Efficient Resources Assignment Schemes for Clustered Multithreaded Processors

Fernando Latorre
Intel Barcelona Research Center
Intel Labs - UPC
fernando.latorre@intel.com

José González
Intel Barcelona Research Center
Intel Labs - UPC
pepe.gonzalez@intel.com

Antonio González
Intel Barcelona Research Center
Intel Labs - UPC
antonio.gonzalez@intel.com

Abstract

New feature sizes provide larger number of transistors per chip that architects could use in order to further exploit instruction level parallelism. However, these technologies bring also new challenges that complicate conventional monolithic processor designs. On the one hand, exploiting instruction level parallelism is leading us to diminishing returns and therefore exploiting other sources of parallelism like thread level parallelism is needed in order to keep raising performance with a reasonable hardware complexity. On the other hand, clustering architectures have been widely studied in order to reduce the inherent complexity of current monolithic processors. This paper studies the synergies and trade-offs between two concepts, clustering and simultaneous multithreading (SMT), in order to understand the reasons why conventional SMT resource assignment schemes are not so effective in clustered processors. These trade-offs are used to propose a novel resource assignment scheme that gets and average speed up of 17.6% versus Icount improving fairness in 24%.

1. Introduction

Every new feature size that industry is manufacturing provides a larger number of transistors per chip. This increase in the number of available transistors allows computer architects to face more complex hardware designs in order to improve the processor performance. However, these new feature sizes also bring new challenges that must be addressed.

Every new feature size transistors become smaller and therefore faster. Unfortunately, wire delays are not reduced the same way. Smaller transistors reduce the area of the chip components and thus the length of the wires needed to communicate different components becomes shorter. Even though this reduction on distance also reduces wire delays, new feature sizes increase wires resistance making signals to travel slower. Therefore, designers must meet a trade-off between wire length and resistance in order not to turn communications into a bottleneck [8]. In conclusion, conventional monolithic designs would find in wire delays one of their main limiting factors when future technologies are in use.

Other important challenges coming out in the current processor designs are the thermal and power budgets at which these designs must operate [10]. Increasing the number of transistors implemented in a given area makes the activity in this area grow along with the power consumption. This additional power consumption turns into heat that must be dissipated from a tiny area.

Proposals like clustering have been analyzed during the last decades in order to alleviate the aforementioned drawbacks. Clustering has been explored in many individual processor components like issue queues or register files. Besides, some authors proposed splitting the processor back-end into multiple clusters where instructions are steered for execution. These clusters are small, simple and are able to operate at high frequencies at the expenses of sometimes communicating through slow and large connections incurring in performance penalties [13].

On the other hand, ILP (Instruction Level Parallelism) is limited and cumbersome to exploit. Architectural enhancements usually raise ILP performance at the expenses of important increase on power consumption. This scenario made researchers to look for alternative sources of parallelism like TLP (Thread Level Parallelism). An example of this kind of parallelism seriously considered by the industry is SMT (Simultaneous Multithreading) [26]. Processors enabling this technology are able to execute multiple applications or threads in parallel augmenting the

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

probability of finding independent instructions to execute and therefore keeping the processor busy. Moreover, current workloads normally comprise multiple applications and in the future is expected these parallel workloads grow playing in favor of this technology.

In conclusion, clustering and SMT are two important players to be considered in future designs in order to have powerful processor able to effectively deal with multithreaded workloads when possible but being also able to efficiently exploit ILP when the number of running threads is limited. On the one hand, clustering will allow designers to deal with wire delays and keep designs simple in order to exploit ILP while meeting a given power and thermal budget [4]. On the other hand, SMT will enable the processor to run workloads consisting of multiple threads that is becoming the most common scenario nowadays. However, few studies have been done regarding the way these two technologies run together.

In this paper we explore the synergy between SMT and clustering. Different alternatives proposed in the literature to distribute resources among threads in SMT processors are evaluated and the reasons why they are not adequate for clustering machines have been studied. Moreover, we take advantage of the study to propose a novel resource assignment scheme designed to work on these clustered approaches getting performance benefits of 17.6% compared to *Icount* and improving fairness in 24%.

The paper is organized as follow: first, we present some related work in section 2 and describe the baseline architecture in section 3. Then, the simulation methodology is discussed in section 4 and the evaluation of the different resource assignment techniques along with our proposal are shown in section 5. Finally, conclusions are presented in section 6.

2. Related work

Clustered microarchitectures have been shown as an effective way to deal with wire delays and complexity [4]. The importance of these architectures has motivated numerous proposals for increasing performance [4][12][13][16][21][22][23] or reducing power dissipation [7][15]. As an example, some conventional processors such as the Alpha 21264 [7] implemented a clustered integer execution core.

Many authors have proposed using the increasing transistor budget on a chip to exploit TLP (see [1][2][6][24][25][32] among many others), in addition to ILP (instruction-level parallelism). However, not much work has been published based on the idea of combining both paradigms in a synergistic way: clustering and multithreading. Krishnan et al. [6] compares SMT clustered architectures where the threads are statically assigned to a number of non clustered execution cores. This design is conceptually similar to the IBM Power5 [27] where two Simultaneous Multithreaded cores are implemented. On the other hand, for Raasch et al. [24] each subset of threads are executed in a single monolithic execution core using all its resources where different partition schemes are evaluated. At the end of the paper they make a first evaluation of a SMT clustered approach where instructions from both threads are steered to the clusters in a round robin fashion. Latorre et al. [28] also evaluate a clustered architecture but in this case the clusters are private per thread and they are either dynamically or statically assigned.

Collins et al. explores multiple ways of processor partitioning in [29]. In this paper the authors propose a thread assignment scheme where the occupancy of the issue queue is considered. We analyze the advantages and disadvantages of this technique among other alternatives and demonstrate that the register file is sometimes an important source of thread starvation that must also be taken into account in clustered architectures.

The goal of this paper is to have a better understanding of the trade-offs between clustering and multithreading while defining a resource assignment scheme for this scenario. For this reason, simple SMT resource assignment policies have been chosen as a first step. Then, adapting more sophisticated schemes like [20][30] and [32] to make them fit in a clustered processor by using the conclusions of this study is part of our future work.

