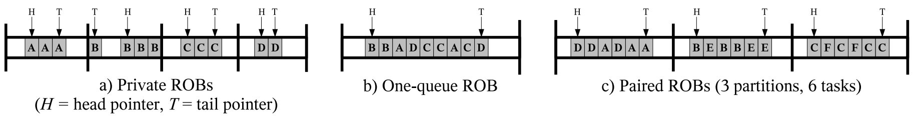

Fig. 1. ROB sharing strategies

The hardware cost analysis presented in this section is based on the required accesses to the ROB, which usually occur in the *dispatch* (insertion), *commit* (extraction), and *recover* (squash) stages of the processor pipeline. In what follows, the number of threads supported by a multithreaded system is represented as n, while s stands for the *total* ROB size in number of entries (i.e., the sum of all per-thread ROB sizes). For the sake of simplicity, a one-way processor is assumed, but results can be easily generalized to w-way superscalar processors, where ROBs are implemented by using w-port RAMs [9].

2.1 Private ROBs

The first straightforward sharing strategy consists in not sharing the ROB at all. The global ROB is split into different partitions, which act as independent FIFO queues, each one associated to a thread (Figure 1a). Each queue is managed by two pointers, namely *head (H)* and *tail (T)*, which track the insertion and extraction point into and from the corresponding queue. The required hardware structures and the actions involving the ROB are the following:

At the dispatch stage, instructions are placed in the ROB at the position indicated by the tail pointer. To this aim, a demultiplexer (or *decoder*), controlled by the ROB tail pointer, sends the instruction information to the corresponding ROB entry. Private ROBs require n independent demultiplexers with $\frac{s}{n}$ outputs, and an additional n-output demultiplexer selects the target private ROB.

At the commit stage, instructions are extracted from the ROB head. Private ROBs can perform this step in parallel with n $\frac{s}{n}$-input multiplexers, which transmit the extracted instruction into the rest of the commit logic.

Finally, the mispeculation recovery mechanism squashes some or all instructions in the ROB. If recovery is performed at the commit stage, the ROB must be just emptied, that is, the head and tail pointers must be reset. On the contrary, if recovery is implemented at the writeback stage, only instructions younger than the recovered one must be squashed, which implies an adjustment of the tail pointer. In any case, recovery requires no read, write, or selection of ROB entries; related hardware must just update the queue pointers.

2.2 One-Queue ROB

In the opposite extreme, the *one-queue ROB* is based on a fully shared circular queue, where all threads place their instructions in the global dispatch order (Figure 1b). There are some advantages and drawbacks in this approach, quantitatively evaluated in Section 4, and described next:

Disadvantages

Inter-thread blocking at commit. The fact that two or more threads share a single FIFO queue can lead to a situation in which the oldest instruction of a thread is ready to commit, but unable to do it for not being located at the ROB head. If the ROB head is occupied by an uncompleted instruction of any other thread, a waste of commit bandwidth is incurred.

Holes at recovery. In the one-queue ROB, instructions from different threads can be intermingled along the structure. This fact breaks the simplicity of a recovery mechanism in which only pointer updates are performed. Instead, only those instructions corresponding to the recovered thread are squashed on mispeculation, leaving empty intermingled slots or holes. These holes remain in the ROB until their are drained at the commit stage, and cannot be assigned to new decoded instruction, so they can cause a premature pipeline stall due to lack of space in the ROB.

Thread starvation. An intuitively potential advantage of a one-queue ROB is that instructions of any hardware thread may occupy all s ROB entries, in contrast to private ROBs, which restrict the ROB usage of a single thread to $\frac{s}{n}$ entries. Although this property brings flexibility, it has been demonstrated that benefits are not straightforwardly achieved [4]. In fact, performance losses occur when stalled threads (e.g., due to a long memory operation) uselessly occupy large portions of the shared ROB, preventing active ones from inserting new instructions into the pipeline.

Regarding hardware cost, one-queue ROBs have a more expensive implementation than private ROBs. For the same total number of entries s, the complexity of hardware structures increases. For dispatch, an s-output demultiplexer is needed, which has a higher delay and area than the n $\frac{s}{n}$-output demultiplexers of the private approach; for commit, an s-input multiplexer is required; finally, recovery can be implemented with a bit line per thread that turns the associated instructions into empty slots.

