Managing SMT Resource Usage through Speculative Instruction Window Weighting

Hans Vandierendonck, André Seznec

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Managing SMT Resource Usage through Speculative Instruction Window Weighting

Hans Vandierendonck — André Seznec

N° 7103

Novembre 2009

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7103--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Managing SMT Resource Usage through Speculative Instruction Window Weighting

Hans Vandierendonck* , André Seznec

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet ALF

Rapport de recherche n° 7103 — Novembre 2009 — 19 pages

Abstract: Simultaneous multithreading processors dynamically share processor resources between multiple threads. In general, shared SMT resources may be managed explicitly, e.g. by dynamically setting queue occupation bounds for each thread as in the DCRA and Hill-Climbing policies. Alternatively, resources may be managed implicitly, i.e. resource usage is controlled by placing the desired instruction mix in the resources. In this case, the main resource management tool is the instruction fetch policy which must predict the behavior of each thread (branch mispredictions, long-latency loads, etc.) as it fetches instructions.
In this paper, we present the use of Speculative Instruction Window Weighting (SIWW) to bridge the gap between implicit and explicit SMT fetch policies. SIWW estimates for each thread the amount of outstanding work in the processor pipeline. Fetch proceeds for the thread with the least amount of work left. SIWW policies are implicit as fetch proceeds for the thread with the least amount of work left. They are also explicit as maximum resource allocation can also be set. SIWW can use and combine virtually any of the indicators that were previously proposed for guiding the instruction fetch policy (number of in-flight instructions, number of low confidence branches, number of predicted cache misses, etc.). Therefore, SIWW is an *approach to designing SMT fetch policies*, rather than a particular fetch policy.

Targeting fairness or throughput is often contradictory and a SMT scheduling policy often optimizes only one performance metric at the sacrifice of the other metric. Our simulations show that the SIWW fetch policy can achieve at the same time state-of-the-art throughput, state-of-the-art fairness and state-of-the-art harmonic performance mean.

Key-words: Simultaneous multithreading, resource management, instruction fetch policy, branch confidence, hit/miss prediction

* Hans Vandierendonck is with University of Ghent

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Managing SMT Resource Usage through Speculative Instruction Window Weighting

Résumé : Les processeurs SMT partagent les ressources du processeur dynamiquement à l'exécution entre les différents processus.

Dans ce rapport, nous présentons une nouvelle politique d'allocation de ressources entre les processus qui permet à la fois un très haut débit et une distribution équitable des ressources.

Mots-clés : Multiflot excution simultanée

Managing SMT Resource Usage through Speculative Instruction Window Weighting

Hans Vandierendonck
Ghent University
Dept. of Electronics and Information Systems
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium
hvdieren@elis.ugent.be

André Seznec
IRISA/INRIA
Centre INRIA Rennes-Bretagne Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
seznec@irisa.fr

abstract

Simultaneous multithreading processors dynamically share processor resources between multiple threads. In general, shared SMT resources may be managed explicitly, e.g. by dynamically setting queue occupation bounds for each thread as in the DCRA and Hill-Climbing policies. Alternatively, resources may be managed implicitly, i.e. resource usage is controlled by placing the desired instruction mix in the resources. In this case, the main resource management tool is the instruction fetch policy which must predict the behavior of each thread (branch mispredictions, long-latency loads, etc.) as it fetches instructions.
In this paper, we present the use of Speculative Instruction Window Weighting (SIWW) to bridge the gap between implicit and explicit SMT fetch policies. SIWW estimates for each thread the amount of outstanding work in the processor pipeline. Fetch proceeds for the thread with the least amount of work left. SIWW policies are implicit as fetch proceeds for the thread with the least amount of work left. They are also explicit as maximum resource allocation can also be set. SIWW can use and combine virtually any of the indicators that were previously proposed for guiding the instruction fetch policy (number of in-flight instructions, number of low confidence branches, number of predicted cache misses, etc.). Therefore, SIWW is an approach to designing SMT fetch policies, *rather than a particular fetch policy.*
Targeting fairness or throughput is often contradictory and a SMT scheduling policy often optimizes only one performance metric at the sacrifice of the other metric. Our simulations show that the SIWW fetch policy can achieve at the same time state-of-the-art throughput, state-of-the-art fairness and state-of-the-art harmonic performance mean.

1 Motivation and Related Work

On simultaneous multi-threading (SMT) processors, several threads share the hardware resources of a wide-issue superscalar processor [13]. Balancing the resource allocation to the threads is a major issue on these SMT processors. The objective of resource allocation is to maximize throughput or fairness, or a combination thereof.

The instruction fetch engine was recognized very early as a natural resource usage controller in a SMT processor [22, 21]. In practice, the fetch policy indirectly controls the usage of all the processor resources. As such, it can be leveraged to avoid phenomena that can starve the concurrent threads, e.g. monopolization of instruction queues by a thread due to long-latency load misses in the last cache levels (FLUSH [20] and PDG [7]). Memory-level parallelism [8] and branch mispredictions [14, 12] can also be taken into account by the fetch policy.

Alternatively, processor resources can be allocated explicitly. The DCRA policy [2] estimates the resource needs of threads based on cache miss counts and by monitoring issue queue usage. Threads are assigned a particular portion of the processor resources based on their cache behavior and their need for specific queues. The Hill-Climbing policy [3] measures the system's performance and continuously changes the resource distribution. The Hill-Climbing policy can be set to optimize any performance metric.

There is a tension between these two approaches to SMT instruction fetch steering. On the one hand, the instruction fetch policy approach can take into account quasi instanstaneously the sources of execution roughness (e.g. a L2 cache miss). Therefore, these policies can make fetch steering decisions that are very accurate in particular scenarios. On the other hand, explicit resource allocation policies optimize resource distribution in the long term: They optimize overall performance by trying to globally assign the resources to the thread that makes the best use of them, but they fail to identify punctual incidents that leads to a waste of resources.

In this paper, we work towards a synergy between implicit and explicit SMT resource allocation policies. The key to this work is a new approach to constructing SMT fetch policies that model the amount of outstanding work in the processor pipeline for each thread. Long-latency loads, low-confidence branches, etc. are modelled as

requiring more work than regular instructions. Implicit resource allocation is performed by fetching instructions for the thread with the least amount of outstanding work. Explicit resource allocation is performed by fixing the maximum amount of outstanding work for each thread.

Our approach to constructing SMT fetch policies is based on Speculative Instruction Window Weighting (SIWW), a technique proposed to steer fetch gating for increasing energy-efficiency [23]. For fetch gating, SIWW estimates the amount of outstanding work in the processor pipeline. When this amount of work is large, then fetch may be gated until some of the work is performed, at which point fetch may continue to build up the outstanding work. In this paper, we show how to apply SIWW to SMT fetch policies and how it can be used to explicitly allocate resources.

Fairness and throughput are often difficult to conciliate as objective for a SMT scheduling policy. A SMT scheduling policy often optimizes only one performance metric at the sacrifice of the other metric. Our simulations show that a SIWW-based SMT fetch policy, using several indicators of execution roughness, is able to achieve at the same time state-of-the-art throughput, state-of-the-art fairness and state-of-the-art harmonic performance mean. In particular, it improves throughput over the Hill-Climbing policy [3] optimized for throughput, fairness over the Hill-Climbing policy optimized for fairness and harmonic performance mean over the Hill-Climbing policy optimized for harmonic performance.

The remainder of the paper is organized as follows. Section 2 recalls the principles of Speculative Instruction Window Weighting and presents its application to SMT fetch policies. Our experimental framework is described in Section 3. Performance of SIWW fetch policies is analyzed Section 4. Finally, Section 5 summarizes this study.

2 Speculative Instruction Window Weighting

Speculative Instruction Window Weighting (SIWW) was introduced for fetch gating control on superscalar processors [23]. The underlying idea was to estimate the "amount of work" already present in the processor through a single measure. In practice, a weight is assigned to each instruction. When the total weight of the speculative instruction window is above some threshold, there is a poor performance benefit in fetching subsequent instructions immediately, thus instruction fetching can be gated.

SIWW seems a natural vehicle for fairness control on instruction fetch policy on SMT processors; instructions are fetched for the thread with the minimum weight, i.e., with the minimum of pending work. Thus the SIW weights of the different threads tend to stay balanced.

