# Selection of the Register File Size and the Resource Allocation Policy on SMT Processors

Jesús Alastruey[1], Teresa Monreal[1], Francisco Cazorla[2], Víctor Viñals[1] and Mateo Valero[2,3]
[1]DIIS-I3A, Universidad de Zaragoza, Spain. {jalastru, tmonreal, victor}@unizar.es
[2]Barcelona Supercomputing Center, Spain. francisco.cazorla@bsc.es
[3]DAC, Universitat Politècnica de Cataluña, Spain. mateo@ac.upc.edu

## Abstract

*The performance impact of the Physical Register File (PRF) size on Simultaneous Multithreading processors has not been extensively studied in spite of being a critical shared resource. In this paper we analyze the effect on performance of the PRF size for a broad set of resource allocation policies (Icount, Stall, Flush, Flush++, Static, Dcra and Hill-climbing) and evaluate them under two metrics: instructions per second (IPS) for throughput and harmonic mean of weighted IPCs (Hmean-wIPC) for fairness. We have found that resource allocation policy and PRF size should be considered together in order to obtain the best score in the proposed metrics. For instance, for the analyzed 2 and 4-threaded SPEC CPU2000 workloads, small PRFs are best managed by Flush, whereas for larger PRFs, Hill-climbing and Static lead to the best values for the throughput and fairness metrics, respectively. The second contribution of this work is a simple procedure that, for a given resource allocation policy, selects the PRF size that maximizes IPS and obtains for Hmean-wIPC a value close to its maximum. According to our results, Hill-climbing with a 320-entry PRF achieves the best figures for 2-threaded workloads. When executing 4-threaded workloads, Hill-Climbing with a 384-entry PRF achieves the best throughput whereas Static obtains the best throughput-fairness balance.*

## 1. Introduction

Simultaneous Multithreading (SMT) processors extend the superscalar execution allowing the issue of instructions coming from different threads in the same cycle. In modern out-of-order SMT processors, this execution model implies the sharing of several resources among the threads, for instance, the Physical Register File (PRF) [6][7]. Therefore, the PRF must be big enough to store the committed and speculative states of all the threads and it must supply every cycle all the operands required by the instructions of the different executing threads. Consequently, SMT processors require large and highly-ported PRFs with high power consumption, area and access time. This last fact may affect the processor's cycle time, limiting its frequency and reducing its performance.

Hence, there exists a trade-off for the processor designer according to the PRF sizing. On the one hand, large PRFs can reduce the number of rename stalls due to the lack of physical registers, improving performance measured in instructions per cycle (IPC). On the other hand, small PRFs may allow higher processor frequencies, leading to more performance measured in instructions per second (IPS).

Another important decision in the SMT processor design is the resource allocation policy, which determines the way processor shared resources are distributed among all running threads. Several policies have been proposed claiming benefits over previous existing ones, but the evaluation of these policies has been carried out for a fixed PRF size [1][2][4][5][11][14][15][16].

The main contributions of this paper are:

- A combined analysis of the performance impact of the PRF size and the resource allocation policy. We consider both a large range of PRF sizes and a broad set of allocation policies (*Icount* [16], *Stall*, *Flush* [15], *Flush++* [2], static [7][11], *Dcra* [1], and *Hill-climbing* [4]). We have found that relative performance among policies strongly depends on the PRF size. For instance, *Flush* is the best policy to manage small PRFs but the worst with large PRFs.
- A simple procedure that, for a given allocation policy, selects a PRF size according to a performance trade-off between throughput and fairness. Thus, we consider two metrics: instructions per second (IPS) to measure throughput while taking into account the effect of accessing PRFs of different sizes, and the harmonic mean of weighted IPCs (Hmean-wIPC) to quantify fairness in the distribution of resources among all running threads [9]. We have applied this procedure to several allocation policies and found that in all cases, there exists a PRF size that maximizes IPS while reaching values close to the maximum of Hmean-wIPC. Finally, for each allocation policy and PRF size pair, we collect their throughput, fairness and energy efficiency values so that to evaluate the best design point according to a particular target.

The rest of the paper is structured as follows. In Section 2 resource allocation policies are classified and described. Section 3 details the experimental framework and methodology. Section 4 discusses results and Section 5 concludes the paper.

## 2. Resource allocation policies

Several SMT processor resources, like the Issue Queue or the PRF, are usually shared by the different executing threads. The distribution of these resources can be indirectly controlled through the instruction fetch policy. This way, resource allocation is driven by the distribution of the fetch bandwidth among the threads.