Advantages

Though the cited disadvantages could make one believe that shared ROBs do not deserve any interest at all, there are situations where this approach completely outperforms any other sharing strategy. Consider a multithreaded processor supporting 8 hardware threads and using private ROBs. A highly intensive computation environment can take take full advantage of multithreading in such machine by running 8 threads on it 100% of the time. However, 8 tasks may not be always available in the system.

The opposite extreme (discarding utter idleness of the system) is the case where only one single software context is being executed. Private ROBs constrain to use only $\frac{1}{8}$ of the total number of ROB entries available. However, a shared ROB permits a full occupation of the ROB by the single software context, enabling 8 times more instructions in flight, while not incurring any previously cited performance-related disadvantage of the one-queue ROB.

2.3 Paired ROBs

After analyzing the pros and cons of private and one-queue ROBs, a new sharing strategy, referred to as *paired ROB*, is proposed with the aim of gathering the main advantages of both private and one-queue ROBs.

Assuming a multithreaded processor with n hardware threads, and being s the total number of available ROB entries, the global ROB is partitioned into $\frac{n}{2}$ independent structures (see Figure 1c). Each partition can be occupied by instructions from at most two threads. The limit of two threads per partition is supported by the results discussed in Section 4.1.

If the system executes less tasks than available hardware threads, the assignment of ROB partitions to tasks is carried out in such a way that the number of tasks sharing a ROB partition is minimized. In other words, new active threads try to allocate an empty ROB partition before starting to share other occupied partition. This policy avoids inter-thread blocking, holes at recovery and thread starvation as long as it is possible.

The benefits of paired ROBs lie both in terms of hardware complexity and performance. Regarding hardware complexity, paired ROBs need $\frac{n}{2}$ $\frac{2s}{n}$-output demultiplexers for dispatch, $\frac{n}{2}$ $\frac{2s}{n}$-input multiplexers for commit, and a single bit line for recovery (only instructions from two different threads are queued in each ROB). The number of queue pointer sets is also reduced to $\frac{n}{2}$. This complexity is higher than for private ROBs, but lower than for one-queue ROBs.

3 Multitask Degree (MTD)

The effectiveness of a specific ROB sharing strategy can strongly vary depending on the number of tasks running on the system. To take this effect into account, the concept of Multitask Degree (MTD) is first defined in this section.

We define the MTD as a value between 0 and 1 that represents the average task parallelism present in a non-idle multithreaded system. A system with a value of $MTD = 0$ is characterized by running a single task all the time, while a value of 1 means that the system if fully loaded, that is, the number of tasks matches the number of hardware threads during all the execution time. Analytically, the MTD can be expressed as $\sum_{i=1}^{n} \frac{i-1}{n-1} t_i$, where n is the number of hardware threads of the system, and t_i is the fraction of the total execution time in which there are exactly i tasks running in the system. For example, in a 2-threaded processor, a value of $MTD = 0.5$ means that 50% of the time only one task is running, while two tasks are running the rest of the time. In general, the MTD value establishes the average number of running tasks as $MTD*(n-1)+1$. Figure 2 plots this equation for a 2- and 4-threaded system.

For the specific case of 2-threaded systems, the MTD conveys univocal information about the distribution of the system load. For example, a 2-threaded system with $MTD = 0.3$ is characterized by executing 1 task 30% of the time and 2 tasks 70% of the time. The MTD, however, does not provide any information about the load distribution of a system with more than 2 threads. For

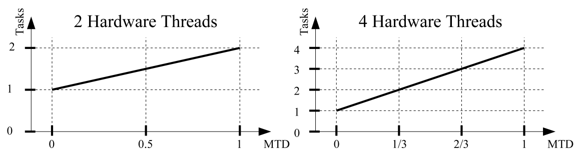

Fig. 2. Number of tasks as a function of the MTD value in 2- and 4-threaded processors

example, a 5-threaded processor with an MTD of 0.5 (which has an average load of $0.5 * (5 - 1) + 1 = 3$ tasks) can either execute 50% of the time 1 task and 5 tasks the rest of the time, or it can execute 2 and 4 tasks during equal portions of the time, or it can execute 3 tasks all the time, etc. In other words, there are multiple combinations of system loads and fractions of time that lead to the same resulting average value.