3. Description of the architecture

The baseline architecture is like the one proposed in [12]. A block diagram is shown in Figure 1. It consists

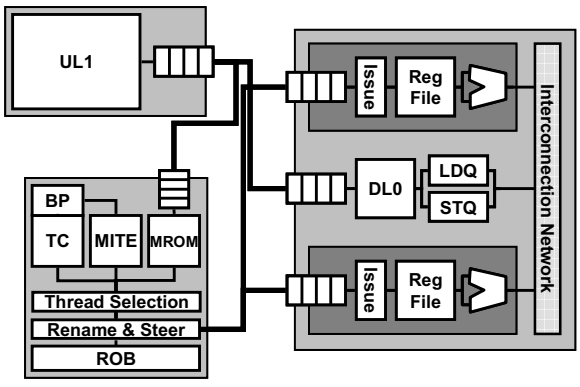

Figure 1. Baseline architecture.

Table 1. Baseline processor configuration.

Parameter	Value	Parameter	Value
Fetch width	6	**Commit width**	6
Misprediction pipeline	14	**ROB size**	128 per thread
Indirect branch	4096	**Gshare entries**	32K
ITLB entries	1024	**ITLB assoc.**	8
Trace Cache size	32K uops	**Issue rate per cluster**	Port0:int,fp,simd Port1:int,fp,simd Port2: int, mem
Issue queue size per	32-64	**MOB**	128
Int. physical registers	64-128	**FP physical registers per**	64-128
SSE physical registers	64-128	**DTLB entries**	1024
DTLB assoc	8	**L1 ports**	2 read/ 2 write
L1 assoc	2	**L1 size**	32KBytes
L1 hit latency	1 cycle	**L2 assoc**	8
L2 size	4MB	**L2 hit latency**	12 cycles
# Point to Point Links	2	**Point to Point latency**	1 cycle
# Data buses (L1 to L2)	2	**Memory Latency**	60 cycles

of a monolithic front-end in charge of fetching, decoding and renaming instructions and a clustered back-end. This front-end fetches x86 macro-instructions and translates them into micro-operations that are stored in the trace cache. The main components in the front-end are the trace cache (TC) where micro-operations are stored, the instruction TLB (not shown in the figure), the branch predictor (BP), and the Macro Instruction Translation Engine (MITE) that translates macro-instructions into micro-operations before storing them into the TC. It also implements the instruction decoding, steering and renaming logic. Another important component in the front-end is a MROM in charge of decoding complex macro-operations like string moves. A detailed description of these components can be found in [14]. The front-end is able to fetch instructions from multiple threads in SMT mode. All main structures in the front-end are shared among the running threads except the global history register of the g-share branch predictor, the renaming tables (there is one per thread) and the ROB. This latter component is split into as many sections as threads are running in parallel as also described in [26]. However, instructions can only be fetched from one thread at a time and renamed from only one thread too. The resource assignment logic is in charge of deciding the thread to be fetched and the one to be renamed every cycle. The former selection policy is called fetch selection policy whereas the latter is called rename selection policy. Fetched instructions from every thread are stored into private queues residing inside the thread selection component. Hence, the rename selection policy chooses a thread from those which queue is not empty. In order to guarantee that the rename selection policy can choose any thread the fetch selection policy always fetches instructions from the thread with the lowest number of instructions in its queue. On the other hand, the rename selection policy is in charge of deciding from which thread instructions must be renamed and therefore steered to the back-ends. Thus, this selection scheme is the main responsible of fairly distributing the processor resources among the threads.

Decoded instructions are steered to one of the two clusters for execution following the dependence- and workload-based algorithm described in [12] (section 3.8). Inter-cluster communication is performed via copy instructions that are generated on-demand by the rename logic. Every cluster includes an issue queue where instructions wait until all their dependences have been computed and are eligible to be executed. It also includes two register files (integer, and floating point/SSE) where both speculative and architectural values are stored. Once an instruction leaves the issue queue, it reads its source operands either from the register files or the bypass logic and executes in one of the functional units of the cluster. Finally, a shared memory order buffer and memory hierarchy is used to process store and load operations.

4. Experimental methodology

The goal of a resource assignment scheme is to improve the resource utilization maximizing a certain metric defined beforehand. Determining the most adequate metric for SMT is still a great source of discussion and many metrics have been proposed during the last years like in [19][25][33]. In this study we will use two metrics in order to quantify the benefits of the techniques. On the one hand conventional throughput will be used to compare the amount of useful work (number of committed instructions) each technique is able to do per time unit. On the other hand, we consider that a system is fair if all the threads experience an equal slowdown compared to the performance they have when executed alone [17]. Hence, the fairness metric can be defined as the minimum ratio between the slowdowns of any two threads running in the system compared to its performance when running alone as shown in [33]. Note that a proposal with good fairness could have very bad throughput and the other way around (i.e. a fair technique could run two threads in parallel where both are slowed down in 90% but its throughput would be much less than running the two threads one after the other). Therefore, a novel SMT resource assignment scheme must improve the processor throughput but keeping fairness either at the same level or enhanced compared to the baseline.

4.1 Simulation methodology

The experiments have been conducted by using an in-house simulator that models the micro-architecture described in Section 3. The simulator is trace-driven but traces hold enough information to faithfully simulate wrong path execution. Our pool of benchmarks comprises of 120 2-threaded traces classified in 11 categories based on their characteristics. Moreover, for every category we have classified the traces in highly parallel traces and memory-bounded traces like in [19] in order to create workloads that cover as many different scenarios as possible. This classification is shown in Table 2. The processor baseline configuration is described in Table 1.

5. Managing shared resources

In order to design an effective resource assignment scheme, it is important to first identify the processor components that may cause thread starvation. These hardware components are those that are shared among threads and a thread allocates for long periods of time. For instance, functional units are typically shared among threads but they are used and released very fast so that starvation is infrequent. By contrast, other resources like issue queue slots or physical registers are allocated for very long periods of time and therefore thread starvation could be very common if they are not properly managed.

