A Simulator for SMT Architectures: Evaluating Instruction Cache Topologies

Ronaldo Gonçalves[1*], Eduard Ayguadé[2], Mateo Valero[2], Philippe Navaux[3]

[1] Departamento de Informática, Universidade Estadual de Maringá
Avenida Colombo 5790, Maringá, Brazil
{ronaldo@din.uem.br}

[2] Departament d'Arquitectura de Computadors[†], Universitat Politècnica de Catalunya
Jordi Girona 1-3, Barcelona, Spain
{eduard, mateo@ac.upc.es}

[3] Instituto de Informática, Universidade Federal do Rio Grande do Sul
Avenida Bento Gonçalves 9500, Porto Alegre, Brazil
{navaux@inf.ufrgs.br}

Abstract
SMT (Simultaneous MultiThreaded) is becoming one of the major trends in the design of future generations of microarchitectures. Its ability to exploit both intra- and inter-thread parallelism makes it possible to exploit the potential ILP (Instruction-level parallelism) that will be offered by future processor designs. SMT architectures can hide high latencies of instructions taking better advantage of the hardware resources through the simultaneous execution of a lot of diversified instructions from different threads. In order to provide detailed and accurate information about the performance of this approach, a SMT simulator has been developed on top of the SimpleScalar Tool Set.

The SMT simulator allows the configuration of a large set of architectural parameters (cache and reservation station topologies, number of slots and branch prediction accuracy) in addition to the parameters originally inherited from the basic simulator (size of the cache memories, tables and queues, instruction scheduling policy and pipeline width). The SMT simulator has been exhaustively tested with workloads composed of some SPEC95 benchmarks and under different instruction cache topologies. The simulator has proved to be an efficient tool for the performance evaluation of these kind of architectures. The paper describes the main features of this simulator and analyses the simulation results.

Keywords— **superscalar, SMT, performance evaluation**

I. INTRODUCTION.

The technological advance in microprocessor design in the recent years has followed two main trends. The first one tries to increase the microprocessor clock frequency using new digital components and modern VLSI solutions. The second one tries to exploit parallelism at the level of instructions by applying more aggressive techniques to exploit the *instruction level parallelism* (ILP): pipelining, superscalar out-of-order execution and simultaneous multithreading.

Pipelining consists in dividing the execution of an instruction in a set of ordered and synchronized stages, which can be operated in parallel. Using this approach, the instructions are executed in subsequent steps, allowing their partial overlapping. The ideal performance for this technique is the retirement of one instruction per cycle. Although it could be interesting to have as much stages as possible (with the subsequent reduction in the cycle time), the penalties incurred by pipeline hazards [HEN 94] forced current designs to have a moderate number of stages.

To boost the performance of the pipelining technique, superscalar architectures [JOH 91, SMI 95] replicate some hardware resources (such as registers and functional units). Superscalar execution allows the initiation of more that one instruction per cycle. Although this approach is used in current commercial microprocessors, such as Pentium [AND 95], Power/PowerPC [CHA 94, DIE 95, YOU 96], MIPS R10000 [MIP 95] and UltraSparc [ULT 96], its performance is limited by instruction dependencies [JOU 89, BUT 91, TRA 92, WAL 93].

Since the dataflow dependences limits the maximum parallelism that can be reached by single-thread applications [LIP 96], some researchers have investigated the simultaneous execution of several instruction flows [HIR 92, YAM 94, TUL 95, GON 98]. This new approach is called SMT (Simultaneous MultiThreaded) and its advantage is based on two main reasons: First, the ILP from

[*] PhD student at the II/UFRGS, supported by CAPES.
[†] Supported by the Spanish Ministry of Education (TIC98-511).

the different threads can be combined if there is no communication among them (for instance, threads come from independent applications). Second, the availability of threads can hide the execution of high-latency instructions in other threads (like for instance memory misses).

The SMT is a modern concept and just a few of performance evaluation studies have been conducted until now. In 1996, Tullsen [TUL 96] analyzed different instruction fetch policies and concluded that the fetch unit must favor those threads with less instructions in the pipeline. In 1998, Hily [HIL 98] analyzed the contention on the secondary cache and concluded that it can not be ignored in order to obtain accurate results from the simulations. Still in 1998, Lo [LO98] concluded that many cache conflicts could be eliminated by software using a suitable policy of virtual-physical address mapping for memory pages and by using per-thread memory address offset. In 1999, Gonçalves [GON 99] showed analytically that cache miss rate, caused by interference among threads, could be reduced by prefetching instruction, before that the thread is scheduled to context switching. Also in 1999, Sigmund [SIG 99] concluded that the choice of the cache replacement policy is fundamental when there are restrictions in the memory bandwidth.

All previous studies have shown the importance of the SMT architectures, analyzing and evaluating some aspects. However, much more must be done. The existence of an efficient SMT simulator that models the whole system (including memory hierarchy) is totally necessary. This paper presents a SMT simulator that has been developed on top of the SimpleScalar Tool Set [BUR 97]. Our simulator is portable and provides many facilities to obtain detailed statistical information about the performance of this new architectural approach, under different configurations. This paper describes the main features of the simulator and analyses the simulation results.

The paper is organized as follows. Section II describes the basic simulator. Section III describes the SMT simulator. Section IV presents and analyses the results of the cache simulations. The conclusions are presented in the section V and the references can be found in the last section.

II. BASE SIMULATOR.

The SimpleScalar Tool Set has been developed at the University of Wisconsin-Madison as part of the MultiScalar Project. SimpleScalar has gained popularity and it is used as the base in the development of current execution-driven architecture simulators. It contains a lot of C functions that can be used to decode SS binaries (a variation of MIPS instruction set), simulate caches and branch predictors, besides other I/O and resource management functions. In addition, there are tools for the generation of SS binaries from C source programs. The package contains pre-compiled SS benchmarks, allowing fast testing of simulators in progress.

The SimpleScalar Tool Set also includes some basic simulators. One of them, called *sim-outorder*, simulates a superscalar architecture with branch prediction, register renaming and out-of-order execution. This simulator uses the RUU (Register Update Unit [SOH 90]) to store instructions and to control both renaming and dependencies. The RUU keeps instructions until they can be committed. This architecture has a pipeline with 6 stages: Fetch, Decode, Issue, Execution, Write-back and Commit, as shown in Figure 1. The Fetch stage fetches instructions from instruction cache (i-cache), stores them in a buffer (i-queue) and predicts branches.