In order to cut down the number of possible combinations and simplify the analysis, we define the concept of *steady multithreaded processor*. An n-threaded processor is said to be *steady*, or to have a *steady load*, when the number of running tasks is exactly either x or $x + 1$ ($1 \leq x < n$). In this way, we limit the model to execution periods where the number of active threads can increase or decrease at most by one.

With this simplification, the number of possible load distributions is reduced to just one, which means that the MTD indeed characterizes the system load distribution univocally in a steady multithreaded processor. For instance, a steady 3-threaded system with $MTD = 0.5$ can be only executing 2 tasks all the time, while the same system with $MTD = 0.25$ can be only achieved by executing 1 task 50% of the time and 2 tasks the rest of the time. Finally, notice that this simplification keeps the model realistic, since operating systems usually assign processes or threads to the CPU in an incremental way. When applied to steady machines, the MTD is hereafter referred to as *Steady Multitask Degree* ($sMTD$).

3.1 Simulation Methodology

Experimental results presented in this paper evaluate different ROB sharing strategies on multithreaded processors with 2, 4, and 8 hardware threads. For each of them, different levels of task parallelism are evaluated, by presenting results as a function of $sMTD$. The algorithm used to obtain a continuous range of performance results from a finite set of simulations is presented next. Consider a steady multithreaded system with n hardware threads, in which a specific sharing strategy is evaluated.

First, n groups of simulations are launched. In the first group, only one task is run on the system, and each simulation uses a different benchmark from the SPEC2000 suite. In the second group, each experiment executes two instances of the same benchmark, each of them running on a different hardware thread, and so on.

Each simulation provides a performance value (IPC), whose harmonic mean is computed independently for each group. In this way, n average performance values are obtained, namely IPC_1, IPC_2, ..., IPC_n. IPC_1 stands for the average performance of the n-threaded processor when executing one single task 100% of the time; IPC_2 quantifies the average performance when executing two tasks during all the simulation, and so on.

Since the objective is to obtain a single performance value (from now on IPC_H) as a function of $sMTD$, individual IPC_i values must be combined by assigning weighting factors to each one. These factors, called hereafter ω_i, are likewise a function of $sMTD$, and determine how strongly each IPC_i must contribute to the computation of IPC_H for a given task parallelism. IPC_H can be defined as a weighted harmonic mean, given by the following equation:

$$IPC_H(sMTD) = \frac{1}{\frac{\omega_1(sMTD)}{IPC_1} + \frac{\omega_2(sMTD)}{IPC_2} + ... + \frac{\omega_n(sMTD)}{IPC_n}}$$

This equation will be used in Section 4 to depict the performance achieved by a given design under different conditions of task parallelism on the system.

The weighting functions ω_i must fulfill the following conditions to be consistent with the $sMTD$ definition:

i) IPC_1 must be weighted to 1 for $sMTD = 0$. The reason is that an $sMTD = 0$ represents a system running 1 task all the time, and thus, the performance of this system is specified just by IPC_1. Mathematically, $\omega_1(0) = 1$ and $\omega_i(0) = 0$ for $i \neq 1$.

ii) Symmetrically, IPC_n must be weighted to 1 for $sMTD = 1$. Mathematically, $\omega_n(1) = 1$ and $\omega_i(1) = 0$ for $i \neq n$.

iii) Finally, if the system is steady, at most two IPC values can be combined with a weight other than 0, and they must be consecutive (e.g. IPC_2 and IPC_3). Mathematically, $\nexists \omega_i, \omega_j | \omega_i \neq 0 \wedge \omega_j \neq 0 \wedge i - j > 1$.

For a generic (non-steady) n-threaded processor with $n > 2$, it is possible to find different weighting functions that comply with conditions *i* and *ii*. Nevertheless,