However, there is no clear definition of "the amount of work" for a given instruction or a given instruction group: the SIW weight is an estimation of this amount of work. Since we target the instruction fetch policy, we leverage previous work on SMT fetch policies that pointed out a number of indicators of *execution roughness* that can be used to assign weights to instructions.

We consider 4 major indicators of execution roughness. First we consider the number of in-flight instructions as in the ICOUNT policy [21]. Second, we use control flow predictability [14, 12] since executing wrong-path instructions reduces the processor's throughput. Third off-chip memory accesses [20] are costly instructions as they take a long time to finish. A fourth important metric of execution roughness is memory-level parallelism (MLP) [4]. If present, MLP implies that multiple off-chip memory accesses take only as much time as a single off-chip memory access.

2.1 SIWW Implementation

As already pointed out, SIWW is intended to be a measure that estimates the "amount of work" already present in the processor for a given thread. An instruction-by-instruction computation was chosen in [23] to facilitate the runtime computation. The speculative instruction window weight is estimated as the sum of the individual weights of the in-flight instructions.

$$\text{SIW weight}(T) = \sum_{i \in \text{SIW}(T)} c(i)$$

The term $c(i)$ represents the weight contribution for instruction i. $\text{SIW}(T)$ represents the set of in-flight instructions of thread T.

At run-time, when an instruction is decoded, it enters the speculative instruction window, while executing the instruction removes it from the speculative instruction window. Consequently, the SIW weight is computed *incrementally*: during any cycle the SIW weight is increased with the weight contributions of the instructions

#	Opcode	Indicators	Contri-bution	Cumulative SIW Weight
1	add		1	1
2	blt	high-confidence	2	3
3	ld	on-chip access (hit)	1	4
4	shl		1	5
5	sub		1	6
6	ld	off-chip, no MLP	128	134
7	st		1	135
8	bne	low-confidence	8	143

Figure 1: Assigning SIW weight contributions to instructions.

Fetch	Cycle	Execute	SIW weight T0 / T1
T0: 1	– 0 –		1 / 0
T1: 1	– 1 –		1 / 1
T0: 1	– 2 –	T0: 1	1 / 1
T1: 1	– 3 –	T1: 1	1 / 1
T0: 1	– 4 –	T0: 1	1 / 1
T1: 1	– 5 –	T1: 1	1 / 1
T0: 1	– 6 –	T0: 1	1 / 1
T1: 1	– 7 –	T1: 1	1 / 1
T0: 1	– 8 –	T0: 1	1 / 1
T1: 1	– 9 –	T1: 1	1 / 1

(a) Equally progressing threads

Fetch	Cycle	Execute	SIW weight T0 / T1
T0: 1	– 0 –		1 / 0
T1: 1	– 1 –		1 / 1
T0: 4	– 2 –	T0: 1	4 / 1
T1: 1	– 3 –	T1: 1	4 / 1
T1: 1	– 4 –	-	4 / 2
T1: 1	– 5 –	T1: 1	4 / 2
T1: 1	– 6 –	T1: 1	4 / 2
T1: 1	– 7 –	T0: 4	0 / 2
T0: 1	– 8 –	T1: 1	1 / 1
T1: 1	– 9 –	T1: 1	1 / 1

(a) Equally progressing threads

Figure 2: Two threads (T0 and T1) execute on a simple SMT processor.

decoded in that cycle and the SIW weight is decreased with the weight contributions of the instructions that write-back in that cycle.

For selecting the weight contribution of the instructions, we rely on the instruction types (e.g. branch, load, other) and on various indicators of execution roughness (e.g. branch confidence, predicted hit/miss in last-level on-chip cache, etc.). An instruction is assigned a weight contribution that intends to represent the amount of resources that it may contribute to block.

An example clarifies the computation of the SIW weight. We assume the weight contributions of the best-performing SIWW SMT fetch policy investigated in this paper (policy SIWW-CF-HM-MLP-gating in Table 4, page 10). Figure 1 illustrates the SIW weights for a sequence of instructions. The respective weight contributions range from 1 for a simple ALU instruction to 128 for loads predicted to miss the on-chip caches. Summing all weight contributions results in a SIW weight of 143 for the whole instruction group.

2.2 SMT fetch policy based on SIWW

SIWW-based policies steer SMT instruction fetch using the SIW weight. Note that a threads' SIW weight continuously tracks the amount of work present in the speculative instruction window for that thread. Fairness is maximized by fetching instructions for the thread with the smallest SIW weight, i.e. the least amount of work in the pipeline.

It is very important to also gate threads if they have too much work in the pipeline, i.e. they threaten to monopolize processor resources. To avoid such situations, we introduce a SIW gating threshold: threads are fetch gated if their SIW weight exceeds the SIW gating threshold.

Figure 2 illustrates the SIWW SMT fetch policy. We assume a simple processor that fetches and executes at most one instruction every cycle. In Figure 2 (a), both threads only encounter instructions with a weight contribution of 1. The threads progress equally fast. In Figure 2 (b), thread T0 encounters an instruction with a weight contribution of 4. As this instruction has a high weight, instruction fetch becomes biased towards thread T1 till the high weight instruction executes.

SIWW is a generic SMT fetch policy. Previously published fetch policies can be easily expressed in the SIWW framework (Table 1). E.g. ICOUNT [21] is obtained by setting all weight contributions to 1 and by setting the SIW gating threshold equal to the reorder buffer size (or larger). Predictive data gating (PDG) [7] predicts whether

a load has long latency in the decode stage and it gates a thread if it has too many long-latency loads in-flight. PDG uses ICOUNT to select among the not-gated threads. PDG is obtained by setting all weight contributions to 1, except for predicted load misses which have a weight contribution equal to N, which is a fixed number that is larger than the maximum number of in-flight instructions. The SIW gating threshold is also set equal to N. These weight contributions imply that threads are gated when at least one predicted load miss is in-flight. If no predicted load-misses are in-flight, then the policy behaves as ICOUNT.

Other examples are Luo *et al.*'s FPG [14] and Kang and Gaudiot's SAFE-T [12] policies. FPG selects the threads with the fewest in-flight low-confidence branches and applies ICOUNT to this reduced set of threads. Additionally, fetch is gated for threads with at least 2 low-confidence branches in-flight. SAFE-T, on the other hand, selects the threads according to ICOUNT, except that ties are broken by the number of low-confidence branches. This behavior is achieved by assigning a slightly larger weight contribution to low-confidence branches than to other instructions.

2.3 A Few SIWW Implementation Issues

Computing the SIWW at execution time requires logic to determine the weight contribution of the instructions and logic to compute the SIW weight. The logic to determine the weight contributions essentially consists of the indicators (branch confidence predictors, hit/miss predictors, MLP predictors) that are also considered for other SMT fetch policies. Logic to compute the global SIW weight essentially consists of a tree of adders/subtracters.

However, some specific situations where the SIW weight is globally affected must be addressed.

2.3.1 Branch Misprediction Recovery

Mispredicted branch instructions require that wrong-path instructions are flushed and that the processor recovers its state to the point of the branch instruction. Implementing exact recovery of the SIW weight is non-trivial as a detailed analysis of all in-flight instructions is necessary.

In [23] it is shown that a sufficient approximation to the SIW weight can be made by setting the SIW weight to zero when recovering from a mispredicted branch. This has the effect of temporarily underestimating the SIW weight, until all instructions preceding the branch have executed.

One precaution must be made to prevent unfairness, which could happen when the branch is resolved before all prior instructions have executed. These instructions will execute after the SIW weight is set to zero, resulting in a negative or permanently underestimated SIW weight. Hereby, the thread may get more than its fair share of the processor resources. This situation is prevented if the program order of instructions can be easily tested, e.g. by assigning sequence numbers to instructions. The SIW weight contribution of an instruction is subtracted from the SIW weight only when it occurs in program order before the last recovered instructions [23].

2.3.2 Correcting Load Hit/Miss Predictions

Load instructions that are predicted to hit in the on-chip caches but that miss clog up the issue queues after all. It is beneficial to increase the SIW weight contribution of these instructions when the cache miss is detected. PDG makes the same design decision [7]. To implement it in the SIWW framework, it is necessary to increment the

Table 1: SIWW formulation of various SMT fetch policies. N is a number larger than the maximum number of in-flight instructions.