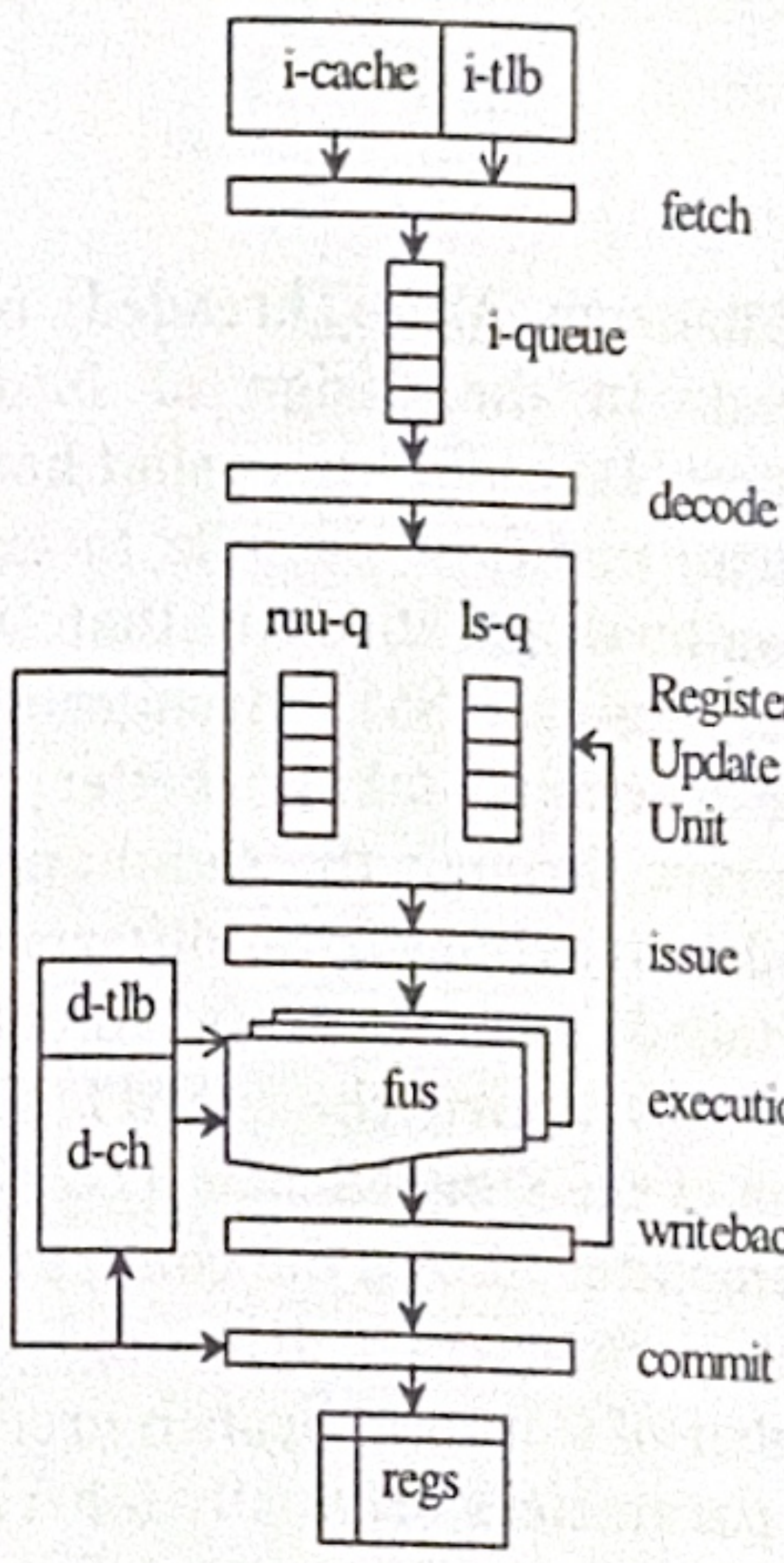

Fig.1 General view of the architecture simulated by *sim-outorder*

The virtual addresses of the instructions and data are mapped to real addresses through the instruction TLB (i-tlb) and data TLB (d-tlb), respectively. Misses in the i-cache or the i-tlb block the Fetch stage for a specific number of cycles. The number of useful fetched instructions per cycle depends on the fetch width, i-queue availability and branch prediction accuracy. The instructions available in the i-queue are decoded and renamed in the Decode stage, in order, and stored in the RUU queue (ruu-q). Load/Store instructions are split in two parts: an Add instruction, which computes the effective memory address and is stored in the ruu-q; and the Load/Store instruction itself which is stored in a load/store queue (ls-q).

Both the ruu-q and the ls-q are pools of reservation stations [TOM 67], ordered such as a reorder buffer [SMI 95]. They contain decode information, operands, busy bits and tags for the dependence controlling. The number of decoded instructions per cycle, depends on both the decode width and the availability of instructions (in the i-queue) and reservation stations (in the RUU). The Issue stage verifies which instructions from both the ruu-q and the ls-q are ready to execute (i.e. they have all the operands available and satisfied memory dependences), and issues them to appropriate functional unit. The number of issued instructions depends on the number of ready instructions, the issue width and the availability of functional units and memory ports.

The Execution stage executes the instructions and keeps each functional unit busy during the operation latency. Among the memory instructions, only Load instructions are executed in this stage. The Store instructions are executed in the Commit stage, when the computation is tagged as non-speculative. Both memory and branch instructions are executed with highest priority. The result of executed instructions are sent back to RUU in order to free the execution of other instructions which are waiting for them. The Commit stage verifies the ruu-q and retires, in order, the concluded instructions. When an Add instruction for a memory address is retired from the end of rru-q, the last entry from ls-q, which must be the other component of the memory instruction, is retired too. When a branch instruction is found, the prediction validation is done. If the prediction was wrong, all other entries of rru-q and ls-q are eliminated and the instruction fetch is redirected to the correct target. The non-speculative results are stored in registers and memory, definitively.

During each simulated cycle, all pipeline stages are executed and execution statistics are collected. The Sim-outorder simulator is totally configurable and allows the definition of L1 and L2 caches, TLB, branch predictor, as well as all the other internal parameters of the architecture mentioned in the previous paragraphs. This simulator has been used to develop the SMT simulator presented in the next section.

III. THE SMT SIMULATOR.

A simulator for SMT architectures was developed[1], as part of SEMPRE[2] Project [GON 98], using the Sim-outorder simulator. The first step in the implementation consisted in developing a multiprocessor version of Sim-outorder, through the replication of all the structures (slots). This mainly implied the expansion of scalar variables to vectors and n-dimensional data structures to (n+1)-dimensional ones. All functions were adapted to accept a new argument that corresponds to the processor identifier (slot's index). Each slot executes is devoted to execute a single application. At each cycle, all processors execute instructions from their applications in parallel. The individual results from each application are the ones reported by the original Sim-outorder simulator. The basic code of the multiprocessor is shown in the Figure 2.

```
for ( ; ; )  /* each iteration is one cycle */
{  n_cycle++;
   for (slot = 1 ; slot <= n_slots ; slot++)
   {  commit (slot, n_inst, finish);
      if (finish) break;
      writeback (slot);
      issue (slot);
      decode (slot);
      fetch (slot);
      total_inst[slot] = total_inst[slot] + n_inst;
   }
   if (finish) break;
}
for (slot = 1 ; slot <= n_slots ; slot++)
   ipc[slot] = (total_inst[slot] / n_cycle);
```

Fig.2 Simplified code for the multiprocessor simulator

After the implementation of the multiprocessor simulator, all pipeline stages were unified and many resources were shared in order to make the SMT simulator, as shown in Figure 3. Note that in this work, each thread corresponds to an independent application. The resource set containing register file, tables and queues, used to keep the context of one thread is still called slot. The new Fetch stage fetches one instruction block per cycle, which is composed of instructions from just one thread (scheduled each time in a round-robin fashion). The other stages schedule just one block of mixed instructions per cycle, which is composed of instructions from different slots in a round-robin fashion, until the corresponding bus width is filled.

There is an i-queue for each slot to ensure that each thread have its instructions fetched and also to ease the mixing of instructions inside the pipeline. The Fetch stage fetches instructions from the il1-cache, giving priority to the thread that has few instructions in the pipeline. This technique, called *ICOUNT* in [TUL 96], achieved better performance. From these fetch buffers, a lot of instructions are decoded and dispatched, in order, to reservation stations (ruu-q and ls-q). From the reservation stations, the instructions are always issued to a shared pool of functional units. Regarding registers, each slot has an individual frame to store a different context.

Many features were inherited from the original *sim-outorder* simulator, such as out-of-order and speculative execution, branch prediction and register renaming. However, new features have been developed. The first one is to control the branch prediction accuracy.

In some cases, a SMT architecture with small hardware budget usually requires a good branch predictor to

[1] Cooperative work between II/UFRGS and DAC/UPC

[2] SMT architecture that executes processes simultaneously, providing facilities to help the operating system.

efficiently exploit the potential ILP. On the other hand, having more resources relieves the system from accurate predictors. With the control of this feature, it is possible to force the accuracy rate in order to evaluate these questions. The simulator also allows the evaluation of different organizations for the reservation stations. Two different topologies have been considered (as shown in Figure 4): (a) per-thread distributed and (b) shared topologies. In that figure, both topologies receive a mix of instructions from decode stage in a SMT architecture with 4 slots (called SMT-4).