1550-6533/08 $25.00 © 2008 IEEE
DOI 10.1109/SBAC-PAD.2008.17

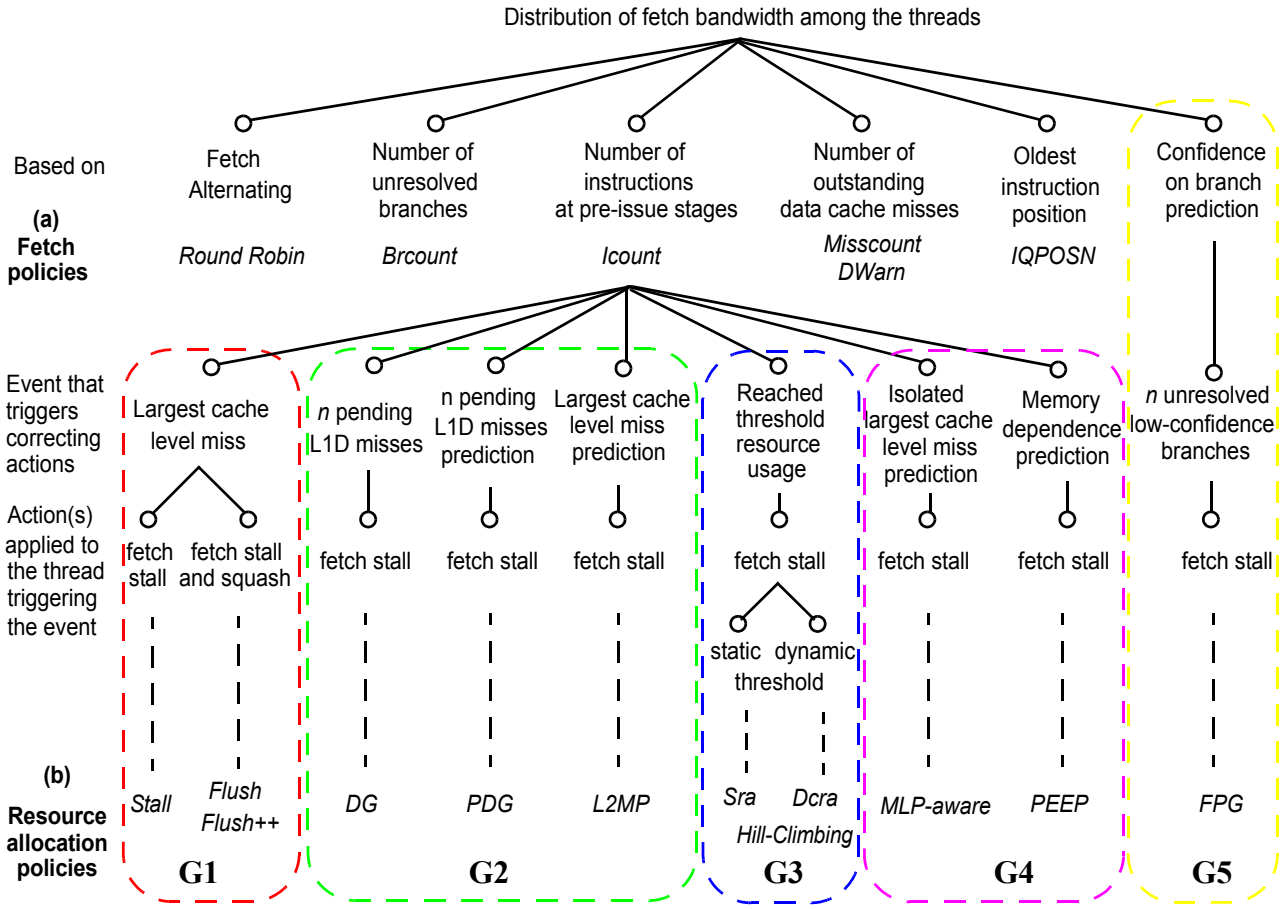


**Figure 1. Classification of fetch and resource allocation policies.**

Figure 1a shows different fetch policies. Except *Round Robin*, that equally shares the fetch unit among all threads, all the schemes give priorities to the threads according to different heuristics such as their number of unresolved branches (*Brcount*), pending data cache misses (*Misscount*), L1-Dcache outstanding misses (*DWarn*), instructions in the pre-issue stages (*Icount*) or the age of their oldest instructions (*IQPOSN*) [3][16]. Among all these fetch policies, Icount is considered the one that achieves best performance and has been adopted as baseline of nearly all the academic proposals of allocation policies.

As stated before, the *Icount* fetch policy gives higher priority to the threads with fewer instructions in the front-end stages [16]. It obtains good results with high-ILP threads, however, when a thread suffers many long-latency loads (loads missing the largest cache level or data TLB), *Icount* keeps allocating resources to this stalled thread. As a result, it may hold many resources, clogging the pipeline and decreasing the processor throughput [6].

There exist more complex proposals that try to overcome the drawbacks of the fetch policies. Some of these schemes react when a thread experiences a long-latency operation, while others try to anticipate their reaction by way of prediction (for instance, predicting L2 cache misses). The goal in both cases is to distribute resources among the threads to maximize performance. We call resource allocation policies to those enhanced fetch policies. Figure 1b shows a classification of several academic and industry proposals. Even though all resource allocation policies can be based on any fetch policy, all but one (*FPG*) are built on top of *Icount* because of its best performance.

A first group of policies (G1) deal with the resource starvation produced by memory-bound threads by undertaking certain actions when a long-latency load is detected. *Stall* is an *Icount* improvement that prevents threads with a pending long-latency load from instruction fetching [15]. Nevertheless, when a long-latency load is detected, it may be too late to prevent a thread from allocating resources. On the contrary, a thread may be stalled although there exists many free resources, arising resource underuse and maybe avoiding exploitation of memory level parallelism.

*Flush* is an extension of *Stall* that squashes a thread when it experiences a long-latency load. All the instructions younger than the offending load are squashed in order to release their allocated resources [15]. Nevertheless, these released resources may not be required by the other threads. Furthermore, power consumption increases as a result of the squashed instructions re-execution.

*Flush++* is based on the fact that *Stall* performs better than *Flush* for workloads that do not require many resources (few long-latency loads) and that *Flush* per-

forms better than *Stall* for workloads that require many resources (many long-latency loads) [2]. Thus, *Flush++* tries to combine the best of *Flush* and *Stall* to increase performance. The number of stalled threads is used to switch between *Stall* and *Flush*. A thread suffering a long-latency load is only squashed if it is the only non-stalled thread.