		Weight contribution			
Instr type	**Indicator**	**ICOUNT**	**PDG**	**FPG**	**SAFE-T**
Simple		1	1	1	N
Cond br	hi-conf	1	1	1	N
Cond br	lo-conf	1	1	N	$N+1$
Indir br	hi-conf	1	1	1	N
Indir br	lo-conf	1	1	N	$N+1$
Returns		1	1	1	N
Loads	pred hit	1	1	1	N
Loads	pred miss	1	N	1	N
SIW gating threshold		n/a	N	$2N$	n/a

SIW weight for a thread also when load instructions execute. This requires extra adders for computing the SIW weight. We reuse these adders to reduce the SIW weight contribution for loads that hit but were predicted to miss.

2.4 Discussion of Weight Contributions

We initially chose SIW weight contributions by making educated guesses of the execution time and occupancy of resource usage by individual instructions. This was then refined by means of experimentation.

We first assigned 1 as the weight of standard ALU operations. Experiments with assigning different weights for medium-latency operations such as floating-point operations or integer multiplication and division showed marginal impact because these latencies are generally well hidden by out-of-order execution. Therefore, those operations are also assigned 1 as a weight.

On the other hand, branch mispredictions and load misses, particularly L2 misses, are not hidden by the pipeline. The SIW weight contributions of these miss events must be larger than those of simple ALU instructions. Therefore we set different weights for high confidence and low confidence branch predictions, and for predicted hit loads and predicted miss loads. However, as these miss events are *predicted*, the execution times and the resource occupancies of individual predicted-miss instructions vary in a large range, depending on the accuracy of the miss predictor. We have to compensate the inaccuracy of the miss predictors by means of the SIW weight contribution. We bias the SIW weight contribution to extreme values for very accurate miss predictors. We select medium SIW weight contributions for very inaccurate miss predictors.

Even when miss events can be accurately predicted, their penalty may differ in a very wide range. E.g., some load miss penalties are quite small when prefetching hides memory latency. Therefore, the weight contribution of a load miss or branch misprediction should be smaller than the worst-case memory latency. It should be also noticed that some penalties overlap in time, implying that their combined execution wastes less issue slots than what each instruction individually suggests. E.g. multiple load misses may be in progress at the same time, an effect called memory-level parallelism (MLP). If MLP is not accounted for by the SIWW policy, then it is wise to reduce the weight contribution of load misses to account for MLP.

In the end, we found that many weight contributions are about equally good, so only a few sparsely positioned values must be evaluated.

3 Evaluation Environment

We evaluate SMT fetch gating techniques using the SPEC CPU2000 benchmarks. These benchmarks are compiled for the Alpha ISA using the Compaq C/C++ and Fortran compilers. We use SimPoint [19] to select 500M-instruction traces.

The SMT fetch policies are evaluated in an SMT simulator that is derived from the Sim-Flex simulator [1]. The processor is adapted for SMT by duplicating the architectural state for every thread. Furthermore, global branch histories and the RAS are duplicated per thread [9].

3.1 Predictors

This work assumes a 64Kbits O-GEHL conditional branch predictor [18]. It is equipped with a free confidence estimator [23], similar to perceptron predictors [10]. Indirect branches are predicted with a cascaded branch target predictor [6]. Confidence is estimated by attaching a resetting 2-bit counter to each entry in the predictor. The counter is incremented on a correct prediction and set to zero on an incorrect prediction. High confidence follows when the counter is saturated in the highest state.

We implement load hit/miss predictors with saturating counters. The prediction table consists of 4 K entries, accessed using the load's program counter. The table contains 4-bit counters that are incremented by 1 on a cache hit and decremented by 2 on a cache miss. A load instruction is predicted to hit if the counter is not less than 8.

We implement MLP predictors as in [8]. A per-thread shift register tracks MLP. The pair (1,PC) is shifted into the register when a long-latency load commits. Otherwise, the pair (0,PC) is shifted in. A 4 K-entry table remembers for each load instruction the amount of MLP it has. This table is updated for the program counter that is shifted out of the shift register upon commit. If the shift register contains long-latency loads, then the shifted out PC exhibits MLP. While the predictor tracks and predicts the amount of MLP, we only use it to predict the presence of MLP.

[1] `http://www.ece.cmu.edu/~simflex`

Table 2: Baseline Processor Model

Processor core	
Issue width	8 instructions
Reorder buffer	128 insn.
Issue queue	32 insn.
Load-store queue	32 insn.
Fetch Unit	
Fetch width	8 instructions, 2 branches/cycle
Cond. branch predictor	64 Kbits O-GEHL
Return address stack	32 entries, checkpoint top 1
Branch target buffer	512 sets, 4 ways
Cascaded branch target predictor	64 sets, 4 ways, 8-branch path history
Memory Hierarchy	
L1 I/D caches	64 KB, 4-way, 64B blocks
L2 unified cache	512 KB, 8-way, 64B blocks
L3 unified cache	4 MB, 8-way, 64B blocks
Cache latencies	1 (L1), 6 (L2), 20 (L3)
Memory latency	min. 200 cycles
Instruction/data TLB	64/128-entry, fully assoc.
TLB miss latency	200 cycles

3.2 Workload Composition

Multi-threaded workloads are composed from multiple single-threaded SPEC CPU2000 benchmarks. The construction of the workloads is performed in three steps.

3.2.1 Workload Characterization

We execute the benchmarks on the baseline processor model (Table 2) and measure key metrics related to the behavior of SMT: branch mispredicts per kilo-instruction (Branch-MPKI), the data TLB misses per kilo-instruction (DTLB-MPKI), the unified L3 cache misses per kilo-instruction (UL3-MPKI), the instructions per cycle metric (IPC) and the amount of memory-level parallelism (MLP). Note that the Branch-MPKI metric includes statistics on indirect branches and procedure returns. The MLP metric is defined as the average number of outstanding long latency cache misses while there is at least one such cache miss outstanding [4]. Table 3 lists the values of these metrics on our benchmarks.

3.2.2 Workload Analysis

K-means cluster analysis [16] is performed on the workload metrics to cluster the benchmarks in groups with similar properties. We have experimented with several cluster sizes and found that 4 clusters matches the data well. We label the resulting clusters as 'd', 'D', 'C' and 'M' (Table 3). Benchmarks in the 'd' and 'D' clusters contain all of the SPECfp benchmarks and a few SPECint. They have good branch predictability. They stress the memory systems either lightly ('d') or heavily ('D'). Most of the SPECint benchmarks are type 'C'. These benchmarks are recognized by worse branch predictability. Finally, the 'M' category is the mcf benchmark as it is the only benchmark in this study that stresses both control flow prediction and the memory system.

3.2.3 Workload Composition

We construct N-thread workloads consisting of N single-threaded programs with $N = 2$ or $N = 4$. Workloads with different properties are constructed by combining traces with different properties.

In the case of two-thread workloads, we construct two workloads for every pair of benchmark clusters. E.g., there are two workloads with one thread from the 'd'-cluster and one thread from the 'C' cluster.[2] Single-threaded programs are selected randomly from within the designated clusters.

[2]There is only one 'M'-'M' workload for obvious reasons.

Table 3: Key benchmark metrics for SMT performance and the assigned clusters.