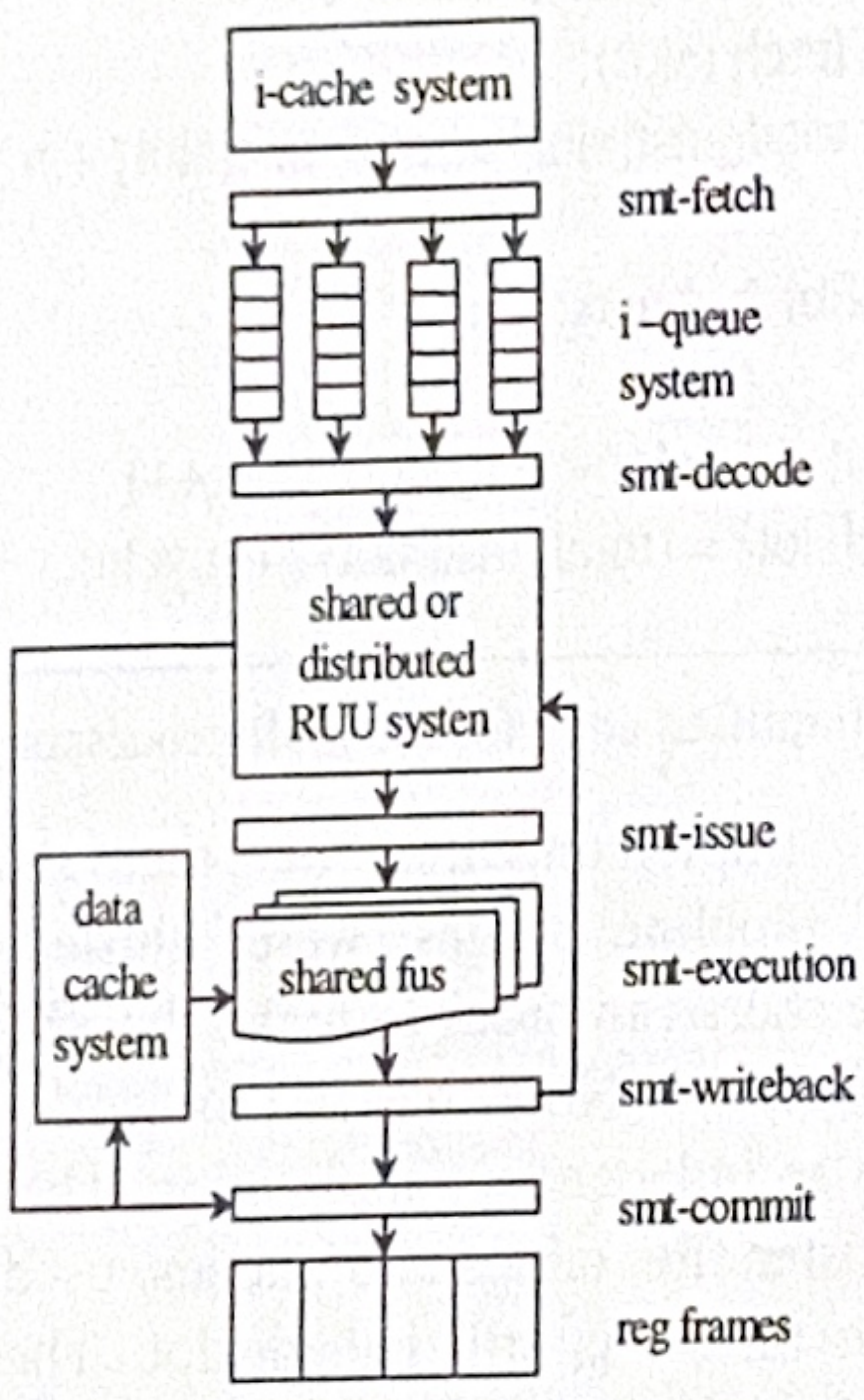

Fig.3 General view of the SMT architecture

In 1995, Jourdan [JOU 95] evaluated the performance of different issue topologies on superscalar processors, showing that the per-functional distributed topology could take better advantage of the reservation stations. In 1997, Palacharla [PAL 97] showed that the issue logic on superscalar processors could become a big bottleneck in the future. However, because of the dynamical nature of the SMT architectures, a new study of the issue topology is necessary. Depending on both the branch prediction accuracy and the cache miss rate, besides others parameters such as cache topologies and types of functional units, this feature can help deciding which is the best situation to improve the performance. [GON 00] concludes that the per-thread distributed topology can be more appropriated. In the present work we are using this topology in order to achieve maximum performance.

Another very important feature of this simulator is the ability to configure the *decode depth*, which defines the number of instructions from each fetch buffer that can be inspected simultaneously per cycle. If each fetch buffer has n entries, n is the maximum *decode depth*. Note that inspected instructions are not necessarily dispatched.

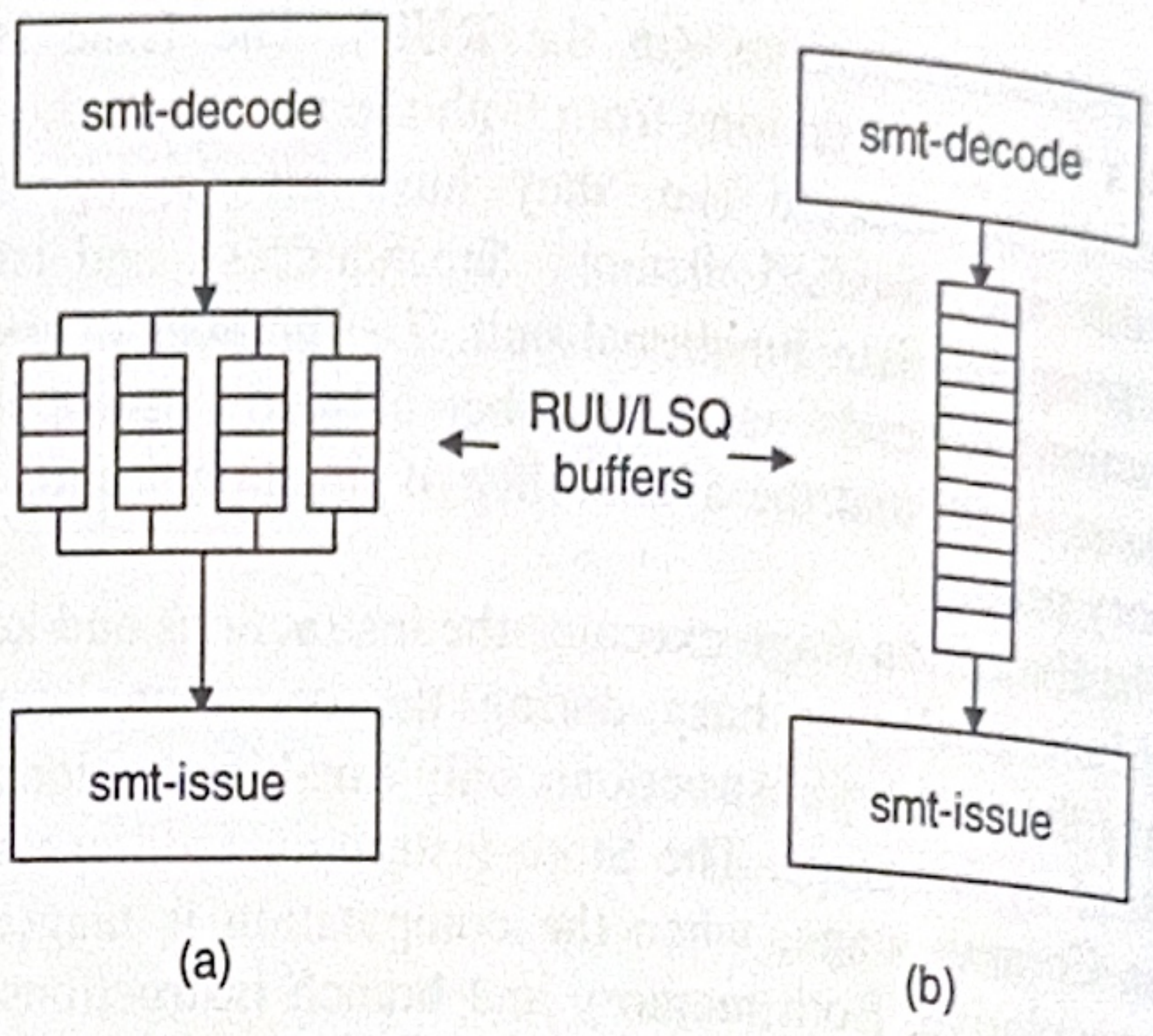

Fig.4 Issue buffer topologies

Previous studies have considered that any entry of any fetch buffer can be inspected and, if possible, decoded to fill the dispatch width. So, the decoder can make the choice of the better schedule to dispatch. However, it is required an expensive decode logic in order to implement it, and the cycle time can be increased. The availability of multiple threads makes possible to reduce the number of instructions inspected from each slot, with minimal waste of performance. As a consequence, the decoder can be simplified and the cycle time reduced. Figure 5 shows two examples of SMT-4 pipelines with *decode depth* of 1 and 2, respectively. In that figure, the first SMT-4 (a) can decode just one instruction from each instruction buffer, and the second SMT-4 (b) can decode up to two instructions. So, the first SMT-4 can inspect up to 4 instructions in order to dispatch up to 4 instructions too. However, the second SMT-4 can inspect up to 8 instructions in order to dispatch up to 4 instructions.