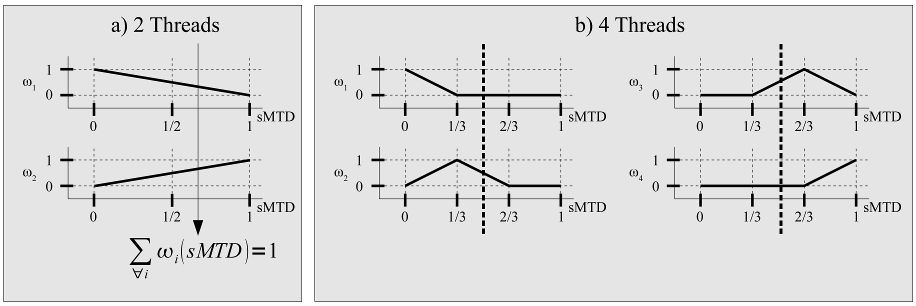

Fig. 3. Weighting functions $\omega_i(sMTD)$ in 2- and 4-threaded systems

Table 1. Baseline processor parameters

Parameter	Configuration
Machine width	8 hardware threads, 8-way fetch/issue/commit.
Storage resources	128-entry shared IQ, 64-entry shared LQ, ROBs with 32 entries per thread, per-thread 196-entry register file.
Functional units (Count/Delay)	Int.Add. (8/2), Int.Mult. (2/3), Int.Div. (2/20), Fp.Add. (2/4), Fp.Mult. (2/8), Fp.Div. (2,40).
L1 Caches (data & inst)	32KB, 2-way, 64-byte line, private per thread, 2 cycles.
L2 Cache (unified)	1MB, 8-way, 64-byte line, shared, 10 cycles.
Branch predictor	McFarling with 4K-entry gShare and 4K-entry Bimodal, 1024-entry 2-way BTB.
TLBs	16K, 4-way, shared.
Main memory	200 cycles

these functions become univocal when the additional restriction is imposed that the system load be steady (condition *iii*). Figure 3 shows a graphical representation of the weighting functions for 2- and 4-threaded steady machines. The fulfillment of condition *iii* can be observed, for example, in Figure 3b where the vertical dashed line cuts all ω_i at $sMTD = 0.5$. At this position, only ω_2 and ω_3 take a value other than 0.

Notice that, for a specific $sMTD$, the sum of the weights is always 1, which makes it unnecessary to normalize them when calculating IPC_H. The weighting functions for an 8-threaded machine are not shown, since they can be easily deduced from the conditions and examples above.

4 Experimental Results

The parameters of the modeled machine are listed in Table 1. This machine is able to fetch multiple instructions from different threads at the same cycle, by using the ICOUNT [3] fetch policy. At the dispatch stage, instructions are picked up from the fetch queue, and registers are renamed using per thread private register files. The misprediction recovery mechanism is triggered at the writeback stage, whenever a mispredicted branch is resolved.

Results shown in this section correspond to experiments with 32 ROB entries per hardware thread. Additional experiments have also been conducted with other ROB sizes, and only slight differences are observed. The simulation environment is Multi2Sim [10], a model of multicore-multithreaded processors, sharing strategies of processor resources, instruction fetch policies and thread priority assignment, among others. Benchmarks from the SPEC2000 suite have been used for the experiments. For each run, 10^8 instructions are executed from each thread, after warming up the system with another 10^8 instructions.

4.1 One-Queue ROBs

As explained in Section 2.2, one-queue ROBs have some disadvantages, such as inter-thread blocking at commit and holes at recovery. Figure 4 quantifies the

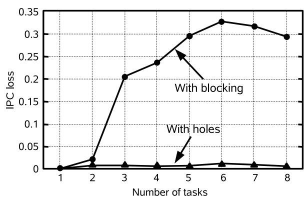

Fig. 4. Impact on IPC of inter-thread blocking and holes at recovery

negative impact on performance of these phenomena when varying the number of threads that share the same ROB.

To evaluate in an isolate manner the performance damage incurred by each phenomenon, a baseline performance value (IPC_{ideal}) has been obtained from a set of simulations of an ideal one-queue ROB. The inter-thread blocking effect is removed from this model by improving the commit logic so that it can select the oldest instruction of a thread regardless of its location. Additionally, the ROB is instantaneously collapsed when a slot is squashed, preventing holes at recovery. Eight different IPC_{ideal} values are obtained for simulations running from one up to eight tasks on the 8-threaded baseline machine.

Next, eight different performance values ($IPC_{blocking}$) are obtained, following the same procedure, and using a model of a semi-ideal one-queue ROB that does not hide the effect of inter-thread blocking. The curve shown in Figure 4 labeled *with blocking* represents the performance loss that the inter-thread blocking effect causes on the one-queue ROB machine, by means of the following equation:

$$IPC_{loss} = 1 - \frac{IPC_{blocking}}{IPC_{ideal}}$$

Likewise, the curve labeled *with holes* plots the IPC loss of the semi-ideal one-queue ROB machine that only suffers from the negative impact on performance that *holes at recovery* incur. This curve is obtained by extracting the IPC_{holes} performance value of an additional set of simulations, and using the corresponding analogous IPC_{loss} equation.