	IPC	Branch MPKI	DTLB MPKI	UL3 MPKI	MLP	C
ammp	2.13	0.83	3.26	0.33	4.81	d
applu	0.91	0.03	0.33	19.00	7.85	D
apsi	4.29	0.00	0.01	0.74	3.70	d
art1	1.13	0.02	3.03	0.00	5.96	d
art2	1.13	0.02	3.04	0.00	5.96	d
equake	0.32	1.44	0.42	24.18	6.00	D
facerec	2.50	0.30	0.14	1.06	8.70	d
fma3d	1.78	0.01	0.04	2.44	4.55	d
galgel	3.55	0.15	0.50	0.03	5.37	d
lucas	0.66	0.01	2.80	21.82	7.23	D
mesa	2.71	1.13	0.44	0.55	3.00	d
mgrid	1.10	0.01	0.15	6.59	1.85	d
sixtrack	3.71	0.17	0.01	0.00	3.09	d
swim	0.68	0.05	12.75	22.78	5.99	D
wupwise	2.28	0.02	0.06	1.97	5.67	d
bzip2	2.86	5.74	0.13	0.30	2.36	C
crafty	3.94	5.24	0.00	0.01	1.31	C
eon	3.73	4.51	0.00	0.00	1.06	C
gap	2.67	0.37	0.01	0.93	1.70	d
gcc	3.50	2.84	0.02	0.58	2.29	C
gzip	2.39	5.59	0.03	2.16	2.45	C
mcf	0.20	8.37	45.90	53.84	9.13	M
parser	2.27	5.78	0.02	0.22	1.86	C
perlbmk	3.50	2.87	0.00	0.00	1.34	C
twolf	1.83	9.38	0.00	0.00	1.65	C
vortex	3.57	0.30	0.31	0.29	2.16	d

For the four-thread workloads, we construct up to 6 workloads for every pair of benchmark clusters. E.g. for the combination of 'C' and 'd' clusters, we create 2 workloads with 3 'C' benchmarks and 1 'd' benchmark, 2 workloads with 2 'C' benchmarks and 2 'd' benchmarks and 2 workloads with 1 'C' benchmark and 3 'd' benchmarks.

3.3 Performance Metrics

Performance metrics for a multiprogrammed system must reflect that there is no one-to-one relationship between the instruction count and the amount of work performed by a thread, as the baseline IPC of the threads may differ. Therefore, we base our metrics on relative IPC of threads.

The relative IPC (rIPC) of a thread is the ratio of the thread's IPC in multi-threaded operation to the thread of the IPC when it is running alone.

$$rIPC = \frac{IPC_{SMT}}{IPC_{SingleThreaded}}$$

Relative IPC is a number between 0 and 1. The value 1 is reached only if the thread does not suffer any slowdown due to the other threads.

We define fairness as the minimum of rIPCs among the threads and we define throughput as the sum of the rIPCs:

$$\begin{aligned} \text{fairness} &= min_i\ rIPC_i \\ \text{throughput} &= sum_i\ rIPC_i \end{aligned}$$

This definition of throughput stresses that SMT processors can get more work done in a unit of time than single-threaded processors. E.g., if throughput equals 1.4, then the SMT processor performs 1.4 seconds worth of work during 1 second. If, however, throughput is less than 1, then SMT execution effectively slows down the processor.

Contrary to the harmonic mean of relative IPCs [15], our definition of fairness stresses that all threads must achieve some minimum performance level in order to be fair. This harmonic mean will also be reported for the sake of completeness.

Table 4: SIW weight contributions.

Instr type	Indicator	CF	CF-HM	CF-HM-gating	CF-HM-MLP-gating
Simple		1	1	1	1
Cond br	hi-conf	2	2	2	2
Cond br	lo-conf	8	8	8	8
Indir br	hi-conf	3	3	3	3
Indir br	lo-conf	8	8	8	8
Returns		2	2	2	2
Loads	pred hit	1	1	1	1
Loads	pred miss, with MLP	1	64	64	16
Loads	pred miss, no MLP	1	64	64	128
SIW gating threshold		n/a	n/a	128	128

In our evaluation figures, the last column presents an average on the benchmarks. But, due to the biased construction of the multi-threaded workloads, the 'M' and 'D' clusters are overrepresented in this average so it must be treated with some reservation.

4 Evaluation

4.1 SIWW-Based SMT Fetch Policies

In this section, we investigate the use of four SIWW fetch policies that are progressively more refined. The first policy uses control-fow information. The utilization of an SMT processor is improved somewhat when focusing instruction fetch to threads that are most likely on the correct execution path [14, 12]. The control-flow-conscious SIWW fetch policy assigns different weight contributions to high-confidence and low-confidence conditional and indirect branches, as well as to return instructions (Table 4). Hereby, the SIWW-CF policy reaches higher throughput and fairness than the ICOUNT policy and than, e.g., Luo's FPG policy [14]. On average, SIWW-CF improves throughput and fairness over the FPG policy by 2.2% and 7.9% respectively on the 2-thread workloads. It improves throughput and fairness by 6.6% and 16.2% respectively on the 4-thread workloads. FPG, however, is less fair than ICOUNT in our evaluation. Compared to long-latency loads, control flow predictability plays a small role for SMT fetch control.

4.1.1 Adding Cache Hit/Miss Information

The SIWW- CF-HM policy incorporates load hit/miss information by assigning high weight contributions to predicted load misses (Table 4). This policy does not limit the number of in-flight load misses, although it strongly penalizes their presence. The SIWW-CF-HM-gating policy uses the same weight contributions as SIWW-CF-HM but fetch gating is enabled on a thread when its SIW weight exceeds a threshold.

Our experiments (Figure 3 and Figure 4) compare SIWW-CF-HM with fetch policies only accounting load misses, e.g. PDG-1-COT (i.e. predictive data gating [7] allowing at most 1 in-flight load miss per thread, improved by the continue-oldest-thread heuristic [1]). With the SIW weights used in the presented experiments, PDG-1-COT policy has an edge on some of the workloads featuring 'C' type threads and SIWW-CF-HM policy makes strong improvements over PDG-1-COT on workloads featuring 'D' and 'M' type threads. Small gains are made for the 'dd' workloads. PDG-1-COT abruptly stops fetch on the thread featuring a cache miss. This radical solution is effective when no MLP is encountered. On the other hand on applications featuring MLP, SIWW-CF-HM is more effective. Experiments using a larger SIW weight (128) makes SIWW-CF-HM behave very similar to PDG-1-COT, abruptly providing the fetch bandwidth to the alternate threads after the first miss.

SIWW-CF-HM-gating further provides some extra throughput and fairness over SIWW-CF-HM. The main difference between the SIWW-CF-HM and SIWW-CF-HM-gating policies occurs for workloads featuring 'C' type threads. These threads contain no MLP and should be gated at the first load miss in the thread to avoid occupancy of resources by stalled instructions. The SIWW-CF-HM-gating policy achieves this gating, provided