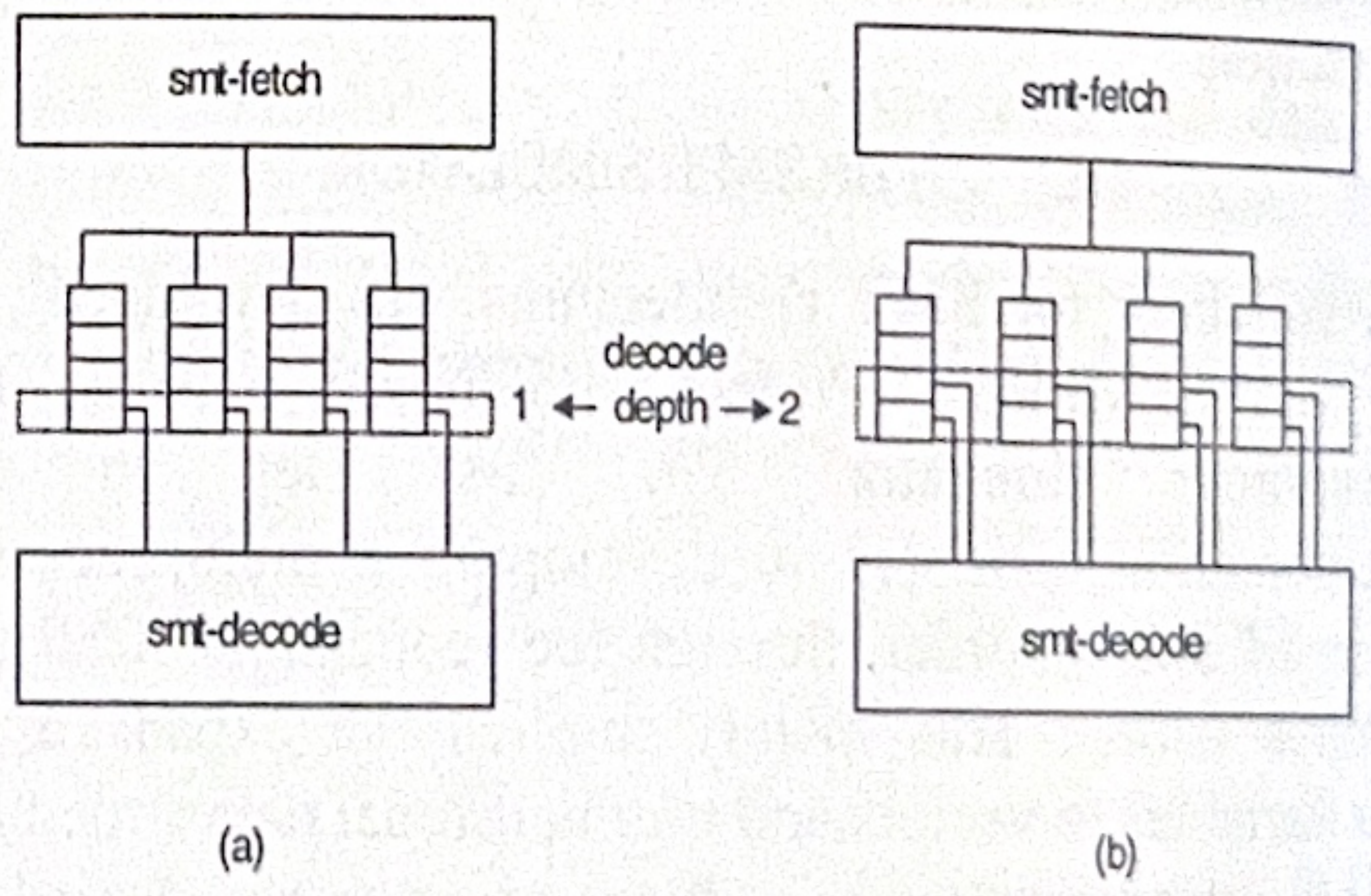

Fig.5 Examples of decode depth

In addition, if the *decode depth* is small, the dispatch width could not be filled completely due to insufficient amount of inspected instruction per cycle. This situation can happen even if there are other instructions in the fetch buffers. [GON 00] concluded that it is not necessary to decode more than 2 instructions per thread and cycle to achieve more than 96% of the best performance. In this paper we are using maximum decode depth in order to achieve to maximum performance.

This simulator also allows the definition of several memory hierarchies, which can use multiplexed banks on the same bus and modules on the independent buses, as exemplified in Figure 6 for SMT-4. There are 2 modules of instruction cache, which can be accessed in parallel. Each one serves 2 fetch buffers. Inside each module, there are 2 banks, which can be accessed exclusively. Each bank can be used by more than one thread, nevertheless, a thread must be located on just one bank. The sharing of the same cache bank is allowed because a field that identifies the slot owner (thread) was included in the cache block. In the present work we have evaluated these questions in order to certify the efficiency of this simulator.

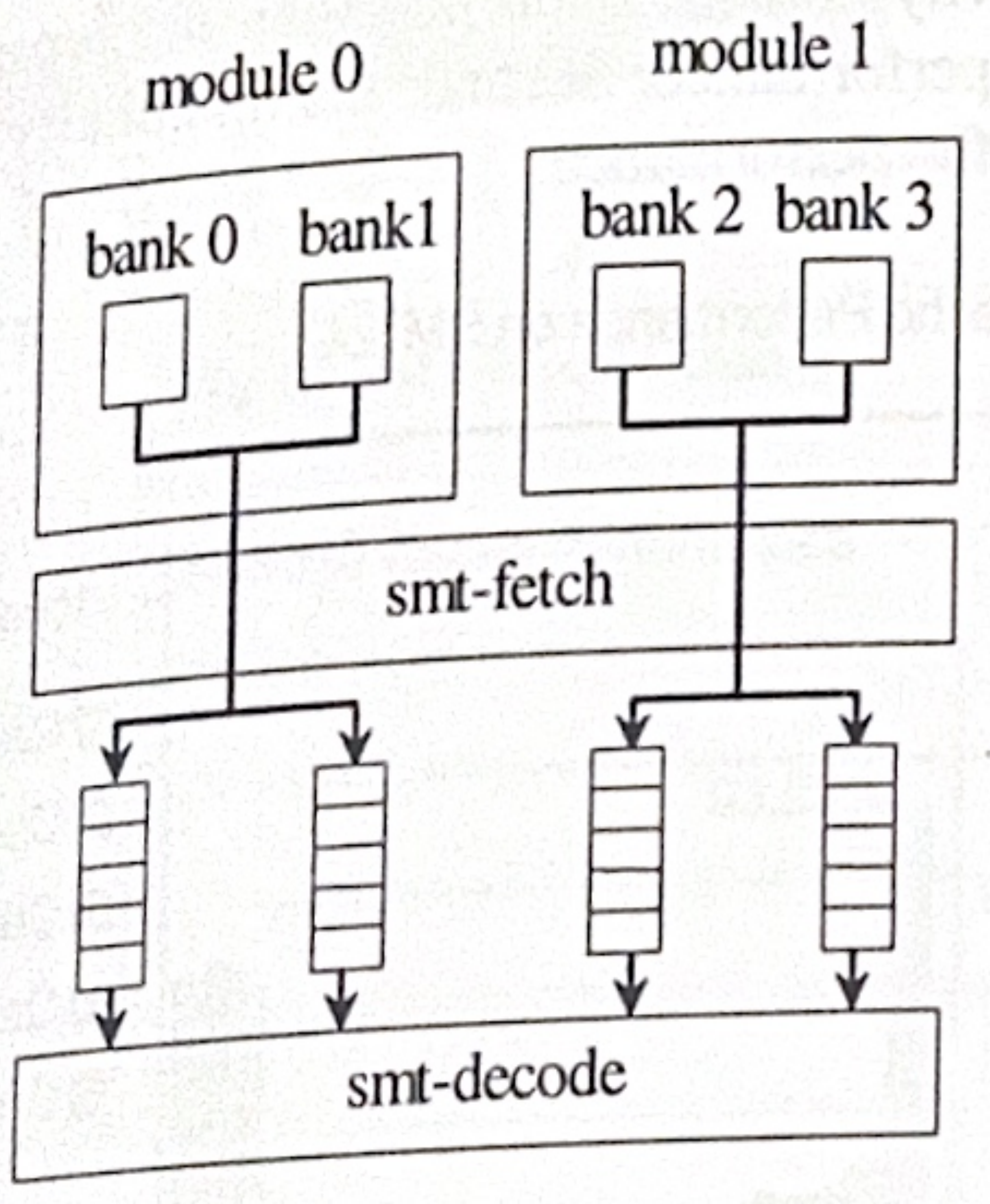

Fig.6 Example of instruction cache topology

IV. EVALUATING INSTRUCTION CACHE TOPOLOGIES.

One of the main problems related to SMT architectures is the low performance of the instruction cache [GON 99] due to memory addressing conflicts [LO 98] from different threads. In this work we have simulated different topologies for the instruction cache that try to overcome this problem.