Results show that *inter-thread blocking* originates a performance loss of almost 21% when three tasks are active in the system, but this penalty is reduced to less than 3% for two tasks sharing one ROB. On the other hand, the curve corresponding to the *holes at recovery* effect shows only a slight performance degradation for any number of tasks running on the system, mainly due to small misprediction rates and fast holes draining.

4.2 ROB Sharing Strategies

Figure 5 shows the performance achieved by private, one-queue and paired ROBs on steady multithreaded processors, ranging the $sMTD$ from 0 to 1, and exploring 2-, 4-, and 8-threaded systems. This range represents a wide set of commercial products that implement different number of hardware threads [11][12]. The

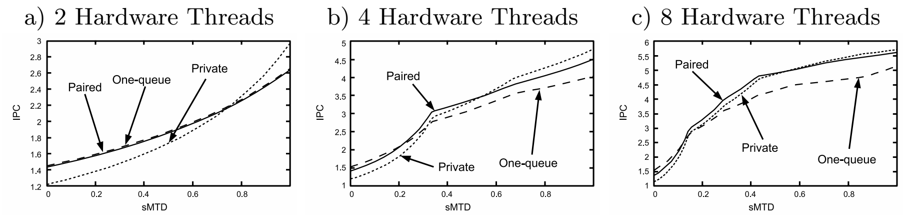

Fig. 5. ROB sharing strategies for different *sMTD* values

performance values shown here are computed by means of the $IPC_H(sMTD)$ function, by following the steps presented in Section 3.1.

In 2-threaded processors (Figure 5a), whose load is always steady by definition, a paired ROB groups both threads in a single ROB, so it is equivalent to the one-queue ROB, as observed in the overlapped curves. When a 2-threaded system is tested using a single task ($sMTD = 0$), performance increases by more than 20% using the one-queue or paired ROB compared with the private approach. On the opposite extreme, an $sMTD = 1$ makes private ROBs outperform the other sharing strategies by 12%.

We can also appreciate that the curves corresponding to all three sharing strategies cut each other at an *sMTD* of approximately 0.8, meaning that a larger *sMTD* is needed to make private ROBs stand out. In other words, private ROBs are only preferable in terms of performance when two tasks are running in the system for more than 80% of the time. This occurs frequently in systems aimed at high performance computing, but it is not as common in personal computers, where multithreading is being also implemented.

In a steady 4-threaded processor (Figure 5b), the following observations can be made. In the *sMTD* range [0...0.35], the one-queue ROB performs better than private ROBs, but this behavior is inverted for the rest of *sMTD* values. The advantages of paired ROBs can be clearly noticed in this figure. First, we can see that paired ROBs constitute the best approach for $sMTD =$ [0.2...0.55]. Moreover, it performs 16% and 13% better for extreme *sMTD*s (i.e., 0 and 1, respectively) compared to the worst approach in each case (i.e., private and one-queue, respectively). The performance loss compared to the best approach in each case is only 7% and 5%, respectively. These observations show paired ROBs as a trade-off solution to improve single-thread performance in lightly loaded multithreaded environments, while exploiting thread level parallelism when multiple tasks run on the system.

The steady 8-threaded processor (Figure 5c) shows similar results as the 4-threaded processor, with an additional advantage for paired ROBs. Namely, the range of *sMTD* values in which paired ROBs are the most appropriate solution grows to [0.15...0.58]. Again, paired ROBs show up as a good trade-off ROB sharing approach, which scales with the number of hardware threads in the system.

5 Related Work

Two previous works [7][13] compare the performance of an aggressive shared ROB, i.e., not affected by *inter-thread blocking* and *holes at recovery*, versus private ROBs. Both papers conclude that the small performance benefits obtained by these aggressive shared ROBs do not justify their higher implementation complexity.

Other related works that study the impact of distributing the resources of the SMT processor (including the ROB) assume a fixed sharing strategy. Sharkey et al. [4] propose a dynamic private ROB sharing strategy that avoids thread starvation by not allocating ROB entries of memory bounded threads. El-Moursy et al. [14] use a private ROB configuration to explore the optimal resource partitioning of a Clustered Multi-Threaded (CMT) processor. In [5] and [6] an aggressive shared ROB is distributed among the threads. In [5], Cazorla et al. dynamically classify each thread by using a set of counters to properly assign processor resources. In [6], Choi and Yeung propose a learning-based resource partitioning strategy which relies on performance metrics that can be obtained offline or during execution.