In this section we evaluate the impact different resource assignment schemes have on these two shared resources. This evaluation is done separately for each of the resources to better understand the way the resource assignment schemes manage every individual resource.

5.1 Issue queue entries

Our first study will address the management of the issue queue. Once an instruction is renamed it is steered to one of the clusters for execution. Previous

Table 2. Benchmarks.

Category	Description	Types	#wkloads
DH	Digital Home algorithms	ILP/MEM/MIX	3/3/2
FSPEC00	Floating Point benchmarks from SPEC2K	ILP/MEM/MIX	3/3/2
ISPEC00	Integer benchmarks from SPEC2K	ILP/MEM/MIX	3/3/2
Multimedia	Mpeg, speech recognition	ILP/MEM/MIX	3/3/2
Office	Power Point, Excel	ILP/MEM/MIX	3/3/2
Productivity	Sysmarks2K	ILP/MEM/MIX	3/3/2
Server	TPC traces	ILP/MEM/MIX	3/3/2
Workstation	CAD, rendering	ILP/MEM/MIX	3/3/2
Miscellanea	Games and matrix algorithms	ILP/MEM/MIX	3/3/2
ISPEC-FSPEC	Mixes of traces from ISPEC00 and FSPEC00	ILP/MEM/MIX	3/3/2
mixes	Mixes of traces from all categories	MIX	32

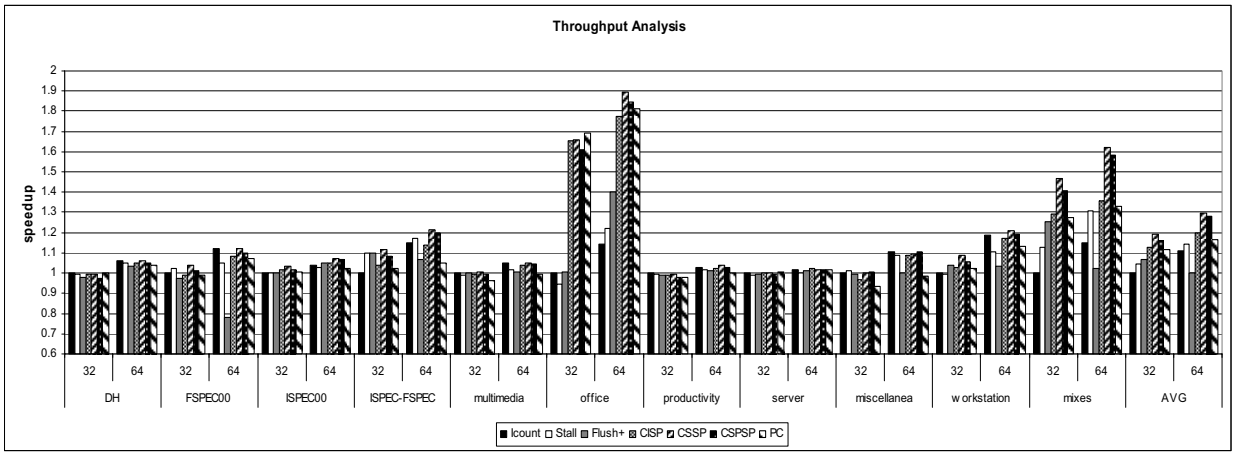

Figure 2. Throughput for the different IQ techniques using 32 and 64 issue queue entries per cluster. Performance is normalized to the obtained with 32 issue queue entries and Icount.

studies for single threaded processors demonstrated that instructions must be steered to a cluster (and thus to an issue queue) in such a way that inter-cluster communication is minimized but also workload is balanced among the different clusters [12]. Moreover, previous work on SMT for monolithic processors have also demonstrated that balancing under certain conditions the available issue queue entries among the running threads is crucial to reach good multi-threaded performance [3]. Therefore, it would seem that a good resource assignment scheme must deal with three main factors in order to properly handle the issue queues on a clustered machine: workload balance, inter-cluster communication and distribution of issue queue slots among threads. The resource assignment schemes described in Table 3 are evaluated in order to better understand the trade-off among these three factors.

Table 3. Resource assigment schemes evaluated to control the issue queue.

Technique	Description/Reason
Icount	Very effective and simple technique proposed in [1] where the thread with the lowest number of instructions between renaming stage and issue is selected.
Stall	It is implemented on top of Icount but stalls a thread that misses in L2 cache until the cache miss resolves. It was proposed in [19].
Flush+	This is a scheme proposed in [25] that improves the original Flush in [19]. It forces a thread with a L2 cache miss pending to release all allocated processor resources until the miss is solved. Flush+ improves the original when there are two threads with pending L2 misses. In Flush+ the one that missed the first is allowed to continue. This technique was also enhanced (Flush++) for the cases where the number of threads was higher than 2. However, since our workloads are two threaded we implemented Flush+.
Cluster-insensitive Static Partitioned (CISP):	This technique allows a thread to use 50% of the issue queue entries regardless the cluster where these entries are located and it has been proposed in papers like [31] to distribute the resources among clusters shared by multiple threads.
Cluster-sensitive Static Partitioned (CSSP):	This technique allows a thread to use 50% of the issue queue entries implemented per cluster.
Cluster-sensitive Partial Static Partitioned CSPSP):	It is like the previous one but only 25% of the entries of each cluster are guaranteed per thread. Threads compete for the rest of the issue queue entries.
Private clusters (PC):	It assigns a cluster to every thread and steers all instructions from a thread to the assigned cluster.

While the former four schemes are obtained from the literature, we propose the latter three in order to understand the impact the three factors (inter-thread communications, issue queue occupancy and workload balance) have on performance. All these schemes are implemented on top of the state-of-the-art steering mechanism proposed in [12] that steers instructions to the thread where most of their source operands reside in order to minimize communications and also controls workload balance.

Figure 2 shows the performance (in throughput) obtained by the different evaluated techniques using 32 and 64 issue queue entries per cluster. The physical register file and the reorder buffer are unbounded for this study in order to avoid side effects on these components. Therefore, the differences in performance are coming only from the way issue queue entries, workload balance and inter-cluster communications are managed by the schemes shown in Table 3.