TABLE 1
Hardware Latencies

Types of latencies	Number of cycles	
l1 hit ; l2 hit ; tlb miss	1 ; 6 ; 30	
l2 miss (for n+1 chunks)	(18 + n*2)	
int-alu functional unit	1	
fp-alu functional unit	2	
ld/st functional unit	1	
int-mul functional unit	Div oper	20
	Mult oper	3
fp-mult functional unit	Sqrt oper	24
	Div oper	12
	Mult oper	4

The experimental evaluation in this section is carried out for two processor configurations, named SMT-4 and SMT-8, with 4 and 8 slots, respectively. Table 1 shows the latencies considered for the functional units and Table 2 the total hardware amount used in the two configurations.

TABLE 2
Total Hardware Amount

SMT-4	SMT-8
Pipeline width = 8, Unif-l2-cache = 128k, Instr-l1-cache = 16k, Data-l1-cache = 16k, 7 Funct-units = (2 int-alu, 2 fp-alu, 1 int-mult, 1 fp-mult, 1 ld/st), (ruu, lsq) sizes = (16, 8) entries.	Pipeline width = 16, Unif-l2-cache = 256k, Instr-l1-cache = 32k, Data-l1-cache = 32k, 14 Funct-units = (4 int-alu, 4 fp-alu, 2 int-mult, 2 fp-mult, 2 ld/st), (ruu, lsq) sizes = (32, 16) entries.

Different memory topologies are considered, called *CacheXYZ*. X is the *Cache Modularity* that defines the number of modules connected on different buses, which can be accessed in parallel. Y is the *Cache Separativity* that defines the total number of multiplexed banks, distributed among the modules. Z is the *Cache Associativity* that defines the number of entries of each cache bank related to the same memory address. The product $Y{\cdot}Z$ (separativity times associativity) is called *ST* (*space of threads*). In order to provide sufficient space for the co-existence of many threads in the cache, the ST must be greater than the total number of threads sharing it. Also, the maximum number of threads located in the same module must be ST/X. Inside the module, the multiplexing of the banks simplifies the external hardware complexity, making possible the use of just one fetch bus. However, the multiplexing forces that just one bank can be accessed per cycle. Figures 7 and 8 show the cache topologies, which have been simulated on the SMT-4 and SMT-8 architectures, respectively. For the two configurations, all the proposed topologies use the same hardware amount, as shown in the Table 2. Also, when the cache is distributed into two modules, the total bus width is split into two parts.

In our analysis we have measured the performance in terms of *ipc* (*instructions per cycle*) and the ratio between two *ipc* values (*Speed-up*). We have used eight programs from the SPEC95 suite: 4 integer benchmarks (*perl*, *ijpeg*, *gcc* and *li*) and 4 floating-point benchmarks (*swim*, *mgrid*, *wave5* and *fpppp*). For the simulation of the SMT-4 configuration, 4 different workloads composed of 4 benchmarks each (2 integer and 2 floating-point) are used. For the simulation of the SMT-8 configuration, a single workload composed of the eight programs is used. Table 3 summarizes the composition of the workloads. All simulations are executed until one of the benchmark in the workload completes 250 millions of instructions, from which 50 millions of initial instructions are skipped to reduce the warm-up stage.

TABLE 3

Benchmarks Arrangements

SMT-4	1	swim, perl, mgrid, ijpeg
	2	wave5, gcc, fpppp, li
	3	li, fpppp, ijpeg, mgrid
	4	gcc, wave5, perl, swim
SMT-8		swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li