A second group of proposals (G2) try to anticipate long-latency loads by different approaches. *Data Miss Gating* (*DG*) stalls threads with frequent L1 data cache misses (*n* pending misses) [10]. This way, resources can be released shortly after being allocated avoiding resource clogging. In a similar way, *Predictive Data Miss Gating* (*PDG*) stalls a thread with more than *n* outstanding data cache misses, either counted or predicted [10]. Since miss prediction is performed when a load is fetched, threads can be stalled before than in *DG*. The main goal of these policies is to reduce the issue queue size occupation, not to improve performance. Hence, they can degrade performance of memory-bounded threads. Finally, *L2MP* uses a load miss predictor to detect largest cache level misses [18]. Threads with predicted long-latency loads are stalled until they are resolved [2].

A third group of policies (G3) explicitly controls resource distribution by setting a threshold that limits the maximum resource usage of a thread. Hence, a thread is prevented from fetching when it consumes all the resources of its share. These strategies can be classified as static or dynamic depending on the share being fixed of variable during execution.

*Static resource allocation* (*Sra*) assigns static resource shares to each of the *n* executing threads. These quotas can range between one nth of the resource (no sharing, partitioning) and all the resources (full sharing) [11]. For example, Intel Pentium 4 partitions the fetch, micro-op and retire queues, uses a threshold sharing scheme for the schedulers (IQs) and fully shares its caches [7]. Power5 microprocessor uses a threshold sharing policy for the Global Completion Table and the Load Miss Queue. This resource-balancing logic also detects a thread reaching a threshold of L2 cache and TLB misses [6].

*Dynamically Controlled Resource Allocation* (*Dcra*) dynamically distributes processor resources among threads depending on their cache behaviour: threads with frequent L1-Dcache misses (likely to suffer long-latency misses) are assigned a limited share so as to prevent stalled threads from clogging resources [1]. This threshold is greater than a proportional share so these threads are allowed to exploit parallelism beyond long-latency loads. Moreover, *Dcra* computes partitions for each thread based on their resource needs, allowing a thread to borrow resources from threads that do not require them.

*Hill-climbing* evaluates different resource distributions trying to find the partition that optimizes performance [4]. It uses a learning algorithm that dynamically selects the partitions that achieved the best runtime performance in the last trial period. Performance monitoring is carried out continuously, so *Hill-climbing* is able to respond to the different resource needs of the workload over time.

Two recent policies exploit memory-related predictions (G4). *Memory-Level Parallelism aware* (*MLP-aware*) tries to overlap the execution of independent long-latency loads, thus hiding memory access penalties [5]. It relies on two mechanisms: one detects or predicts long-latency loads and the other tries to determine whether such offending loads are isolated or not (MLP prediction). In the latter case, the predictor must also determine the number of additional instructions that should be fetched to overlap multiple long-latency loads. A thread suffering an isolated long-latency load is stalled, and could be even flushed. Compared with *Icount*, this policy is better when executing MLP intensive workloads but it degrades performance when executing ilp-intensive workloads (more than 20% of slowdown in some cases). This policy relies on the relatively hard load miss and MLP predictions. In a different approach, a later proposal exploits the high predictability of memory dependencies. *Proactive Exclusion-Early Parole* (*PEEP*) stalls a thread when a fetched load is predicted to have a dependence with a previous store [14]. Another prediction mechanism tries to restart the stalled thread so that to resume its execution as soon as the dependence has been resolved.

Finally (G5), *Fetch Prioritization and Gate* (*FPG*) sets up priority to a thread according to its number of unresolved low-confidence branches [8]. Besides, a thread is excluded from fetching when it reaches a threshold of pending low-confidence branches. This is the only described policy not built on top of *Icount*.

Unfortunately, none of the previously reported references performs a comprehensive evaluation of PRF size, using most of times a fixed PRF size. We have evaluated the performance of some of these policies with a large range of PRFs sizes with two goals in mind: analyze the joint effect on performance of PRF size and resource allocation policy, and find out the best PRF size that optimizes performance for a given allocation policy. On the one hand, this work warns the processor designer that resource allocation policy and PRF size should be considered together to maximize performance. For instance, the best policy for a certain PRF size could be the worst for a different PRF size. On the other hand, we propose a simple procedure that, for a given allocation policy, selects the PRF size that achieves the best throughput-fairness balance. Since we have applied it to a broad range of resource allocation policies, it could also assist an SMT designer to find a suited allocation policy for a given PRF size.

It is not our aim to determine which resource allocation policy achieves the best performance for each PRF size. Some policies have design options that greatly affect their performance (for instance, predictor sizing and training). Thus, an accurate performance comparison would require fine tuning of many parameters for a lot of resource allocation policies. Moreover, it would require a comprehensive evaluation with several processor configurations and different kinds of workloads. That goal is out of the scope of this paper. Hence, we have focused on the classical (G1) and threshold-based resource allocation policies (G3) to give empirical evidence of our contributions.

# 3. Experimentation

## 3.1 Processor model

We use a detailed cycle-based simulator consisting of a trace-driven front-end and an improved version of smtsim back-end [17]. A basic-block dictionary that contains all the static instructions allows mispredicted-path execution.