As it can be seen in Figure 2 *Flush+* and *Stall* are usually helpful in order to increase performance when the number of IQ entries is an important bottleneck.

However, for situations where starvation in the IQ is not high like when the number of IQ entries is increased these techniques not only loose effectiveness but also degrade performance (for instance, *miscellanea* with 64 entries). The reason is that these techniques (especially *Flush+*) overreacts penalizing threads with cache misses and the other thread is not able to get enough performance benefits to compensate the penalties. On the other hand, techniques that statically partition the issue queue have a more stable behavior. It can be seen that the cluster-sensitive scheme (*CSSP*) usually outperforms the cluster-insensitive scheme (*CISP*) and the private assignment (*PC*).

For the sake of simplicity from now on we will focus our study on a configuration with 32 IQ entries per cluster. However, we have observed that the trend remains for configurations with 64 entries even though benefits are reduced because increasing the amount of resources available alleviates thread starvation.

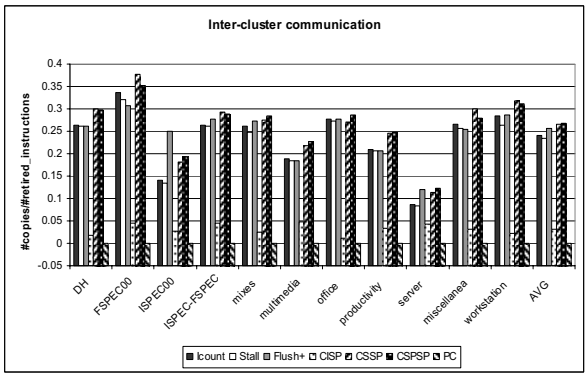

Figure 3. Analysis of inter-cluster communication.

Inter-cluster communication

In this section we analyze the relation between the number of values communicated among clusters by using each technique and the performance observed. Figure 3 shows the number of copies (inter-cluster communications) produced per retired instruction. On the one hand it can be seen that configurations like *PC* do not reach good performance (Figure 2) even though communications are avoided by sending instructions from the same thread to the same cluster. In this case performance is affected by workload imbalance as we will discuss in the next sections. By contrast, *CSSP* achieves the best performance having one of the highest amounts of copies per instruction (0.26 in average). Note that although *CISP* is the policy with the lowest amount of copies among those that allow instructions from a thread to go to both clusters; it does not reach as good performance as *CSSP*. The reason is that *CISP* does not allow a thread to occupy more than half of the total IQ entries but it does not specify the cluster where these entries must be located. Therefore, a steering logic based on dependences like the one being used in this study makes most of the instructions from the same thread to go to the same cluster. Overall, we have observed that *CISP* behaves almost like *PC* but it sometimes allows instructions from a thread to use the other cluster improving workload balance and then performance. However, although the same effect should be expected in other cluster-insensitive schemes like *Icount*, *Flush+* or *Stall*, they do not follow this trend as shown in Figure 3.

The main difference between *CISP* and the other cluster-insensitive approaches is that the former forbids a thread to occupy more than 50% of the total IQ entries. Then, in *CISP* a thread occupies most of a cluster and a little bit of the other before it is stalled. However, the other cluster-insensitive techniques do not have any additional constraint that prevents a thread from occupying both IQs. Therefore, in *Flush+*, *Stall* and *Icount* as soon as a thread is either stalled or flushed because of cache misses or branch mispredictions the other thread can invade both issue queues. Once the flushed/stalled thread resumes it is not guaranteed that it can steer its instructions to the cluster where their dependents reside. Since there are not a minimum number of entries reserved per cluster, the other thread may have populated the preferred cluster. Then, instructions are steered to the non preferred cluster instead. This situation makes that even though the steering logic favors instructions from the same thread to go to the same cluster as in *CISP*, threads are forced to ping pong within time between the two clusters increasing the number of copy instructions.

Finally, since *CSSP* splits the issue queue of each cluster into two, it produces a high number of copies. The steering logic favors instructions from the same thread to go to the same cluster; but as soon as the number of instructions from the same thread exceeds 50% of the issue queue of a cluster, *CSSP* forces this thread to go to the other cluster.

In conclusion, the ratio of inter-cluster communications is not crucial in clustered SMT architectures (conversely to what happen in single threaded ones) due to the fact that having two simultaneous threads partially hides the communication penalties.

Issue queue stalls

Figure 4 shows the number of renaming stalls because of lack of issue queue entries per number of retired instructions. Note that we consider stall when an instruction is not able to go to the preferred cluster because the issue queue is either full or exceeds the limit defined by the resource assignment scheme for that cluster. As it can be seen in Figure 4 resource assignment schemes like *Flush+* and *Stall* are very effective dealing with the issue queue entries. However, they are so conservative avoiding threads to eagerly allocate issue queue entries that performance sometimes drops apparently because of underutilization of the issue queue slots as shown in Figure 2. Interestingly, cluster-sensitive resource assignment schemes incur in the highest ratio of issue queue stalls. The reason is not that its management of issue queue slots is worse than the other techniques but a side effect of the cluster-sensitive policies. Actually, the management of the issue queue by *CISP* and *CSSP* should have similar efficiency in terms of renaming stalls. However, *CSSP* shows a higher ratio of stalls because it forces a thread to steer instructions to the non-preferred cluster (a cluster different than the chosen by the steering logic) when it exceeds 50% of the issue queue occupancy of the cluster. By contrast, *CISP* would still steer instructions to the preferred cluster showing a lower number of issue queue stalls than *CSSP*. Note though that the additional stalls *CSSP* has over *CISP* do not block the renaming logic but instructions are steered to the non-preferred cluster instead. Therefore, these additional stalls only incur in extra copies whose latency is typically hidden. Indeed, the number of stalls that make the renaming logic to stall is similar to *CSSP* and lower than *Icount*.