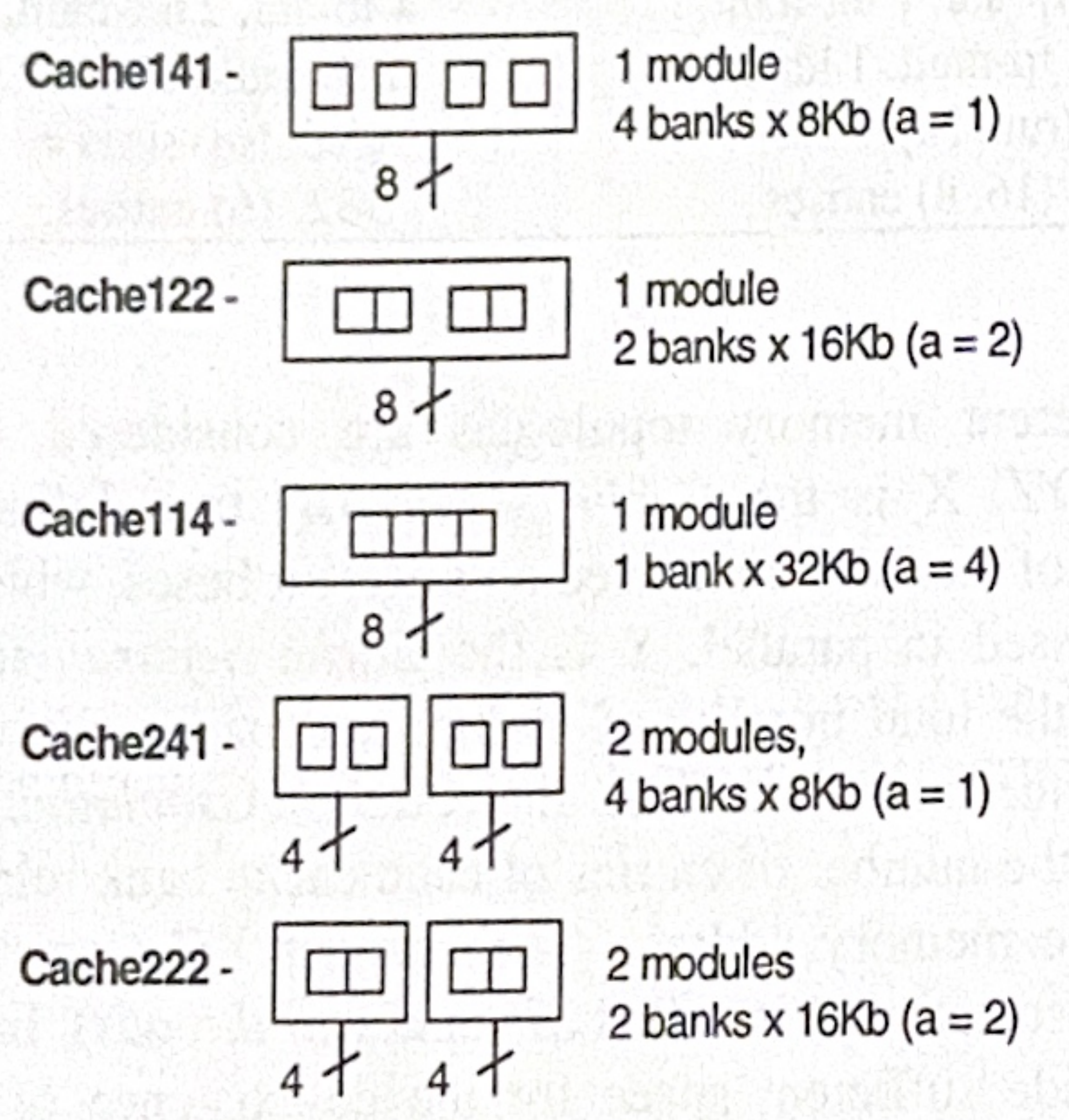

Fig.7 Instruction cache topologies for SMT-4

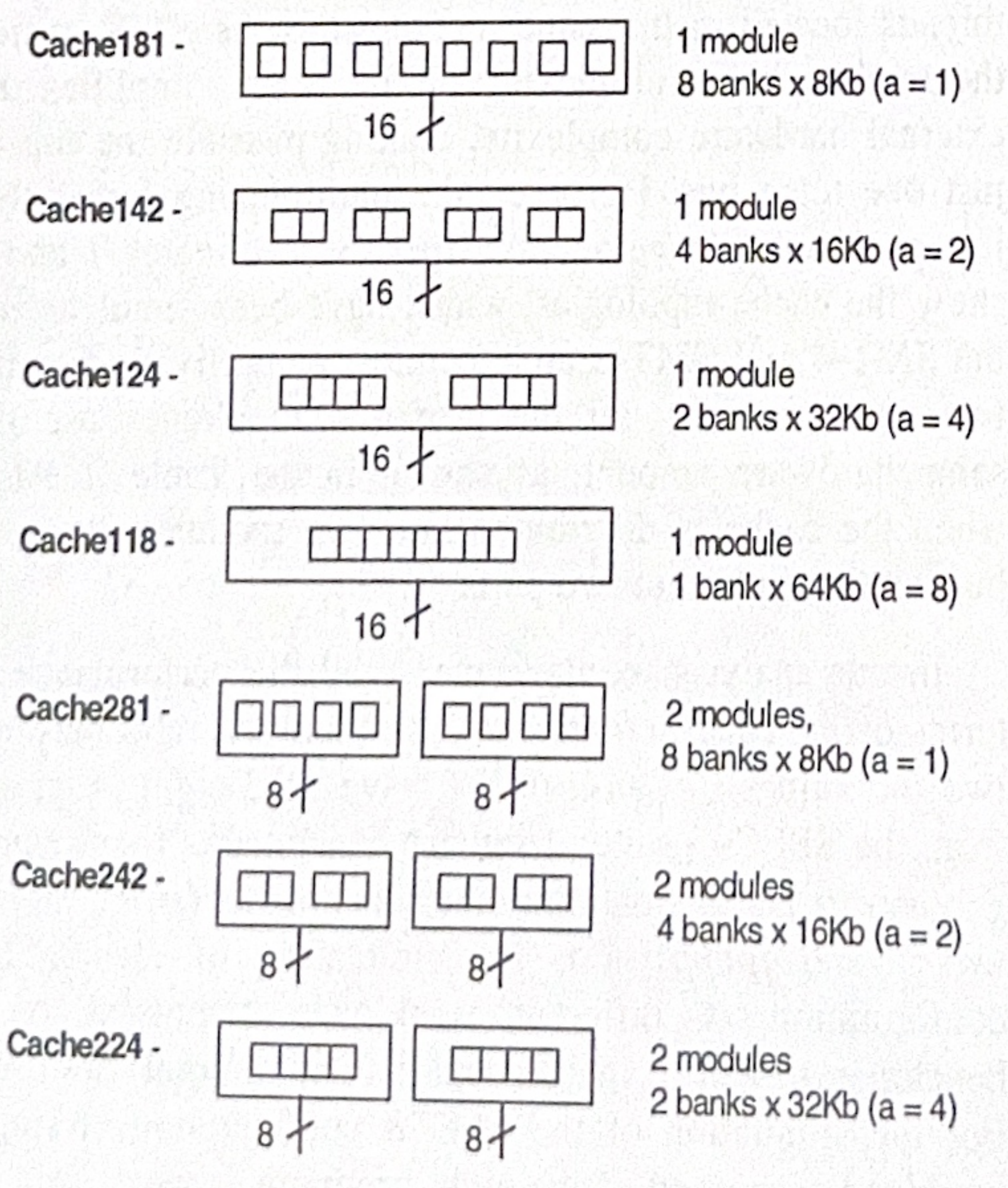

Fig.8 Instruction cache topologies for SMT-8

Figure 9 shows the performance achieved by the SMT-4 architecture for the cache topologies previously defined. In that figure it is possible to see that *ijpeg* achieves the best individual performance (over 1 ipc) and *perl* achieves the worst individual performance (close to 0.5 ipc). Notice that the overall performance reaches more than 3 ipc for all topologies. Also, we can note that changing the topology do not cause a significant difference in the individual performance of each benchmark. Regarding the overall performance, this difference is higher, as shown in the Table 4. This table shows that the speed-up of the *Cache114* topology (best overall performance) over the *Cache241* topology (worst overall performance) reaches 9.48%. Moreover, in that figure, we can make two considerations about the vectorial space of threads. First, the associativity is more important than the separativity to improve the performance. Second, the modularity of 2 does not improve the performance.

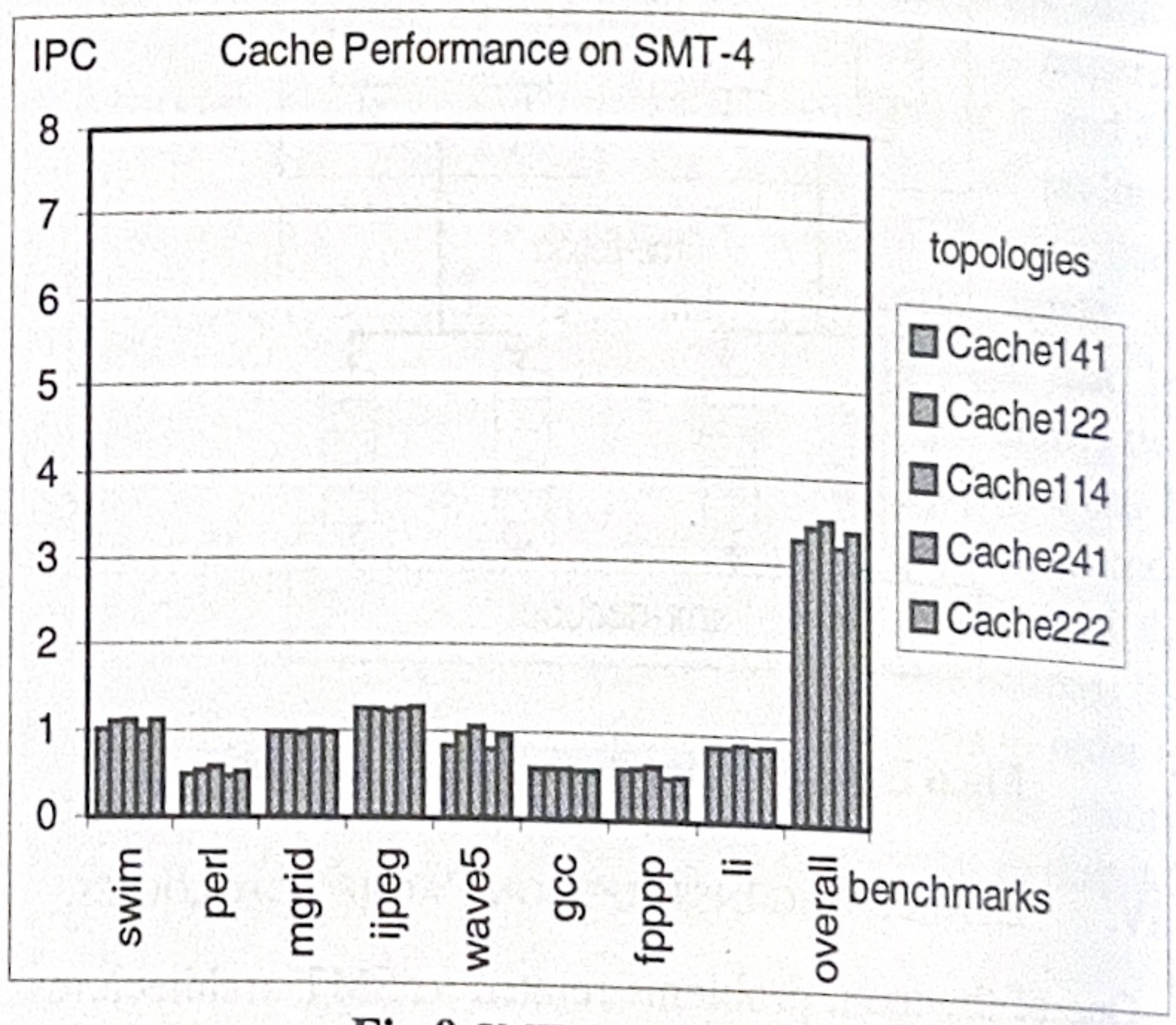

Fig.9 SMT-4 performance

Figure 10 shows the results obtained for the SMT-8 architecture. The overall performance is much higher than that obtained with the SMT-4 architecture, reaching more than 7 *ipc* for both *Cache242* and *Cache224* topologies. The best speed-up is achieved by the *Cache224* topology over the *Cache181* reaching 34.74%, as shown in the Table 4. However, the average performance of each benchmark is similar on both SMT-4 and SMT-8 architectures, as shown in Figure 11. This situation has occurred because the pipeline stages of these architectures are based on a round-robin algorithm, which gives priority for the inter-thread parallelism instead of the intra-thread parallelism. The differences in the average performance between SMT-4 and SMT-8 are 1.62%, 1.31% and 0.34%, respectively, for *perl*, *mgrid* and *ijpeg*, as shown in the Figure 11. The maximum speed-up reaches about 10.70% in swim on SMT-4. The best speed-ups between SMT-4 and SMT-8 are obtained with the floating-point benchmarks *wave*, *fpppp* and *swim*, reaching from 8% to 11%.