We model a wide-issue SMT processor with an 11-stage pipeline (see Figure 2). The details of the simulated

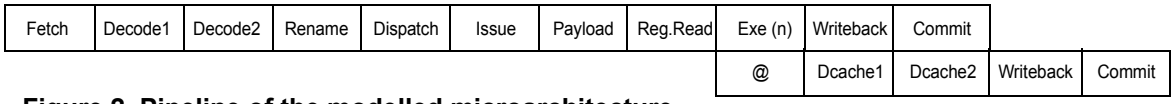


**Figure 2. Pipeline of the modelled microarchitecture.**

**Table 1. Parameters of the simulated SMT processor.**

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Number of threads | 2 and 4 | L1 Icache | 64 KB, 4-way, 4 banks, 64 byte lines, latency: 1 cycle |
| Fetch width | 8 instructions (max: 2 threads) | L1 Dcache | 64 KB, 8-way, 4 banks, 64 byte lines, latency: 2 cycles |
| Decode and commit width | 8 instructions | L2 Ucache | 2048 KB, 8-way, 8 banks, 64 byte lines, latency: 15 cycles |
| Issue width | 6 int + 3 fp+ 4 mem | Main memory | Access time: 300 cycles |
| Reorder Buffer (ROB) | 512 entries, shared, per-thread commit | TLB | ITLB: 64 entries, DTLB: 128 entries, miss penalty: 360 cycles |
| Issue Queue sizes (IQ) | int: 80 entries, fp: 80 entries mem: 80 entries | Functional Units (*latency*) | 6 int (*simple: 1, mult/div: 4*), 4 load/store (*2*), 3 fp(*simple/cmp/mult: 4,div: 17, sqrt: 19*) |
| Branch prediction | gshare 16K entries BTB: 256, 4-way, RAS: 256 | PRF | int: 80-640 entries, 16r+10w ports fp: 80-640 entries, 6r+7w ports, access time: 1 cycle |

microarchitecture are shown in Table 1. Issue queues, issue width and ROB have been oversized so as to adapt to the biggest PRF sizes. This way, the effect of the PRF size variation can be better analyzed. A bypass network provides operands to dependent instructions issued back-to-back avoiding PRF reads. The L1-Dcache is made up of four 16KB line-interleaved banks whose conflicts are properly modelled and penalized.

## 3.2 Implementation details of evaluated resource allocation policies

The main parameters used in the implementation of the considered allocation policies are shown in Table 2.

With respect to the G1 group, we have implemented *Stall*, *Flush* and *Flush++* with an improvement called *Continue the Oldest Thread* (*COT*) [2]. When all the threads are stalled due to long-latency misses, the thread with the oldest offending load is allowed to continue fetching. The idea is to advance the execution of the thread most likely to be the first that resolves its cache miss.

Related to the policies that explicitly control resource allocation (G3), all have been built on top of *Icount*. Our *Dcra* implementation distributes the Issue Queue entries (integer, floating point and memory) and the integer and floating point rename registers, whereas our *Sra* and *Hill-climbing* implementations distribute the integer Issue Queue entries, the integer rename registers and the Reorder Buffer. Per-thread caps of *Sra* correspond to hard partitioning (no resource sharing) for 2-threads and very limited sharing for 4-threads. According to [11], these thresholds provide good performance and enforce fairness. We have explored other resource limits and have not found better throughput and fairness figures. We choose IPC to direct SMT performance feedback to the *Hill-climbing* algorithm (named HILL-IPC in [4]) because it allows the most complexity-effective implementation. All other metrics considered in [4] need to evaluate the standalone IPC of each thread, so they have to be periodically executed alone during certain periods of time, whereas IPC measurement can be done without disabling SMT execution. In order to adapt its value to a wide range of PRF sizes, the Delta parameter (constant value of 4 in [4]), has been replaced by a value proportional to the number of rename registers. Finally, the minimal amount of shared resources assigned to each thread has been set to Delta.

## 3.3 Workload

All the benchmarks of SPEC2000 have been used but *fma3d* which could not be executed within our framework. Traces of 300-millions of instructions have been built according to the ideas introduced in [13]. Benchmarks have been categorized into *high-ILP* and *low-ILP* according to their single-threaded IPC. This way, three different workloads types can be distinguished: *high-ILP* (all threads are high-ILP), *low-ILP* (all threads are low-ILP) and *mix* (composed by high-ILP and low-ILP threads). For each workload type, eight combinations of two benchmarks and four combinations of four benchmarks have been selected, see Table 3. We have tried to equally include all the benchmarks and to balance integer and floating point combinations. Simulations end when all the threads have finalized its execution at least one time (*last* methodology [19]). When a thread finalizes before the end of the simulation, it is re-executed. This way, the workload is composed always by the specified number of threads.

## 3.4 Metrics

We consider two metrics to evaluate performance. IPS considers the effect of the different PRF access times on performance. Hmean-wIPC has been widely used to characterize SMT performance [1][2][3][4][5][9][14].