It has been seen in Figure 4 that *Flush+* and *Stall* are very effective preventing issue queue stalls. However, issue queue entries in clustered architectures are more abundant than in monolithic designs and therefore the benefits they get by preventing issue queue stalls is not enough to compensate the penalties incurred by stalling/flushing threads. On the other hand, *CISP* and *PC* also show a reduction on issue queue stalls compared to *Icount* and they make a good job minimizing inter-cluster communications as shown in Figure 3. Nevertheless, *CSSP* and *CSPSP* are the schemes that obtain the best performance. Even though issue queue occupancy and inter-cluster communication are important, workload balance is the main player a resource assignment scheme should consider.

Workload balance

The previous section has shown the reasons why conventional *Stall* and *Flush+* schemes are not adequate for clustered architectures. In this section we give some insights regarding the reasons why *CSSP* is better than *CISP* and *PC* by evaluating the impact of these techniques in the workload balance.

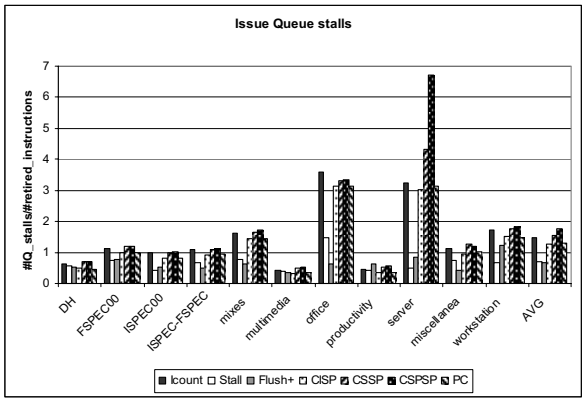

Figure 4. Analysis of stalls due to lack of issue queue entries.

Figure 5 measures the workload imbalance between clusters for *Icount*, *CISP*, *CSSP* and *PC*. Workload imbalance stands for the number of ready instructions that could not be executed in one cluster because of lack of issue slots but they could have been executed in the other cluster. For example, a workload imbalance of 1 for an integer instruction means that the instruction could not be executed because all issue slots in its cluster were busy but there were at least one free issue slot in the other cluster.

Although the steering logic implemented tries to minimize the workload imbalance among clusters, the resource assignment scheme sometimes decides to steer instructions to the non preferred cluster to reduce starvation and therefore affects the final workload balance.

Figure 5 shows the average workload imbalance for every category classified per type of instruction. Remember that every cluster has 3 execution ports for integer where two of them can also execute Fp/Simd operations and the other can also execute memory operations. Hence, 0 Integer for instance stands for the percentage of cycles where integer ready instructions in one cluster could not have been executed in any of the two clusters. On the other hand, 1 Fp/Simd stands for the percentage of cycles where a Fp/Simd instruction could not be executed in one cluster but the other had available execution ports. Thus, perfect workload balance would make sections 0 Integer, 0 Fp/Simd and 0 Mem to sum 100%.

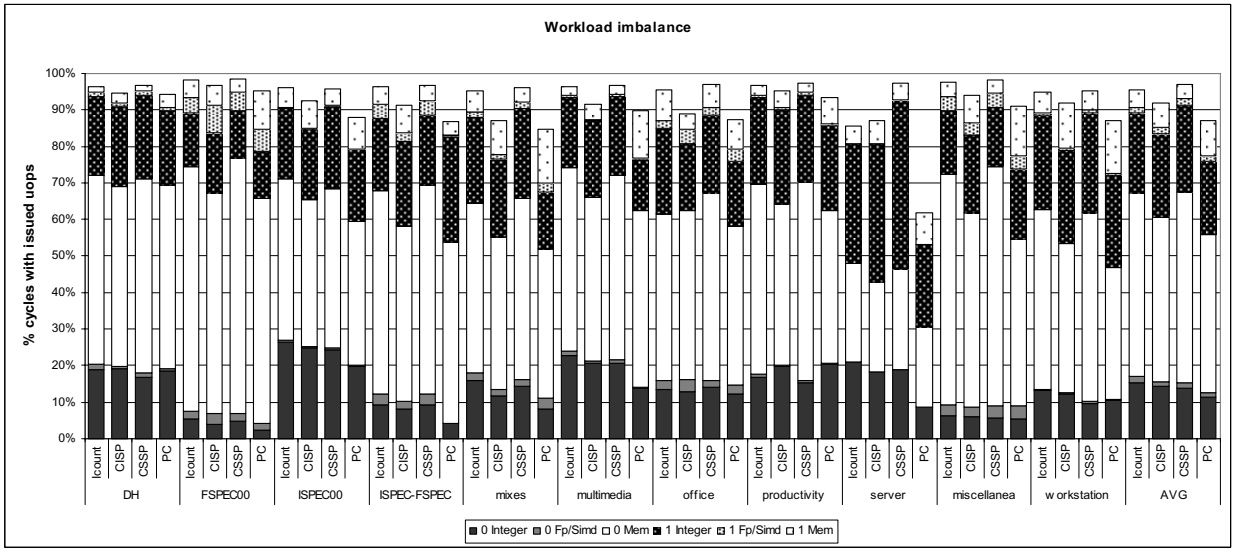

Figure 5. Workload-imbalance analysis.

Figure 5 shows CSSP has better workload balance than *PC* and *CISP*. *PC* statically partitions the issue bandwidth among threads by statically binding threads to clusters. However, splitting the issue bandwidth among threads degrades performance as demonstrated for monolithic designs by Raasch et al. in [24]. Figure 5 shows that in clustered architectures, even though statically binding clusters among threads is good in terms of inter-cluster communication, workload balance is dramatically reduced (for instance in *mixes* category). On the other hand, *CISP* does not statically split the issue bandwidth but the dependence based steering algorithm usually makes *CISP* to behave as *PC*. As commented before, *CISP* makes most of the instructions from a thread to go to the same cluster and just few of them to the other. Therefore, every cluster becomes eventually full of instructions from the same thread preventing the other thread from using it splitting the issue bandwidth. By contrast, *CSSP* guarantees issue queue slots for all threads in every cluster avoiding the partition of the issue bandwidth and then improving workload balance.