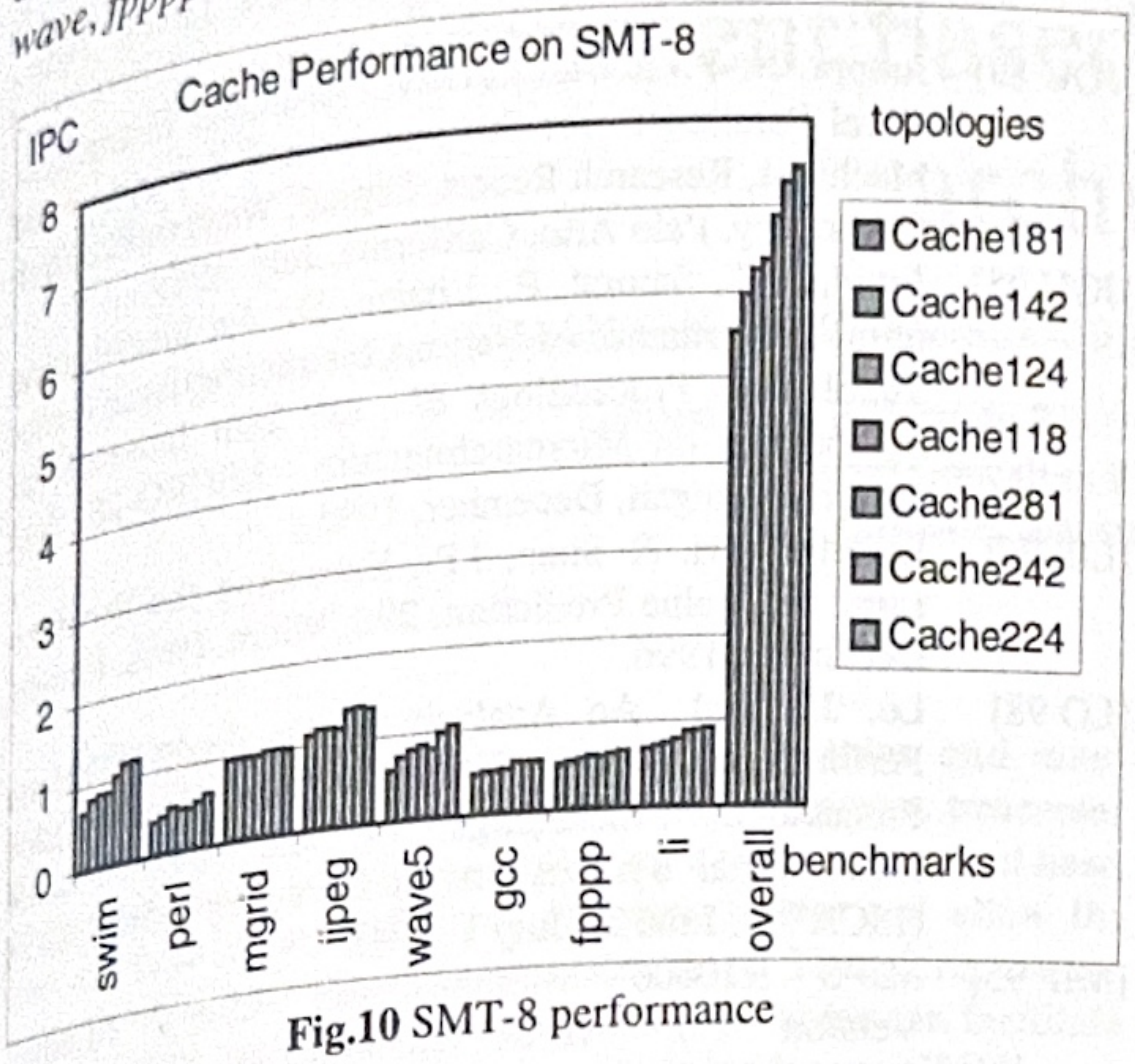

Fig.10 SMT-8 performance

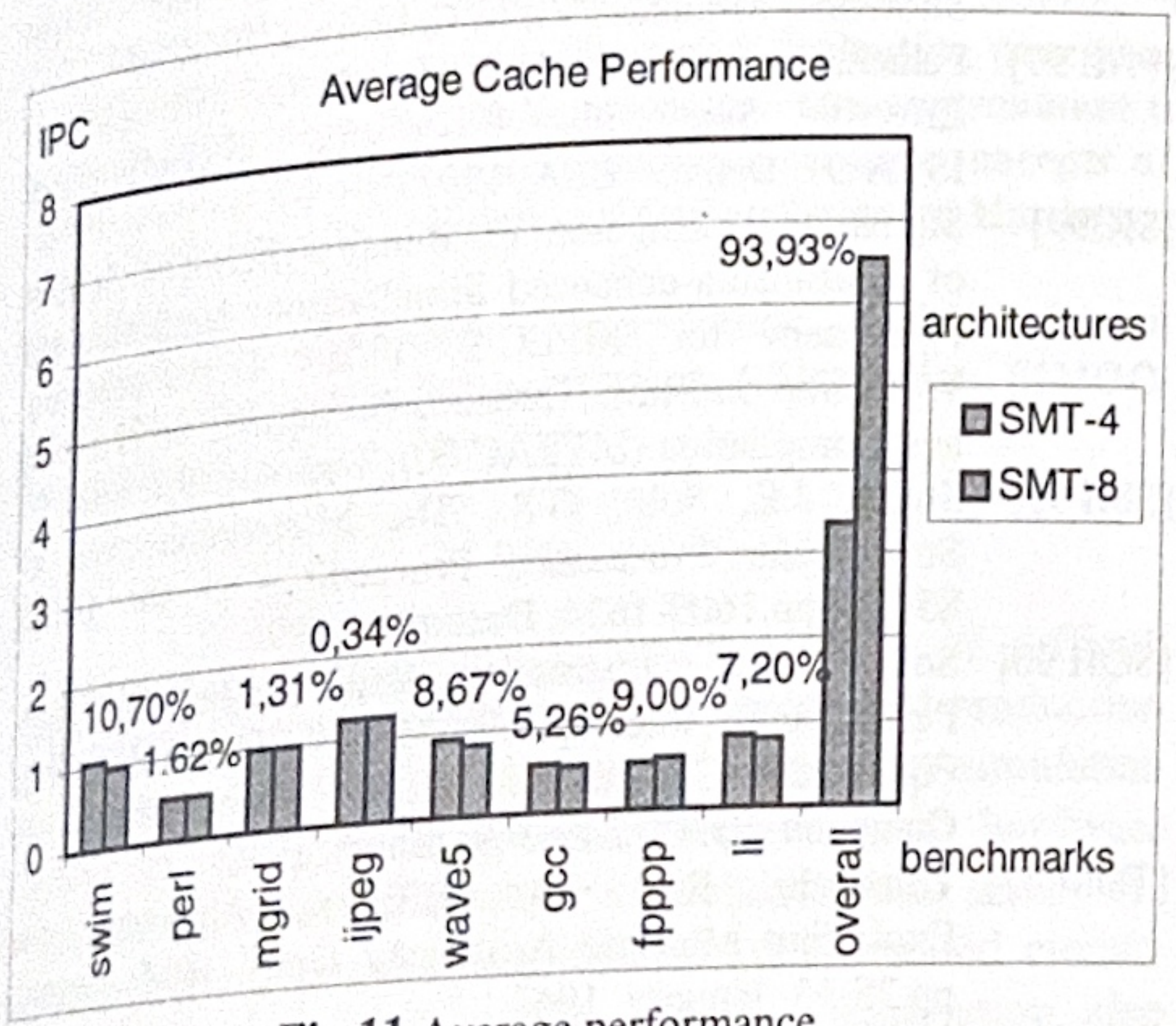

Fig.11 Average performance

These simulations allow us to verify that the associativity is more important than the separativity to improve the performance on SMT-8, just like the SMT-4. This fact has happened because the separativity does not allow the use of the banks by any thread, in opposite to the associativity. However, in opposite to the SMT-4, the utilization of modularity 2 on the SMT-8 provides better performance. This situation has happened due to fetch width. In the SMT-4 architecture, a fetch width of 4 instructions is not sufficient to explore the ILP available inside each thread, due to breaking of the basic blocks. This has not happened on SMT-8 because the use of a fetch width of 8 instructions ensures the fetching of a greater number of complete basic blocks.