- IPS: it quantifies the performance (throughput) obtained by an SMT microarchitecture. It is the ratio between the IPC and the cycle time of the processor ($T_c$). IPC is obtained as the addition of the multi-

**Table 2. Main parameters of the implemented resource allocation policies.**

| Policy | Parameter | Value |
|---|---|---|
| Stall, Flush, Flush++ | L2 delay to suppose a miss | 20 cycles |
| Sra | Per-thread caps [11] | 2 threads: 50%, 4 threads: 30% |
| Hill-climbing | Epoch Size | 64k cycles |
| | Delta | #rename registers / 64 |

**Table 3. Multithreaded workloads.**

| | | Workload type | | |
|---|---|---|---|---|
| | | **high-ILP** | **low-ILP** | **mix** |
| **2 threads** | **int** | bzip2-eon, gzip-gcc | vpr-mcf, vortex-twolf | perlbmk-vortex, gcc-gap |
| | **int-fp** | perlbmk-apsi, crafty-galgel | gap-swim, parser-mgrid | crafty-art, gzip-mgrid |
| | | bzip2-mesa, eon-sixtrack | vpr-lucas, mcf-equake | twolf-galgel, parser-ammp |
| | **fp** | mesa-sixtrack, ammp-wupwise | lucas-equake, applu-art | apsi-applu, facerec-swim |
| **4 threads** | **int** | bzip2-eon-gzip-gcc | vpr-mcf-vortex-twolf | perlbmk-vortex-gcc-gap |
| | **int-fp** | perlbmk-apsi-crafty-galgel | gap-swim-parser-mgrid | crafty-art-gzip-mgrid |
| | | bzip2-mesa-eon-sixtrack | vpr-lucas-mcf-equake | twolf-galgel-parser-ammp |
| | **fp** | mesa-sixtrack-ammp-wupwise | lucas-equake-applu-art | apsi-applu-facerec-swim |

threaded IPC reached by each thread ($IPC_i$) divided by the number of threads (N). We assume that the processor cycle time is constrained by the PRF access time, computed for each PRF size according to the Rixner model for 0.18μ [12].

$$IPS = \frac{IPC}{T_c} = \frac{\frac{\sum_i IPC_i}{N}}{T_c}$$

- Hmean-wIPC: it considers the weighted IPCs of each thread ($wIPC_i$), that is, the ratio between multi-threaded and single-threaded IPCs ($IPCst_i$), and averages them through the harmonic mean. It combines performance and fairness in the allocation of resources to the threads [9], so it penalizes policies that obtain high performance speeding up high-ILP threads at the expense of slowing down low-ILP threads. This metric does not depend on the processor cycle time and it could also be stated as the harmonic mean of the weighted instructions per second (wIPS) of each thread.

$$Hmean\_wIPC = \frac{N}{\sum_i (wIPC_i)^{-1}} = \frac{N}{\sum_i \frac{IPCst_i}{IPC_i}} =$$

$$= \frac{N}{\sum_i \frac{IPSst_i}{IPS_i}} = \frac{N}{\sum_i (wIPS_i)^{-1}} = Hmean\_wIPS$$

# 4. Results

Figure 3 shows IPS and Hmean-wIPC for all the 2-threaded workloads (left column) and all the 4-threaded workloads (right column). In order to make the figures clearer, we have not plotted the *Flush++* lines (they are similar to the *Flush* ones).

For each metric, we first describe its behaviour when PRF size (equally-sized integer and floating point register files) varies. Then we compare the performance obtained by the considered resource allocation policies. Finally, we compare the figures for 2 and 4-threaded workloads.

## 4.1 IPS

In order to evaluate this metric we considered that the PRF access is in the processor critical path, so that the PRF access time limits the processor frequency.

The upper graphs of Figure 3 show that the smallest PRFs obtain low performance. In spite of their access time allowing high processor frequencies, the effect of the large number of rename stalls is more pronounced. On the other hand, the high access time of the largest PRFs limits the processor frequency, so affecting performance. An interesting design point is the PRF size that maximizes IPS. This size varies with the resource allocation policy: for 2-threaded workloads, maximum IPS is reached between 224 and 320 registers, meanwhile for 4-threaded workloads it is reached between 320 and 448 registers. Pressure on resources is higher in the latter case, so all polices benefit from bigger PRFs.

If we compare resource allocation policies, we can observe that for 2-threaded workloads, *Flush* obtains the best performance with small PRFs (below 160 registers). This is because it avoids rename stalls by squashing stalled threads so as their allocated registers can be assigned to other active threads. For larger PRFs, *Hill-climbing* outperforms the rest of the policies: its adaptative resource allocation algorithm is able to find the best resource distribution. Pipeline squashes of *Flush* are less effective due to the reduction of register stalls. *Icount* (the simplest policy) becomes the worst choice for all sizes.

When executing 4-threaded workloads, PRFs below 320 registers reach the best IPS with *Flush.* Pressure on