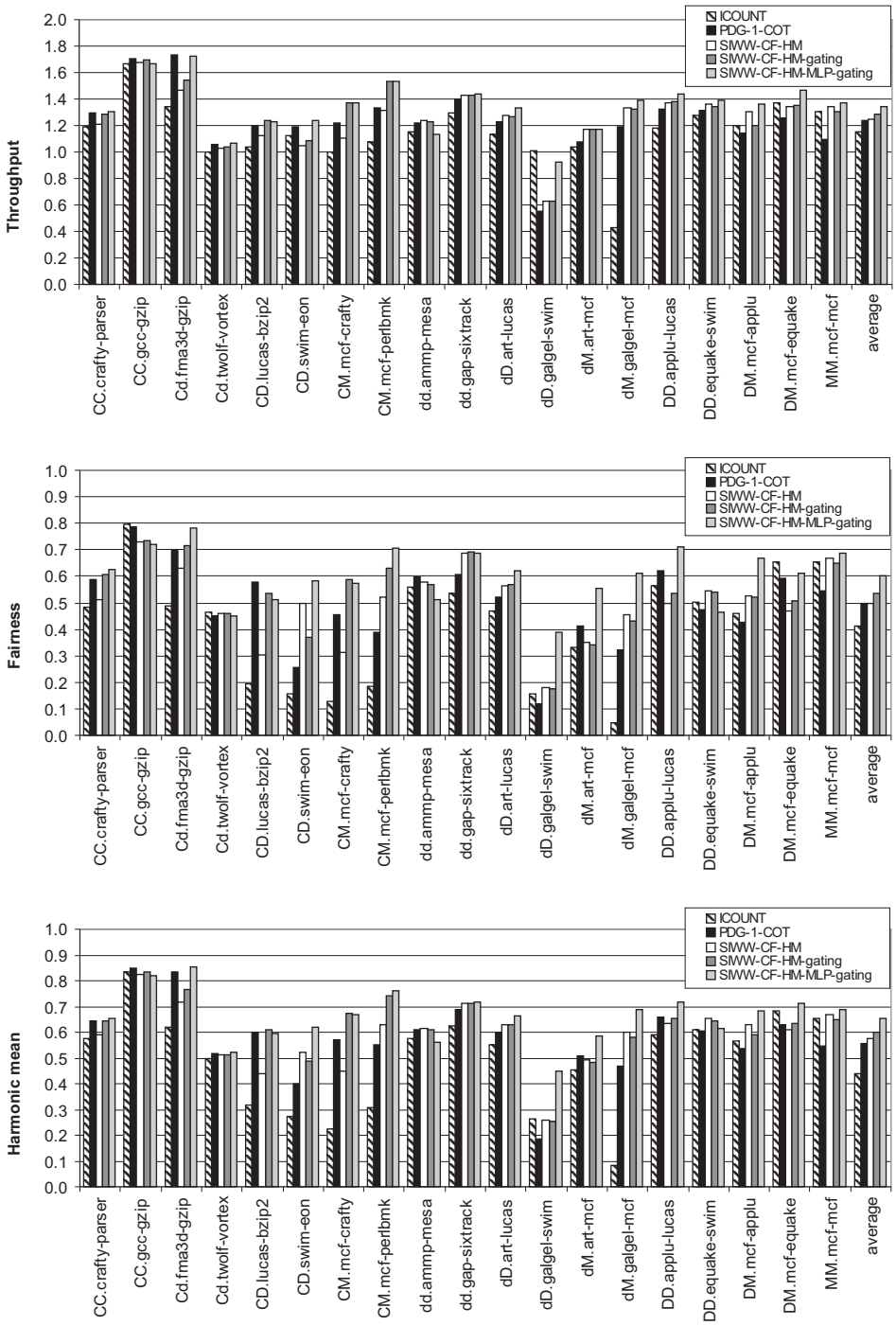

Figure 3: Throughput and fairness for SIWW fetch policy taking control flow predictability and load hit/miss predictions into account, as well as a policy leveraging MLP. Results for 2-thread workloads.

that other in-flight instructions bring the SIW weight close to the gating threshold. In contrast, the SIWW-CF-HM policy allows all threads to build up a large, unbounded, SIW weight.

4.1.2 Leveraging Memory Level Parallelism

The fourth indicator of execution roughness in our study is memory-level parallelism. The SIWW-CF-HM-MLP-gating policy assigns lower weights to load misses with MLP than those without MLP (Table 4).

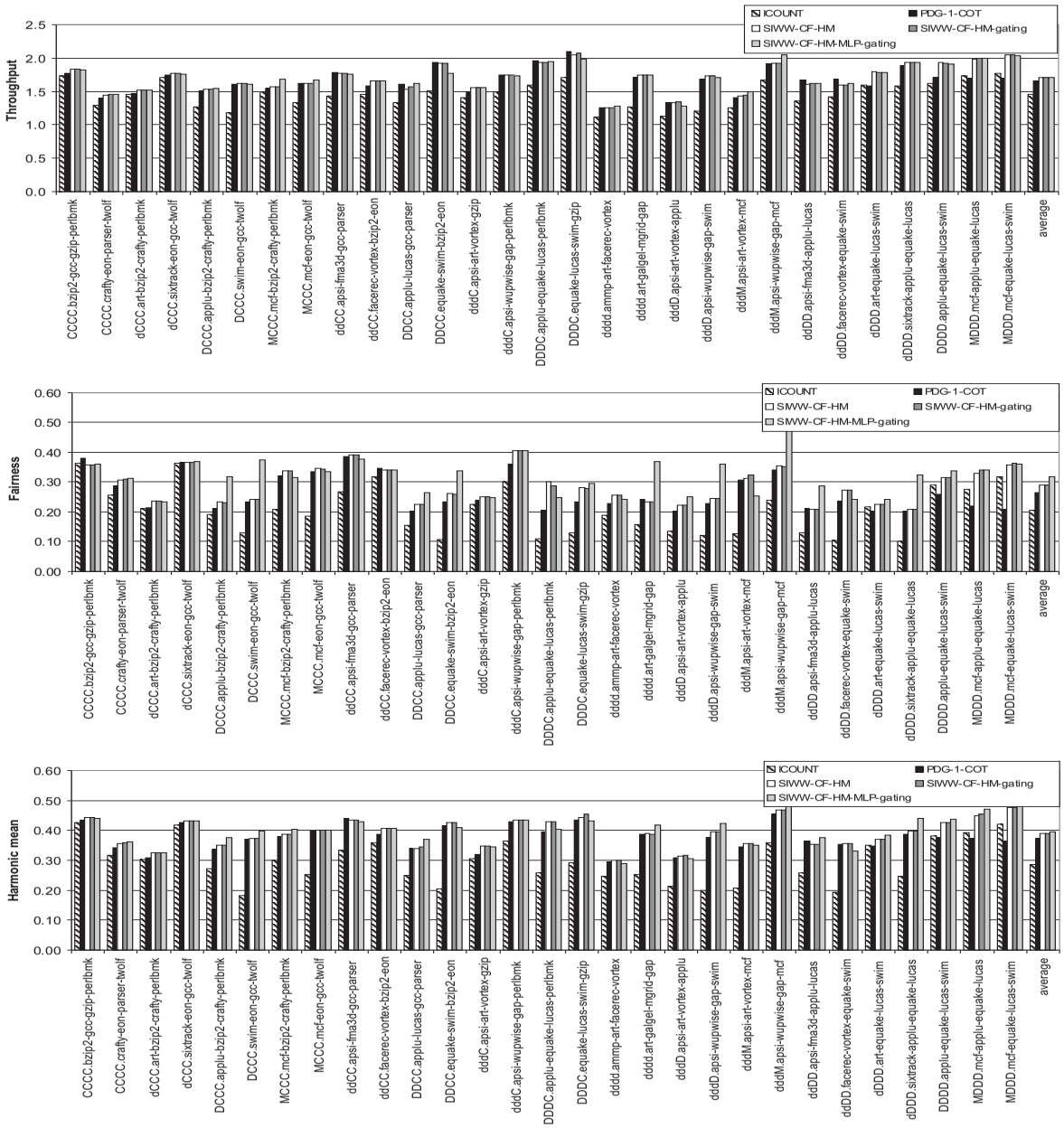

Figure 4: Throughput, fairness and harmonic mean for SIWW fetch policy taking control flow predictability and load hit/miss predictions into account, as well as a policy leveraging MLP. Results for 4-thread workloads.

The SIWW-CF-HM-MLP-gating policy achieves 8.6% and 3.0% higher throughput than the PDG-1-COT policy on the 2-thread and 4-thread workloads respectively (Figures 3 and 4). However the most visible impact is on fairness (21.6% and 20.7% on the 2-thread and 4-thread workloads respectively) and on harmonic mean (17.7% and 6.3% on the 2-thread and 4-thread workloads respectively)

4.1.3 Predictionless Policies

Predictors for load misses and MLP can be avoided by assuming a load hits until a cache miss occurs, at which time the SIW weight contribution is bumped up. The benefit of this approach depends strongly on the workload. Making SIWW-CF-HM-gating non-predictive improves throughput and fairness by 0.3% and 6.5%, respectively,

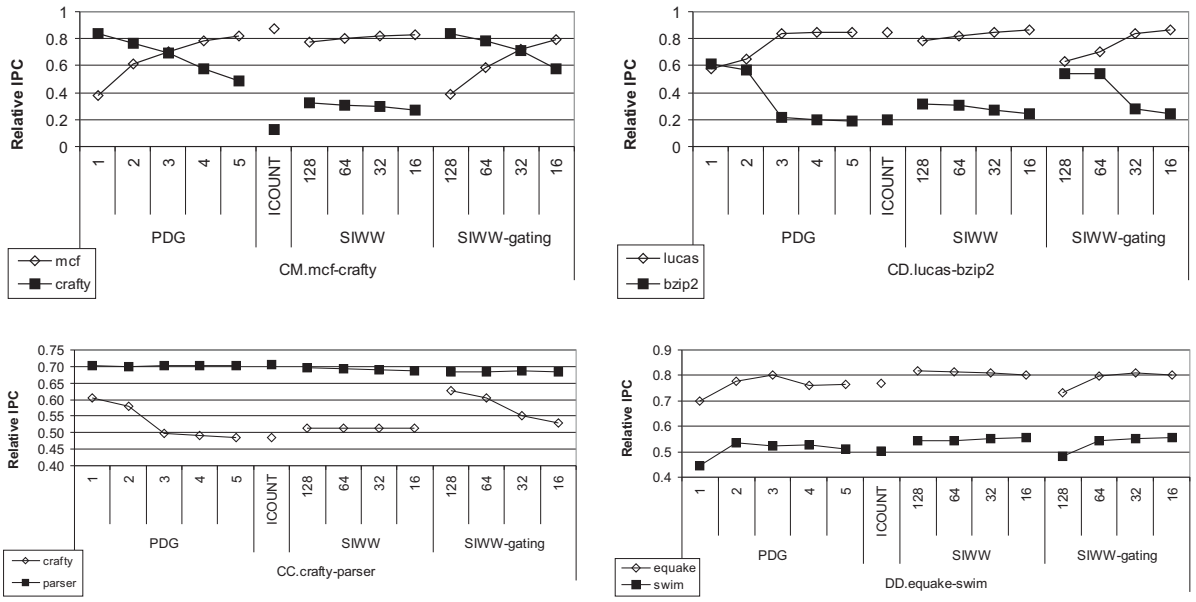

Figure 5: The impact of the load miss weight contribution on throughput and fairness for the 2-thread workloads. The SIWW policies use the control flow and load hit/miss indicators. The weight contribution of load misses is varied between 16 and 128 for the SIWW policies.