From this study we conclude that what matters in SMT clustered architectures is not only the number of issue queue entries available per thread as in monolithic designs but also the cluster where these entries reside. Therefore, the resource assignment scheme should be able to guarantee certain amount of issue queue entries per thread and cluster.

CSSP will be used from now on in the evaluations because it showed the best behavior controlling the allocation of issue queue entries.

5.2 Physical register file

The other main shared resource where thread starvation occurs is the physical register file. Our proposed architecture has two register files per cluster; one for integer values and the other for Fp/Simd data. The goal of the next experiment is to find out the best approach to handle the physical register files. For this experiment the configuration shown in Table 1 is used with 32 issue queue entries per cluster.

Figure 6 shows the throughput of the techniques shown in Table 4 with 64 and 128 registers per cluster normalized to the performance obtained by implementing *Icount* with 64 physical registers per cluster. As it can be seen, performance differences between having 64 and 128 registers is small so that the physical register file is not a big source of thread starvation for this size. However, even though the number of physical registers is in general enough to satisfy the demand of all workloads, some categories like *ISPEC00* incur in very high pressure on the integer

Table 4. Techniques evaluated to control the register file.

Technique	Description/Reason
Icount	Baseline.
CSSP	it is the technique that better managed the issue queues in the previous section.
Cluster-Sensitive Static Partioned Register File (CSSPRF)	CSSP handles the issue queue but a thread is only able to use half of the register file of each kind on each cluster.
Cluster-Insensitive Static Partitioned Register File (CISPRF)	CSSP handles the issue queue but a thread is only able to use half of the total register file of each kind no matter the cluster where registers are allocated.

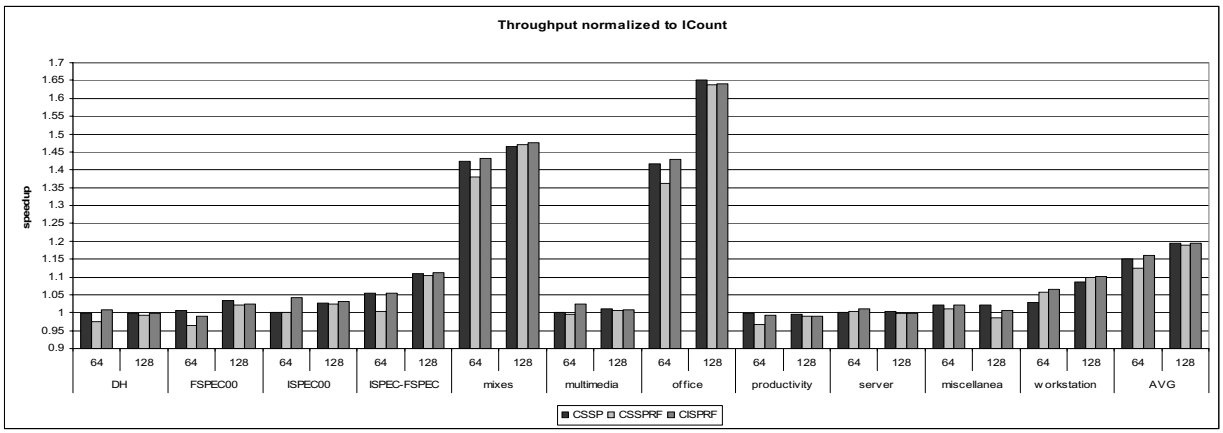

Figure 6. Throughput normalized to Icount for configurations with 64 and 128 physical register per cluster.

register file getting performance benefits of up to 14% (4% in average) when the register file is partitioned. By contrast, other categories like *ISPEC-FSPEC* or *mixes* show performance drops when partitioning the register file. The main difference between *ISPEC00* and *ISPEC-FSPEC* is that whereas the former only uses the integer register file and therefore it becomes a bottleneck, the latter executes a benchmark with high integer activity in parallel with another with low integer activity and high FP/Simd activity. Therefore, the required resources by the two workloads are almost disjoint reducing thread starvation. While resource partitioning reduces starvation when both threads use a resource, it also incurs in hardware underutilization when resources are barely shared as it happens in *ISPEC-FSPEC*. In conclusion, a dynamic mechanism that makes the partition depending on the pressure over the hardware resources is needed in order to avoid outliers due to either thread starvation or hardware underutilization.

It is important to determine whether this adaptive scheme should be implemented in a clustered-sensitive way or not. As it can be seen in Figure 6, *CSSPRF* always performs worse than *CISPRF*. The reason is that *CSSPRF* sometimes changes the decision taken by the steering logic and *CSSP*, degrading performance in the following situations:

- When the source operands reside in one cluster and *CSSP* allows the instruction to go to any of the two clusters; the register file control logic could make the instruction to go to a cluster different to where the source operands reside if the preferred one has its register file partition full. Therefore, the number of copies is increased while workload balance is not improved.
- When a thread has its issue queue partition full in one cluster and its register file partition full in the other cluster CSSP avoids the instruction to go to one cluster while CSSPRF avoids it to go to the other so that the instruction is stalled incurring in resource underutilization.

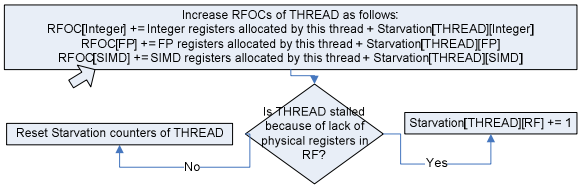

Figure 7. Update the RFOC and Starvation Counters every cycle.

Therefore, conflicting decisions between the management scheme for the issue queue and the physical register file could be avoided by handling one resource in a cluster-sensitive way while the other is handled in a cluster-insensitive fashion. Hence, since building a cluster-insensitive scheme for the issue queue has been demonstrated to be inadequate to keep the workload balance; we decided to make the management of the physical register file cluster-insensitive.

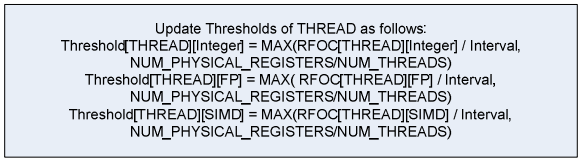

Figure 8. Update of thresholds per thread and type of register file at the end of the interval.