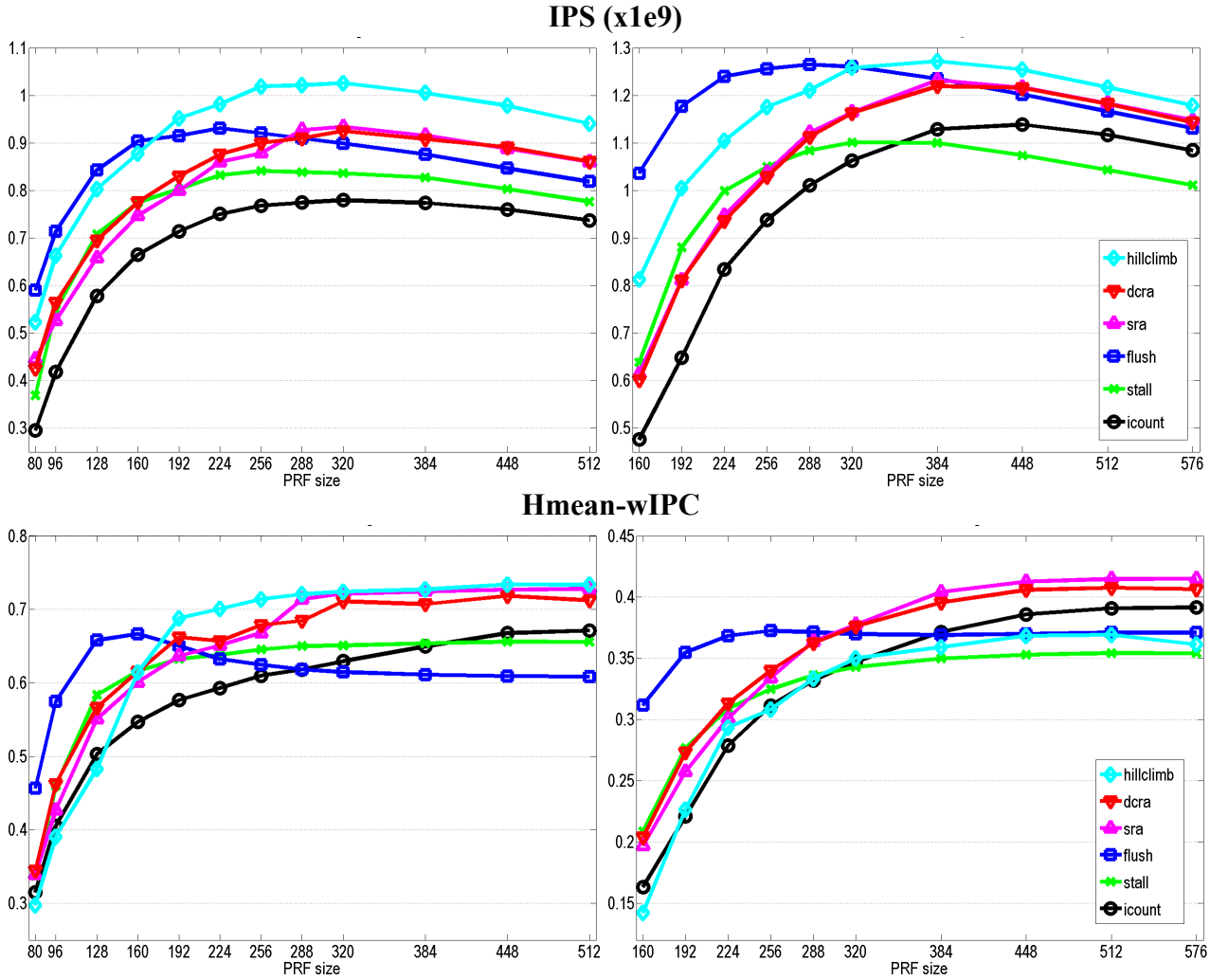


**Figure 3. IPS (top) and Hmean-wIPC (bottom) vs. PRF size for *Icount*, *Stall*, *Flush*, *Sra*, *Dcra* and *Hill-climbing* policies. The two figures on the left and the two ones on the right correspond to 2 and 4-threaded workloads, respectively. Note the different scales of both x and y axes for the 2 and 4-thread figures.**

resources is higher than in 2-threaded mode, so this policy is now more effective up to significative bigger PRFs. Beyond that amount of registers, *Hill-climbing* obtains the best results, but with smaller speedups respect to the rest of the policies than with the 2-threaded workloads. *Icount* is the worst choice for small PRFs but it is the policy that keeps improving its performance up to the biggest PRFs size (448 registers), overcoming *Stall* above 320 registers and achieving IPS figures close to the rest of policies.

It is interesting to point that in spite of its different threshold settings (dynamic and static), *Dcra* and *Sra* behaves in a very similar way for both 2 and 4-threaded workloads and for all register file sizes.

Finally, we compare the throughput obtained by a given policy for 2 and 4-threaded workloads. If we are interested in the maximum IPS point, in all cases it is reached with 4-threaded workloads but at the expense of bigger PRFs. If we compare the throughput obtained by a given PRF size, it can be seen that at least 224 registers are needed to get throughput speed-up when executing 4-threads instead of 2-threaded workloads. With smaller PRFs, *Flush* is the only policy that exploits thread-level parallelism.

## 4.2 Hmean-wIPC

This metric is low with small PRFs and increases to saturation with larger PRFs, except for *Flush*, that reaches its maximum value with much less registers (160 and 256 registers for 2 and 4-threaded workloads, respectively).

When executing 2-threaded workloads, tight PRFs (up to 160 registers) are best managed with *Flush*. Likewise IPS, it is able to reduce register rename stalls by way of releasing resources allocated to stalled threads. Medium-sized PRFs (up to 288 registers) obtain the best Hmean-wIPC with *Hill-climbing* and larger PRFs reach the best fairness figures with policies of the G3 group, with *Hill-climbing* slightly outperforming *Sra* and *Dcra*.

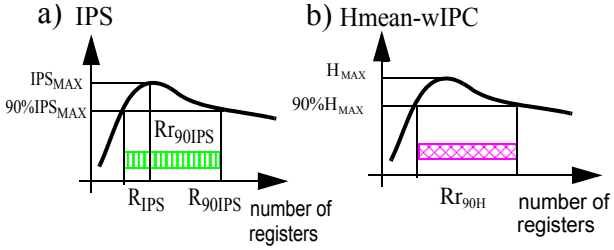


**Figure 4. Ranges $Rr_{90IPS}$, $Rr_{90H}$ and parameters $R_{IPS}$ and $R_{90IPS}$ (a, b).**

For 4-threaded workloads, *Flush* is again the best policy if the PRF is small or medium sized (up to 288 registers), whereas *Dcra* and *Sra* are the best schemes with large PRFs. In this case, *Hill Climbing* may be affected by the use of IPC as the performance feedback metric for its learning algorithm. If the goal is to maximize fairness, Hmean-wIPC should be used to direct SMT performance feedback (named HILL-HWIPC in [4]).