for the 2-thread workloads. Making SIWW-CF-HM-MLP-gating non-predictive decreases throughput and fairness by 0.34% and 3.7%, respectively. This policy still outperforms state-of-the-art policies like hill climbing (Section 4.3). Thereby, *load hit/miss prediction and MLP prediction are not strictly necessary for building SIWW policies*.

4.2 An Analysis of Fetch Gating

The ICOUNT policy is naturally biased towards threads with high IPC. To improve the SMT fetch policy, one needs to mitigate such an unfair bias. Figure 5 shows the relative IPC for three workloads and several SMT fetch policies. The SIWW policies use the control flow and load hit/miss indicators. The weight contribution of load misses is varied between 16 and 128 for the SIWW policies. Note that ICOUNT allows mcf to monopolize resources.

By gating threads, the PDG policy can increase fairness; and it can even become unfair towards one thread, e.g. the mcf, bzip2 and crafty threads. When increasing the PDG gating threshold from maximum 1 outstanding cache miss to maximum 5, the mcf thread gets gradually more resources while the crafty thread gets less, until the situation of ICOUNT is obtained (PDG equals ICOUNT for large gating thresholds).

The SIWW-CF-HM policy has much less impact on fairness. No matter how large or small we choose the load miss weight contribution, threads that tend to monopolize resources are still allowed to do so. Only by introducing gating in the SIWW policy (SIWW-CF-HM-gating) can we impact fairness. This impact is higher when the load miss weight contribution is larger, also leading to unfairness towards one of the threads. On the other hand, when the load miss weight contribution is smaller, then relatively fair division of resources can be obtained. Note, however, that the load miss weight contribution at which a fair division of resources is obtained varies, as everything, between workloads.

As a side remark, note that Figure 5 illustrates the impact of SIW weight contributions on the performance metrics. One can see here that changing one particular SIW weight contribution changes performance in a monotonous and quite predictable manner, making it easy to experimentally determine suitable SIW weight contributions.

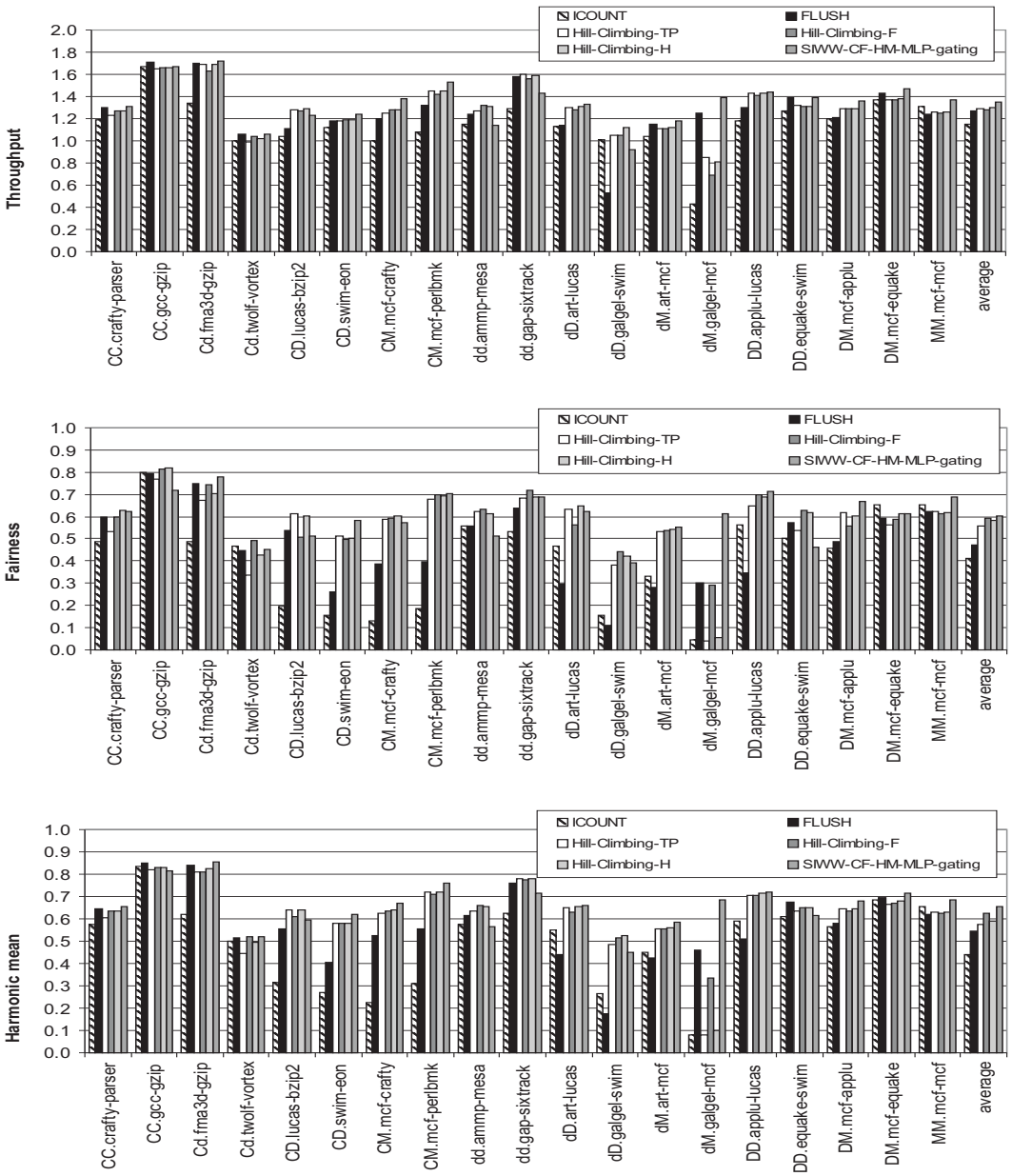

Figure 6: Throughput, fairness and harmonic mean for the SIWW-CF-HM-MLP-gating fetch policy, compared to several known fetch policies. Results for 2-thread workloads. The hill climbing policy labeled -TP, -F and -H optimize throughput, fairness and harmonic mean, respectively.

4.3 Comparison to State-of-the-Art

We compare our best SIWW policy to several previously published SMT fetch policies, including ICOUNT, PDG-1-COT, FLUSH [20] and Hill-Climbing [3] (Figure 6 and 7). FLUSH [20] and Hill-Climbing [3] are often considered as state-of-the-art policies. Overall, throughput numbers show small variations due to the size of the reorder buffer and issue queue, which limit the overall speedup of SMT.