Finally, some research efforts [15][16] are devoted to dynamically assign threads to the back-ends of a given clustered multithreaded architecture. In [15], each backend can only support one thread at a given execution time, although thread switching is allowed by sharing the ROB temporarily. In [16], the back-ends are heterogeneous and multithreaded. However, each thread has its own private ROB.

6 Conclusions

In this paper, we have proposed paired ROBs as a new sharing strategy of the ROB in SMT processors. By considering the number of available software tasks allocated to hardware threads, we have developed an evaluation methodology to compare paired ROBs with the fully private and fully shared approaches. As a trade-off solution, paired ROBs minimize the effects known as inter-thread blocking, holes at recovery and starvation, while conferring flexibility to allocate ROB entries.

Experimental results show that paired ROBs provide higher performance than private ROBs for $sMTD$ values lower than 0.5 in 4- and 8-threaded processors. For 2-threaded machines, paired ROBs outperform private ROBs for an $sMTD$ up to 0.8. In all cases, the implementation of paired ROBs has similar complexity than private ROBs. This fact makes paired ROBs a cost-effective ROB sharing scheme.

Acknowledgements

This work is supported by Spanish CICYT under Grant TIN2006-15516-C04-01, by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, and by Universidad Politécnica de Valencia under Grant PAID-06-07-20080029.

References

1. Tullsen, D., Eggers, S., Levy, H.: Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: Proc. of the 22nd Annual International Symposium on Computer Architecture (1995)
2. El-Moursy, A., Albonesi, D.H.: Front-End Policies for Improved Issue Efficiency in SMT Processors. In: Proc. of the 9th International Conference on High Performance Computer Architecture (February 2003)
3. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In: Proc. of the 23rd Annual International Symposium on Computer Architecture (May 1996)
4. Sharkey, J., Balkan, D., Ponomarev, D.: Adaptive Reorder Buffers for SMT Processors. In: Proc. of the 15 International Conference on Parallel Architectures and Compilation Techniques, pp. 244–253 (2006)
5. Cazorla, F.J., Ramírez, A., Valero, M., Fernández, E.: Dynamically Controlled Resource Allocation in SMT Processors. In: Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (December 2004)
6. Choi, S., Yeung, D.: Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In: Proc. of the 33rd Annual International Symposium on Computer Architecture (June 2006)
7. Raasch, S.E., Reinhardt, S.K.: The Impact of Resource Partitioning on SMT Processors. In: Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques (October 2003)
8. Sharkey, J., Ponomarev, D.V.: Efficient Instruction Schedulers for SMT Processors. In: Proc. of the 12th International Symposium on High-Performance Computer Architecture (February 2006)
9. Yeager, K.C.: The MIPS R10000 Superscalar Microprocessor. IEEE Micro. 16(2), 28–41 (1996)
10. Ubal, R., Sahuquillo, J., Petit, S., López, P.: Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In: Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing (October 2007), `http://www.multi2sim.org`
11. Intel Pentium Processor Extreme Edition (4 threads), `http://www.intel.com`
12. SPARC Enterprise T5240 (8 threads/core), `http://www.fujitsu.com/sparcenterprise`
13. Liu, C., Gaudiot, J.L.: Static Partitioning vs Dynamic Sharing of Resources in Simultaneous Multithreading Microarchitectures. In: Cao, J., Nejdl, W., Xu, M. (eds.) APPT 2005. LNCS, vol. 3756, pp. 81–90. Springer, Heidelberg (2005)
14. El-Moursy, A., Garg, R., Albonesi, D.H., Dwarkadas, S.: Partitioning Multi-Threaded Processors with a Large Number of Threads. In: Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (March 2005)
15. Latorre, F., González, J., González, A.: Back-end Assignment Schemes for Clustered Multithreaded Processors. In: Proc. of the 18th Annual international Conference on Supercomputing (June 2004)
16. Acosta, C., Falcon, A., Ramírez, A.: A Complexity-Effective Simultaneous Multithreading Architecture. In: Proc. of the 2005 International Conference on Parallel Processing (June 2005)