Finally, Hmean-wIPC figures are much higher for 2-threaded workloads. It provides up to a 100% speed-up relative to the 4-threaded workload in the case of *Hill-climbing*. If a design needs to enforce fairness, it has to be taken into account that, when executing 2 threads, any of our evaluated PRF sizes obtains Hmean-wIPC values that are unreachable when 4-threads are executed.

### 4.3 PRF sizing and resource allocation policy selection

In the previous subsections we have observed that the PRF size and the resource allocation policy should not be selected in an isolated way. Now, our aim is to select both the PRF size and the resource allocation policy that maximizes processor performance and fairness. The selection procedure is carried out in two steps. First, for each resource allocation policy, we look for a PRF size that balances the two metrics considered in this paper (IPS and Hmean-wIPC). This analysis simplifies the throughput-fairness comparison among different allocation policies. Secondly, we compare the selected design points using performance and energy efficiency metrics.

According to our results, Hmean-wIPC increases with the number of registers saturating at sizes close to the ones that achieve the peak value of IPS. Such sensitivities to the PRF size suggest that IPS could be maximized while obtaining a Hmean-wIPC value close to its maximum.

Thus, for a given allocation policy, our procedure considers the PRF sizes that obtain at least the 90% of the two metrics maximum values (see Figure 4):

- $Rr_{90IPS}$: range of registers needed to obtain 90% of the maximum IPS. We also define $R_{90IPS}$ as the maximum value in that range and $R_{IPS}$ as the number of registers needed to reach the maximum IPS.
- $Rr_{90H}$: range of registers needed to obtain 90% of the maximum Hmean-wIPC.

Table 4 collects these ranges and $R_{IPS}$ for each resource allocation policy and for both 2 and 4-threaded workloads. It can be seen that in all cases, the two ranges overlap and that $R_{IPS}$ lies in that intersection.

To summarize, for each resource allocation policy, the procedure searches for a PRF size that obtains simultaneously at least 90% of the maximum IPS and Hmean-wIPC. For all the analyzed policies, we have found several PRF sizes meeting these requirements. Among those design points and with performance as main target, we have selected the PRF size that maximizes IPS. These sizes are highlighted in Table 4 with shadowed background. The procedure can be easily modified to meet other design goals. For example, if hardware costs were the main target, we could select the smallest PRF size that lies in the $Rr_{90IPS}$ and $Rr_{90H}$ intersection.

Finally, for these selected PRF size and allocation policy pairs, Table 5 collects IPS and Hmean-wIPC (best scores are highlighted). It also collects dynamic energy per instruction (EpI) consumed by the PRF as a first-order approximation for energy efficiency. Total energy has been computed by multiplying the number of PRF accesses by the energy per each access, obtained for each PRF size according to the Rixner model [12]. EpI is used because instruction counts executed by each policy may be very different. We find that the EpI values of the best performance design pairs are very similar. We have also observed that *Flush* does not take advantage of its smaller PRF sizes due to the energy cost of reexecuting squashed instructions.

*Hill-climbing* with a 320-entry PRF seems the best choice for executing 2-threaded workloads. If our target is a 4-threaded SMT processor, we could either optimize throughput or fairness. If we are more interested in throughput, the best option is *Hill-climbing* with a 384-entries because it obtains the best IPC and IPS figures. However, if we prioritize the performance-fairness trade-off, *Sra* and *Dcra* are the best policies.

## 5. Conclusions

This paper analyzes the combined impact of the PRF size and the resource allocation policy on SMT performance, showing that a decoupled analysis is not a right choice. We have explored a large range of PRF sizes and a broad set of allocation policies and have found that for small PRFs, *Flush* is the policy that obtains the best throughput (IPS) and fairness (Hmean-wIPC) figures. For larger PRFs, our results show that *Hill-climbing* always achieves the best throughput and fairness scores, except for processors supporting 4 threads that have to be executed in a balanced way (with Hmean-wIPC). In that case *Sra* and *Dcra* are the best policies.

Related to simultaneously achieving the best throughput and fairness, we propose a simple procedure to find the best PRF size for a given resource allocation policy. Applying such procedure to a broad set of allocation policies, our results show that it is possible to find a PRF size that maximizes IPS while reaching values close to the maximum of Hmean-wIPC. Among the selected design points, *Hill-climbing* with 320 registers is the best scheme for 2-threaded workloads. For 4-threaded workloads, 384 registers is the best choice, either with *Hill-climbing* (best throughput) or with *Sra* or *Dcra* (best fairness). This last design point provides an average IPS improvement of 23% with respect to the 2-threaded SMT processor, at the expense of a 17% increase in the PRF size.