4.3.1 FLUSH [20]

When an off-chip memory access occurs, the FLUSH policy flushes all subsequent instructions of the same thread from the pipeline [20]. Note that the FLUSH policy does not perform very well on our throughput and fairness metrics because it is an unfair policy, giving precedence to threads without off-chip accesses. In the original

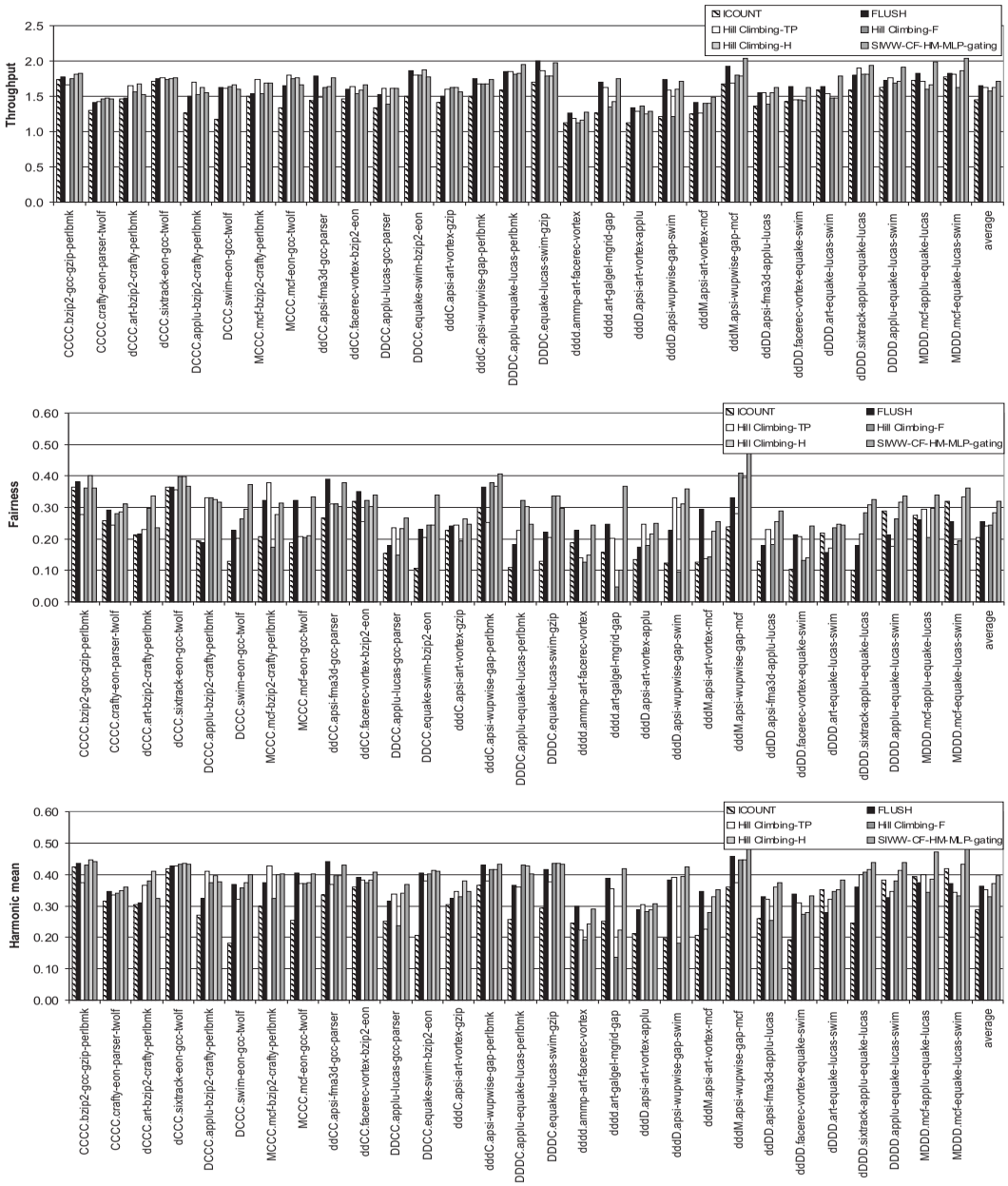

Figure 7: Throughput, fairness and harmonic mean for the SIWW-CF-HM-MLP-gating fetch policy, compared to several known fetch policies. Results for 4-thread workloads. The hill climbing policies labeled -TP, -F and -H optimize throughput, fairness and harmonic mean, respectively.

publication [20], FLUSH was evaluated on the total IPC over all threads, which in our simulations improves over ICOUNT by 52.9% on average for the 2-thread workloads, which is better than the other policies we evaluated.

4.3.2 Hill-Climbing [3]

Hill-Climbing policies continuously monitor the performance metric (e.g. total IPC, throughput or fairness) and adjust the resource distribution (expressed as the number of issue queue and reorder buffer entries) to optimize the performance metric [3]. Hill-Climbing is a purely black-box approach: it is enterily agnostic of the instructions in individual threads.

In our implementation, the algorithm divides issue queue resources among threads by setting a maximum occupation limit for each thread, N_i, $i = 1, \cdots, T$ with T the number of threads. The sum of all N_i equals the issue queue size. The algorithm remembers a base configuration consisting of the N_i values. The tuning algorithm

divides time in epochs of 64K cycles. It uses a state machine to continuously cycle through T states. In each state, the base configuration is modified to study if it can be improved. In state i, the tuning algorithm assigns δ fewer issue slots to all threads, except for thread i which gets $\delta(T-1)$ additional issue slots. $\delta = 4$ in our experiments. This configuration is enforced during the entire epoch and performance is measured for it. After T epochs, the configuration with the best performance is copied to the base configuration and the tuning algorithm starts again. We experimented with different parameter settings for epoch size and δ and with applying the tuning algorithm to the issue queue or the reorder buffer or both. The best simulation results are presented in the paper, which is for tuning the issue queue size.

We believe that we have made reasonable efforts to ensure that the simulation results are as good as possible for the Hill Climbing policy and any other policy that we have implemented.

We present results on three hill-climbing policies: one that optimizes throughput, one that optimizes fairness and one that optimizes harmonic mean.

4.3.3 Simulation results analysis

Overall, despites our proposal is not targeting any particular performance metric, SIWW-CF-HM-MLP-gating achives the best throughput, the best fairness and the best harmonic mean. In particular, it achieves higher throughput than Hill Climbing policy optimized for throughput, higher fairness than Hill Climbing policy optimized for fairness and higher harmonic mean than Hill Climbing policy optimized for harmonic mean. Furthermore, Hill-Climbing optimizes only one metric and diminishes results on the others.

Our simulation results confirm that Hill-Climbing is efficient at optimizing a performance metric. Hill-Climbing performs about as well as FLUSH when optimizing throughput. We observe that the difference between these policies is largest when there is more diversity in the workload, in particular, in the CM, dM and DM workloads. Hill-Climbing optimizes fairness particularly well: it increases fairness by 25.0% over FLUSH and 43.3% over ICOUNT for the 2-thread-workloads. Note, however, that hill climbing policies optimize the performance metric on an interval-by-interval basis, a strategy that does not guarantee an overall optimum [17]. This also holds for the throughput and fairness metrics used in this paper which is to the disadvantage of Hill-Climbing. The Hill-Climbing policy has problems to correctly divide resources for the galgel-mcf workload (dM). This is most pronounced when optimizing throughput or harmonic mean, in which cases fairness and harmonic mean drop to unacceptably low levels. Note that this does not happen for the fairness metric: because we optimize the minimum of relative IPCs, we also guarantee a lower bound to the harmonic mean. Due to this single benchmark result, the harmonic mean of the Hill-Climbing policy is larger when optimizing fairness than when optimizing harmonic mean.

However the three versions of Hill-Climbing are outperformed by SIWW-CF-HM-MLP-gating and for the three considered metrics. For instance, on 4-thread, SIWW-CF-HM-MLP-gating outperform Hill Climbing optimized for throughput by 4.9% on throughput, 32.3 % on fairness and 13.1% on harmonic mean.

The SIWW-based SMT fetch policies have several advantages over other fetch policies. First, as explained above, SIWW can take into account multiple events and it can trade-off their relative importance by properly selecting SIW weight contributions.

Second, SIWW-based policies can make an accurate estimate of the amount of work in each thread's in-flight instructions, provided that sufficient indicators of execution roughness are included. Hereby, they can change thread priorities quickly. In contrast, the FLUSH policy only changes thread priorities when off-chip memory accesses start or end. Hill-Climbing reconsiders resource partitioning only on interval boundaries (e.g. 64 K cycles).

Third, SIWW policies *predict* the properties of an instruction stream. As such they can act ahead of time, fetch-gating a thread even before a long-latency load miss occurs. Policies such as FLUSH and Hill-Climbing only act after the fact. Predicting the events in an instruction stream is at the same time a weakness of SIWW policies, as the predictions may be wrong and the predictors also bring an implementation cost.[3]

4.4 Background/Foreground Scenario

To underline the explicit resource allocation capabilities of SIWW, we explore the use of SIWW for executing a foreground thread at single-thread performance (rIPC ≥ 0.99) while making some progress on background threads.