TABLE 4
Maximum Performance Speed-up

SMT-4	SMT-8
Cache114/Cache241	Cache224/Cache181
9,48%	34,74%

We can not ignore the tradeoff between performance and complexity when evaluating associativity, separativity and modularity. High associativity is very expensive to implement in terms of cycle time, because it is necessary to verify a large number of cache positions in order to find the target address. However, this technique can reduce conflicts and make better use of the cache. On the other hand, the separativity provides fast access to the target bank through the multiplexing and reduces conflicts too. However, the under-utilized banks can not be used for other threads.

Regarding the modularity, the utilization of more than 1 module requires the duplication of the fetch stage. Moreover, the under-utilized fetch buffers by a module can not be used by another module. However, due to simplicity of each fetch sub-stage and the absence of conflicts among modules, this approach could be used to improve performance in some cases. We believe that combined solution of these concepts could be better.

A final consideration about our simulations it that our threads come from independent applications. Consequently, there is no communication and synchronization among them. This topic is very important but out-of-scope of this paper, in which we have been interested in the use of multithreading to achieve better workload performance.

V. Conclusions

Research activities around SMT architectures are expanding widely due to potential benefits that can be derived from multithreaded applications. From our point of view, it is important to analyze and evaluate all pipeline components in order to design efficient SMT architectures. The SMT simulator described in this paper is the key tool that allows the analysis and performance evaluation of different configurations for these SMT architectures, easing the design phase of aggressive microprocessors. The functionality and capabilities of the simulator have been tested through the evaluation of several instruction cache topologies.

Several conclusions have been drawn from our study. First, as expected, an SMT-8 architecture can provide better overall performance than SMT-4 architecture. However, the individual performance of each benchmark is kept equivalent on both architectures.

Second, the utilization of a *space of threads* smaller than the number of active threads on SMT architecture can significantly decrease performance or make impossible the execution of applications. This situation happens due to memory addressing conflicts, which enlarge the amount of cache misses. Sometimes, the processor is not able to execute useful instructions, making just cache replacement.

Third, despite of hardware complexity, cache associativity can contribute to better performance more than the cache separativity. In addition, the modularity can be good to improve the performance, depending on the fetch width.

A final remark is that when the cache topologies are organized appropriately, they can provide a reasonable speed-up, reaching more than 9% on SMT-4 and more than 30% on SMT-8, using SPEC95 benchmarks.

REFERENCES

[AND 95] Anderson, D. & Shanley, T., Pentium Processor System Architecture, Second Edition, MindShare, Inc., Addison-Wesley, Massachusetts, 433p., February, 1995.

[BUR 97] Burger, D., Austin, T. M., The SimpleScalar Tool Set, Version 2.0, Technical Report #1342, University of Wisconsin – Madison, June, 1997.

[BUT 91] Butler, M., et all, Single Instruction Stream Parallelism Is Greater Than Two, Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May, 1991.

[CHA 94] Chakravarty, D. & Cannon, C., PowerPC: Concepts, Architecture, and Design, J. Ranade Workstations Series, McGraw-Hill, USA, Inc.,p.363, 1994.

[DIE 95] Diep, T.A, Nelson, C., Shen, J.P., Performance Evaluation of the PowerPC 620 Microarchitecture, Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June, 1995.

[GON 98] Gonçalves, R. A. L., Navaux, P. O. A., SEMPRE: Superscalar Architecture with Multiple Processes in Execution (in portuguese), X SBAC-PAD, Búzios, Brazil, September, 1998.

[GON 99] Gonçalves, R. A. L., Sagula, R. L., Divério, T. A., Navaux, P. O. A., Process Prefetching for a Simultaneous Multithredead Architecture, SBAC-PAD'99 (11st Symposium on Computer Architecture and High Performance Computing), Natal, Brasil, Sept/October, 1999.

[GON 00] Gonçalves, R. A. L., Ayguadé, E., Valero, M., Navaux, P. O. A., Performance Evaluation of Issue Topology and Decode Depth on Simultaneous Multithreaded Architectures, Technical Report, UFRGS, Brazil, April, 2000.

[HEN 94] Hennessy, J., Patterson, D. A., Computer Architecture: A Quantitative Approach., San Mateo, CA: Morgan Kaufmann, 1994.

[HIL98] Hily, S., Seznec, A., Standart Memory Hierarchy Does Not Fit Simultaneous Multithreading, Proceedings of the Multithreaded Execution, Architecture and Compilation - MTEAC, 1998.

[HIR 92] Hirata, H. Et al, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, Proceedings of the 19th Annual International Symposium on Computer Architecture, ACM & IEEE-CS, pp.136-145, May, 1992.

[JOH 91] Johnson, M., Superscalar Microprocessor Design, Prentice Hall Series in Innovative Technology, PTR Prentice Hall, Englewood Cliffs, New Jersey, 288p., 1991.

[JOU 89] Jouppi, N. P. & Wall, D. W., Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, Research Report, Digital, Western Research Laboratory, Palo Alto, California, July, 1989.

[JOU 95] Jourdan, S., Sainrat, P., Litaize, D., An Investigation of the Performance of Various Instruction-Issue Buffer Topologies, Proceedings of the 28th International Symposium on Microarchitecture - MICRO-28, Ann Arbor, Michigan, December, 1995.

[LIP 96] Lipasti, M.H. & Shen, J.P., Exceeding the Dataflow Limit via Value Prediction, 29th Micro, Paris, France, December, 1996.

[LO 98] Lo, J. et al., An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors, Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98), June 29-July 1, 1998.

[MIP 95] MIPS R10000 Microprocessor User's Manual, Version 1.0, MIPS Technologies, Inc. North Shoreline, Mountain View, California, June, 1995.

[PAL 97] Palacharla, S., Jouppi, N. P., Smith, J. E., Complexity-Effective Superscalar Processors, Proceedings of ISCA'97, Denver, USA, 1997.

[SIG99] Sigmund, U., Ungerer, T., Memory Hierarchy Studies of Multimedia-enhanced Simultaneous Multithreaded Processors for MPEC-2 Video Decompression. Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC 00), Toulouse, 8.1.2000

[SMI 95] Smith, J.E, Sohi, G.S., The Microarchitecture of SuperScalar Processors, Proceedings of the IEEE, 83(12), pp.1609-1624, December, 1995.

[SOH 90] Sohi, G. S., Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers, IEEE Transactions on Computers, 39(3):349-369, March, 1990.

[TOM 67] Tomasulo, R.M., An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal, pp.25-33, January, 1967.

[TUL 95] Tullsen, D. M., et all, Simultaneous Multithreading: Maximizing On-Chip Parallelism, Proceedings of the ISCA'95, Santa Margherita Ligure, Italy, Computer Architetcure News, n.2, v.23, 1995.

[ULT 96] UltraSPARC User's Manual, UltraSPARC-I/UltraSPARC-II, Revision 2.0, Sun Microsystems, Mountain View, CA, USA, May, 1996.

[WAL 93] Wall, D. W., Limits of Instruction-Level Parallelism, Research Report, Digital, Western Research Laboratory, Palo Alto, California, June, 1993.

[YAM 94] Yamamoto, W. et all, Performance Estimation of Multistreamed, Superscalar Processors, Proceedings of the Hawaii International Conference on Systems Sciences, January, 1994.

[YOU 96] Young, J. L., Por Dentro do Power PC, Editora Berkeley Brasil, 313p., São Paulo, 1996.