Dynamic Register File Scheme

In this section we propose a scheme that dynamically identifies the minimum number of registers of each type a thread needs and partitions the

register files accordingly. As it has been shown in the previous section this technique should be cluster-insensitive and it assumes a register file of each kind with as many registers as the sum of the registers implemented per cluster. Hence, the average number of registers a thread uses is computed in order to guarantee at least this amount for the execution of the thread. Once this requirement is satisfied for all threads; the rest of the registers in the register file can be allocated by any thread. This computation is done for the two types of register files (integer and Fp/Simd) independently. The average number of registers required per thread is computed by using a counter per thread and type of register file named *RFOC* (Register File Occupancy). This counter accumulates the number of registers each thread is using per cycle. Then, *RFOCs* are periodically checked in order to measure the average register requirements per register file and thread. This number of registers (up to half of the register file) will be guaranteed during the next period. The scheme does not allow private regions greater than half of the register file because it would not be fair for the other thread. This reservation of registers is done by using a threshold per thread and type of register file. When a thread exceeds its threshold it can only allocate registers as long as the register file can satisfy the reserved number of registers of the other thread. However, the average occupancy per thread is sometimes not properly quantified because a thread is starved by the other. This situation is handled by including an additional counter per thread and type of register file named Starvation. Whenever a thread is stalled because of lack of physical registers its appropriate Starvation counter is increased by one or reset otherwise. Hence, *RFOC* is incremented every cycle by the number of allocated registers plus the Starvation counter of the register file. This mechanism makes the threshold grow very fast when starvation occurs and therefore it gives a big private region to the starved thread to properly measure its average occupancy during the next period. Figure 7 shows the flow diagram of the computation to be done per cycle while Figure 8 shows the required computation to define the thresholds for the next interval. In our experiments we found that 128K cycles was a reasonable interval. We chose this value because it is a power of 2 so that dividing the *RFOC* by the interval is a simple shift operation.

Figure 9 shows the behavior of this dynamic scheme (*CDPRF*) for the workloads in the *ISPEC-FSPEC* category. We chose this category because this is where we found the main slowdowns for the static partitioning. As it can be seen, this simple dynamic approach minimizes the slowdowns of the static partitions and indeed turns them into speedups in some cases. In average *CDPRF* gets an additional 5% performance improvement on top of the 5% already obtained by *CSSP* in this category. However, since the categories where benchmarks have very similar register type requirements do not take advantage of this adaptation (the dynamic scheme ends up statically partition the register files) and the physical register file is not a bottleneck as important as the issue queue, this technique provides a modest 1.6% on top of the 16% already provided by *CSSP* for all the categories (*AVG All*). Nevertheless, it is very effective to fix those workloads that were losing performance because of register underutilization.

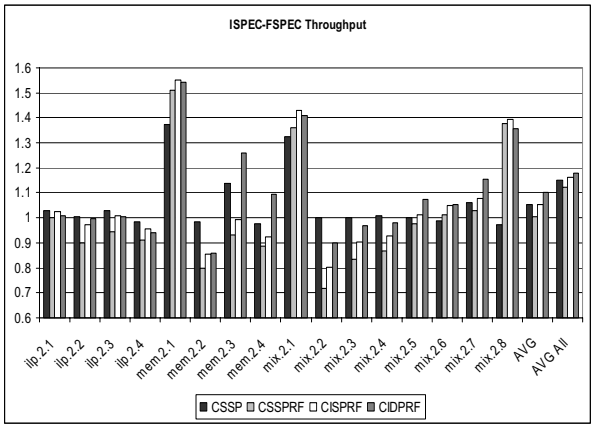

Figure 9. CDPRF performance for ISPEC-FSPEC category.

Finally, our last evaluation compares the fairness of all techniques by using the metric proposed in [17] in order to understand whether this performance improvement is equally distributed among the running threads.

Figure 10 shows the fairness speedup reached by the most representative techniques studied in this paper compared to *Icount*. Fairness can be defined as the minimum ratio between the slowdowns of any two threads running in the system compared to their single threaded execution as defined in [33]. Then, a fairness speedup close to 1 shows that the additional performance is equally distributed among the running threads. By contrast, when fairness speedup is lower than 1 it shows fairness degradation compared to the baseline (not all threads in the workload are taking advantage of the performance improvement). Finally, a speedup greater than 1 represents an overall fairness improvement.

As it can be seen, categories that execute different kinds of applications like *mixes* show fairness improvement. This is reasonable because unfair

situations can be reached with ease especially in the issue queues when the running applications have very different characteristics. Note also that all techniques outperform *Icount* fairness in 13%, 14% and 24% for *Stall*, *Flush+* and *CDPRF* respectively. However, even though *Flush+* has fairness improvement compared to *Icoun*t, flushing the threads that miss in cache extremely penalizes a thread in favor of the other negatively affecting the fairness equation. By contrast, *CDPRF* is very careful penalizing threads and therefore it gets stable performance improvements and also reaches a fair situation 24% better than *Icount*.

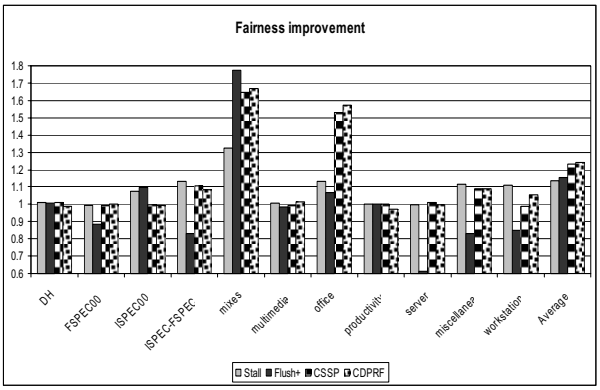

Figure 10. Fairness improvement compared to Icount.