IBM's POWER5 processor implements such a mechanism: the foreground thread is assigned a higher percentage of all decode cycles [11]. Figure 8 shows the relative IPC of the foreground and background threads obtained

[3] Although in many cases the predictors of execution roughness may be present in the architecture for other purposes also.

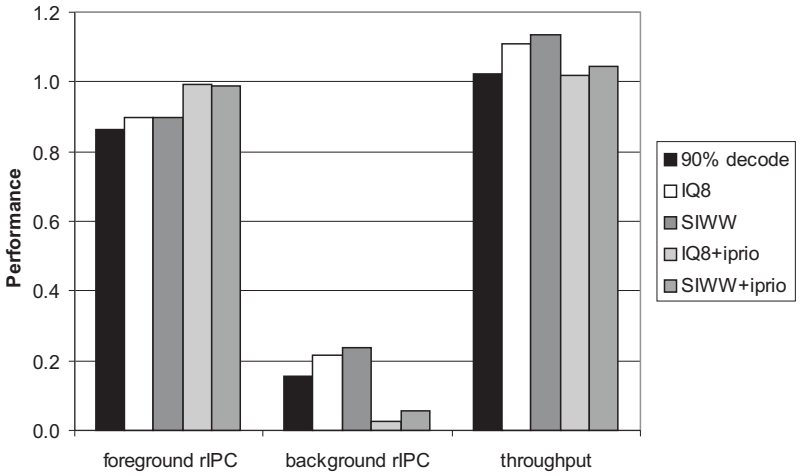

Figure 8: Relative IPC of foreground and background threads and total throughput averaged over all 2-thread workloads.

with this policy. The results show the average over all 2-thread workloads where the first thread in the workload is the foreground thread. Assigning a fixed percentage of decode slots to the foreground thread does not allow it to achieve a commensurate percentage of single-thread performance.

Dorai and Yeung [5] propose several mechanisms to maximize processor utilization. We highlight two of these. First, they limit the amount of issue queue resources for the background thread to at most 8 issue queue entries while otherwise applying ICOUNT. This increases the performance of the foreground thread, but has, however, the same short-comings as ICOUNT: it treats all instructions equally. SIWW-based policies can increase processor utilization. Figure 8 shows that SIWW-CF-HM-MLP-gating (our best SIWW policy) increases the performance of the background thread while maintaining foreground thread performance. Hereto, we set the SIW gating threshold to 256 for the foreground thread and to 16 for the background thread.

Dorai and Yeung also prioritize threads in the issue stage: each cycle, the foreground thread's instructions issue before the background thread's instructions [5]. This technique brings the foreground thread close to its single-thread performance when combined with the IQ8 policy. However, combining SIWW with issue stage prioritization also increases background thread performance. Foreground thread performance "decreases" from 0.992 to 0.988.

The bottom line is that SIWW-based policies manage processor resources more intelligently, *allowing the background thread access to a larger portion of the processor's resources without hindering the foreground thread*. Different trade-offs between foreground thread performance and background thread performance can be achieved with different settings for each thread's gating threshold.

5 Conclusion

The effective behavior of a SMT processor on a parallel or multi-programmed workload depends on a careful management of the hardware resources shared between the threads. We categorize resource allocation policies into implicit and explicit policies. Implicit policies build on the SMT fetch policy to steer instruction fetch towards those threads that will use resources best. These policies include, e.g. ICOUNT, predictive data gating (PDG) and FLUSH. Explicit resource allocation policies such as DCRA and Hill Climbing divide the resources by explicitly counting them.

Both types of policies have their benefits: the implicit policies can make instantaneous fetch steering decisions that are locally optimum, while explicit policies optimize resource distribution in the long term.

This paper proposes a new approach to constructing SMT fetch policies that combine the benefits of both approaches. These policies use Speculative Instruction Window Weighting (SIWW) to estimate the amount of outstanding work for each thread. These policies are implicit as they steer fetch to the threads with the least

amount of work. On the other hand, they perform explicit resource allocation by fixing the maximum amount of outstanding work for each thread.

In practice SIWW is a framework to design SMT fetch policies addressing any cause of hardware resource misusage by a thread. We have proposed a SIWW fetch policy taking into account branch mispredictions, off-chip memory accesses and memory-level parallelism. Targeting fairness or throughput is often contradictory and a SMT scheduling policy often optimizes only one performance metric at the sacrifice of the other metric. Our simulations have shown that the SIWW fetch policy can achieve at the same time state-of-the-art throughput, state-of-the-art fairness and state-of-the-art harmonic performance mean.

Acknowledgements

Hans Vandierendonck is Postdoctoral Research Fellow with the Fund for Scientific Research - Flanders (FWO). His research was also sponsored by Ghent University, by the Institute for the Advancement of Science and Technology in the Industry (IWT). André Seznec was partially supported by an Intel research grant. The collaboration was initiated in the framework of the European Network of Excellence on High-Performance Embedded Architectures and Compilation (HiPEAC).

References

[1] CAZORLA, F. J., FERNANDEZ, E., RAMIREZ, A., AND VALERO, M. 2003. Improving memory latency aware fetch policies for SMT processors. In *Intl. Symp. on High-Performance Computing (ISHPC)*. Vol. 2858. 70–85.

[2] CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNANDEZ, E. 2004. Dynamically controlled resource allocation in SMT processors. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 171–182.

[3] CHOI, S. AND YEUNG, D. 2006. Learning-based smt processor resource distribution via hill-climbing. In *ISCA '06: Proc. of the 33rd Ann. Intl. Symp. on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 239–251.

[4] CHOU, Y., FAHS, B., AND ABRAHAM, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In Proceedings of the 31st annual international symposium on Computer architecture. *SIGARCH Comput. Archit. News*, 76.

[5] DORAI, G. K. AND YEUNG, D. 2002. Transparent threads: Resource sharing in SMT processors for high single-thread performance. In *11th International Conference on Parallel Architectures and Compilation Techniques*. 30–41.

[6] DRIESEN, K. AND HOLZLE, U. 1998. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceeding of the 30th Symposium on Microarchitecture*.

[7] EL-MOURSY, A. AND ALBONESI, D. H. 2003. Front-end policies for improved issue efficiency in SMT processors. In *he Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*. 31–40.

[8] EYERMAN, S. AND EECKHOUT, L. 2007. A memory-level parallelism aware fetch policy for SMT processors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 240–249.

[9] HILY, S. AND SEZNEC, A. 1996. Branch prediction and simultaneous multithreading. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques*. 169–173.

[10] JIMÉNEZ, D. A. AND LIN, C. 2002. Composite confidence estimators for enhanced speculation control. Tech. Rep. TR-02-14, Dept. of Computer Sciences, The University of Texas at Austin. Jan.

[11] KALLA, R., SINHAROY, B., AND TENDLER, J. M. 2004. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro 24*, 2, 40–47.

[12] KANG, D. AND GAUDIOT, J.-L. 2004. Speculation control for simultaneous multithreading. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. 76–85.

[13] LO, J. L., EMER, J. S., LEVY, H. M., STAMM, R. L., TULLSEN, D. M., AND EGGERS, S. J. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst. 15*, 3, 322–354.

[14] LUO, K., FRANKLIN, M., MUKHERJEE, S. S., AND SEZNEC, A. 2001. Boosting SMT performance by speculation control. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*.

[15] LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 164–171.

[16] PELLEG, D. AND MOORE, A. 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*. 727–734.

[17] SAZEIDES, Y., KUMAR, R., TULLSEN, D. M., AND CONSTANTINOU, T. 2005. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Comput. Archit. Lett. 4*, 1, 1.

[18] SEZNEC, A. 2005. Analysis of the O-GEometric History Length branch predictor. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 394–405.

[19] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[20] TULLSEN, D. M. AND BROWN, J. A. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 318–327.

[21] TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. Philadelphia, PA, 191–202.

[22] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 392–304.

[23] VANDIERENDONCK, H. AND SEZNEC, A. 2007. Fetch gating control through speculative instruction window weighting. In *2nd HiPEAC Conference*. 120–135.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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