**Table 4. For each allocation policy, ranges of registers that obtain 90% of the maximum IPS ($Rr_{90IPS}$), 90% of the maximum Hmean-wIPC ($Rr_{90H}$), and numbers of registers that obtain the maximum IPS ($R_{IPS}$).**

| | 2-threads | | | 4-threads | | |
|---|---|---|---|---|---|---|
| | $Rr_{90IPS}$ | $Rr_{90H}$ | $R_{IPS}$ | $Rr_{90IPS}$ | $Rr_{90H}$ | $R_{IPS}$ |
| Hill-climbing | 192-512 | 192+ | 320 | 256+ | 288+ | 384 |
| Dcra | 224-512 | 192+ | 320 | 288+ | 320+ | 384 |
| Sra | 224-512 | 256+ | 320 | 288+ | 320+ | 384 |
| Flush | 128-448 | 128+ | 224 | 192-576 | 192+ | 288 |
| Stall | 160-512 | 160+ | 256 | 224-576 | 256+ | 320 |
| Icount | 192+ | 256+ | 320 | 320+ | 384+ | 448 |

**Table 5. For each allocation policy, PRF sizes (R) selected through the procedure described above and figures for those PRF sizes: IPS (billions), Hmean-wIPC and dynamic energy per instruction (EpI, measured in nJ) consumed by the PRF.**

| | 2-threads | | | | 4-threads | | | |
|---|---|---|---|---|---|---|---|---|
| | R | IPS | Hmean | EpI | R | IPS | Hmean | EpI |
| Hill-climbing | 320 | 1.03 | 0.72 | 2.5 | 384 | 1.27 | 0.36 | 2.7 |
| Dcra | 320 | 0.92 | 0.71 | 2.5 | 384 | 1.22 | 0.40 | 2.8 |
| Sra | 320 | 0.93 | 0.72 | 2.5 | 384 | 1.23 | 0.40 | 2.8 |
| Flush | 224 | 0.93 | 0.63 | 2.2 | 288 | 1.27 | 0.37 | 2.7 |
| Stall | 256 | 0.84 | 0.65 | 2.1 | 320 | 1.10 | 0.34 | 2.4 |
| Icount | 320 | 0.78 | 0.63 | 2.5 | 448 | 1.14 | 0.39 | 3.3 |

## 6. Acknowledgments

This work was supported in part by Diputación General de Aragón grant "gaZ: Grupo Consolidado de Investigación", Spanish Ministry of Education and Science grants TIN2007-66423 and TIN2007-60625, and European Union Network of Excellence HiPEAC-2 (High-Performance Embedded Architectures and Compilers, FP7/ICT 217068).

## 7. References

[1] F. J. Cazorla, A. Ramirez, M. Valero and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. *Int'l Symp. on Microarchitecture (MICRO)*, pp. 171–182. IEEE Computer Society, December 2004.

[2] F.Cazorla, E.Fernández, A.Ramírez and M.Valero. Improving Memory Latency aware fetch policies for SMT processors. *Int'l Symp. High Performance Computing*, Oct. 2003.

[3] F.Cazorla, A.Ramírez, M.Valero and E.Fernández. DCache Warn: an I-Fetch Policy to Increase SMT Efficiency. *IPDPS*, pp. 74-83, 2004,.

[4] S.Choi and D.Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. *Int'l Symp. on Computer Architecture (ISCA)*, 2006, pp. 239-251.

[5] S.Eyerman, L.Eccklhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors, *HPCA*, 2007, pp. 240-249.

[6] R.Kalla, B.Sinharoy and J.Tendler, IBM Power5 Chip: A Dual-Core Multithreaded Processor, *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 40-47.

[7] D.Koufaty, D.Marr, Hyperthreading technology in the netburst microarchitecture, *IEEE Micro,* vol 23, no 2, Mar.-Apr. 2003, pp.56-65.

[8] K.Luo, M.Franklin, S.Mukherjee and A.Seznec. Boosting SMT Performance by Speculation Control. *IPDPS*, pp. 9-16, April 2001.

[9] K.Luo, J.Gummaraju and M.Franklin. Balancing throughput and fairness in SMT processors. *ISPASS*, Nov. 2001.

[10] A.EI-Moursy and D.Albonesi. Front-End Policies for Improved Issue Efficiency in SMT processors. *HPCA*, pp. 31-40, February 2003.

[11] S.E.Raasch, S.K.Reinhardt, The impact of resource partitioning on SMT processors, *PACT*, pp. 15-25, Sept. 2003.

[12] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi and J.Owens, Register Organization for Media Processing. *HPCA*, pp. 375-386, Jan. 2000.

[13] T.Sherwood, E.Perelman, G.Hamerly and B.Calder. Automatically Characterizing Large Scale Program Behavior. *ASPLOS*, pp. 45-57, Oct. 2002.

[14] S.Subramaniam, M.Prvulovic and G.Loh. PEEP: Exploiting Predictability of Memory Dependences in SMT Processors. *HPCA*, pp. 137-148, Feb. 2008.

[15] D.Tullsen and J.Brown. Handling long-latency loads in a simultaneous multithreaded processor. *Int'l Symp. on Microarchitecture (MICRO)*, pp. 318–327. Dec. 2001.

[16] 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. *ISCA*, pp. 191–202, Apr. 1996.

[17] D.Tullsen, S. Eggers and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Int'l Symp. on Computer Architecture (ISCA)*, pp. 392-403, 1995

[18] A.Yoaz, M.Erez, R.Ronen and S.Jourdan. Speculation techniques for improving load related instruction scheduling. *ISCA*, May 1999.

[19] J.Vera, F.Cazorla, A.Pajuelo, O.Santana, E.Fernandez and M.Valero. FAME: FAirly MEasuring Multithreaded Architectures. *PACT*. September 2007.