6. Conclusion

Resource assignment schemes designed for monolithic processors are not adequate for clustered machines because they do not take into account important factors like inter-cluster communication or workload balance. The importance of considering these additional factors when designing a clustered SMT processor has been evaluated. Workload balance has been shown as the main factor to be considered by the resource assigment policy while inter-cluster communication has shown a poor impact because it is hidden by the multithreaded execution. Finally, we have motivated that resource assignment schemes that control the issue queue should be clustered sensitive in order to avoid a thread to take control of a whole cluster. By contrast, our results show that physical register files should be handled in a cluster-insensitive fashion in order to avoid conflicting decisions with the issue queue control logic. Hence, while statically partition the issue queues is a reasonable simple solution, using the same technique for the register file incurs in performance slowdown for some workloads because of resource underutilization. Therefore, an enhanced version that dynamically partitions the register files has been proposed in order to maximize its utilization. The final proposal reached a 17.6% average performance speedup compared to *Icount* and up to 40% average speedup for certain categories. Finally, we have observed that *CDPRF* is not only effective by improving performance but also do it equally among threads outperforming *Icount* fairness in 24%. Hence, using these conclusions in order to adapt more sophisticated schemes like [20][30] and [32] to make them fit in a clustered processor is part of our future work.

7. Acknowledgments

This work has been partially supported by the Spanish Ministry of Education and Science under grants TIN2004-03702 and TIN2007-61763 and Feder Funds.

8. References

[1] D. Tullsen, S.J. Eggers, and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, In Proceedings of the22th Annual International Symposium on Computer Architecture, 1995.

[2] W. Yamamoto, and M. Nemirovsky,“Increasing superscalar performance through multistreaming”, In Proceedings of PACT, Jun, 1995.

[3] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm,“Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”, In proc. of 23nd Annual International Symposium on Computer Architecture, 1996.

[4] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic,“The Multicluster Architecture: Reducing Cycle Time through Partitioning”, In Proceedings of MICRO-30. 1997.

[5] S. Palacharla, “Complexity-Effective Superscalar Processors”, Ph.D. thesis, Univ. of Winsconsin-Madison, 1998.

[6] V. Krishnan, and J. Torrellas, “A Clustered Approach to Multithreaded Processors”, In International Parallel Processing Symposium, 1998.

[7] D. Albonesi, “Dynamic IPC/clock rate optimization”, Proceedings of ISCA-25, June 1998.

[8] R. Kessler, “The Alpha 21264 Microprocessor”, IEEE Micro, 19(2):24-36, March/April 1999.

[9] M.J. Flynn, P. Hung, and K. Rudd,"Deep-Submicron Microprocessor Design Issues", IEEE Micro, 19(4): 11-22, July/August 1999.

[10] V.V. Zyuban, "Low-Power High-Performance Superscalar Architectures", PhD Thesis, Dept. of Computer Science and Engineering, University of Notre Dame, Jan. 2000.

[11] A. Snavely, and D. Tullsen,"Symbiotic Jobscheduling for a Simultaneous Multithreading Processor", In proceedings of Architectural Support for Programming Languages and Operating Systems, 2000.

[12] R. Canal, J.M. Parcerisa, and A. González, "Dynamic Cluster Assignment Mechanisms", In proceedings of International Symposium on High Performance Computer Architectures, 2000.

[13] A. Baniasadi, and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors", In Proc. of the 33rd. Ann. Intl. Symp. On Microarchitecture, pp. 337-347, December 2000.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", Intel Technology Journal, February 2001.

[15] R. Iris Bahar, and S. Manne, "Power and Energy Reduction Via Pipeline Balancing", In Proceedings of ISCA-28, July 2001.

[16] A. Aggarwal, and M. Franklin,"An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors", In Proceedings of ISPASS, 2001.

[17] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in SMT processors", In Proc. of the International Symposium on Performance Analysis of Systems and Software, pages 164–171, 2001.

[18] Y. Sazeides, and T. Juan, "How to Compare the Performance of Two SMT Microarchitectures", In Proc. of ISPASS 2001.

[19] Dean M. Tullsen, and Jeffery A. Brown, "Handling long-latency loads in a simultaneous multithreading processor", In Proc. of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001.

[20] A. El-Moursy, and D. H. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors", In Proceedings of the 9th International Conference on High Performance Computer Architecture, February 2003.

[21] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors", In Proceedings of the ISCA-30, June 2003.

[22] R. Bhargava, and L. John Friendly, "Improving dynamic cluster assignment for clustered trace cache processors", In Proceedings of the ISCA-30, June 2003.

[23] P. Racunas, and Yale N. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitectures", In proceedings of ICS, 2003.

[24] S. E. Raasch, and S. K. Reinhardt, "The impact of resource partitioning on SMT processors", In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, Sept. 2003.

[25] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "Improving Memory Latency Aware Fetch Policies for SMT Processors", Proc. Fifth Int'l Symp. High Performance Computing (ISHPC), Oct. 2003.

[26] D. Koufati, and D.T. Marr, "Hyperthreading technology in the netburst microarchitecture", Appears in IEEE Micro, Vol. 23 Issue 2 page(s) 56-65; March-April 2003.

[27] B. Sinharoy, "POWER5 Architecture and Systems", Keynote presentation, International Symposium on High Performance Computer Architecture, Feb. 2004.

[28] F. Latorre, J. González, and A. González, "Back-end Assignment Schemes for Clustered Multithreaded Processors", In Proceedings of ICS, 2004.

[29] J. D. Collins, and D. M. Tullsen, "Clustered Multithreaded Architectures -- Pursuing Both IPC and Cycle Time", In proc. of 18th IPDPS, April, 2004.

[30] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically Controlled Resource Allocation in SMT Processors", In Proceedings of the 37th International Symposium on Microarchitecture, pages 171–182. IEEE Computer Society, December 2004.

[31] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Partitioning Multi-Threaded Processors with a Large Number of Threads", In proc. of ISPASS 2005.

[32] S. Choi, and D. Yeung, "Learning-Based SMT Processor Resource Distribution via Hill-Climbing", In proc. of the 33rd. International Symposium on Computer Architecture, June 2006.

[33] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and Throughput in Switch on Event Multithreading", In proc. of the 39th International Symposium on Micro-architecture. Dec, 2006.