Process Prefetching for a Simultaneous Multithreaded Architecture*

Ronaldo A. L. Gonçalves †, Rafael L. Sagula ‡, Tiarajú A. Diverio, Philippe O. A. Navaux §

Federal University of Rio Grande do Sul
Informatics Institute and PPGC/UFRGS
Po. Box 15064 - 91501-970
Porto Alegre, RS, Brazil
{ronaldog, sagula, diverio, navaux}@inf.ufrgs.br

Abstract—
Traditional superscalar architectures shall eventually prove incapable of taking full advantage of billions of transistors to be available in the future generations of microprocessors if they remain limited by dataflow dependencies. Thus, SMT (Simultaneous Multithreaded) architecture may be a possible solution to this problem, as far as it can fetch and execute a great deal of instruction flows and at the same time hiding both high latency operations and data dependencies. But this capability of SMT architecture depends on the existence of multithreaded applications and on some effective fetching instruction mechanism that will guarantee the presence of ready threads in the L1 i-cache to be used throughout context switching.

SEMPRE (Superscalar Execution of Multiple PRocEsses) is a type of SMT architecture which makes use of various processes to be found in today's operating systems developed to supply instructions to its SMT pipeline. This paper proposes and evaluates an effectual mechanism that prefetches instructions from awaiting processes in order to guarantee adequate context switching. An analytical model of such a mechanism was developed through using DSPN (Deterministic and Stochastic Petri Nets) and the results have shown that its use improves the dispatch width by 25% when realistic parameters are used. This method reduces the problem of cache degradation (present on many SMT architectures) and tolerates L2 delays of up to 9 cycles in some cases without the loss of performance.

Keywords—**SMT, Prefetch, Modeling.**

I. Introduction

History shows that despite technological advance on VLSI and reduction of clock cycle time, a number of other techniques has been developed to increase the performance of computers [PAT 90]. One of the most promising proposals is the simultaneous multithreaded architecture (SMT), which can execute a lot of instructions from different streams simultaneously, maximizing the hardware utilization.

Although SMT turned out to be a very good technique its development has been held back by 3 well-known problems: 1) the lack of multithreaded applications; 2) the increase of i-cache degradation as the number of threads raises ([TUL 95], [GUL 96]); and 3) volume of hardware implementation.

This architecture was designed to execute multiple processes instead of multiple threads, by advantage being taken of existent parallelism among different applications, which is greater than the one among threads of the same application. The existence of communication among applications is far more rare than among threads. So, the problem number 1 is solved by replacing threads by processes.

Consequently, a single SEMPRE processor can substitute for many conventional processors, making possible its utilization by different users through terminals. Such possibility is expected to warrant financial investments on the SEMPRE design. Moreover, the high hardware volume required to develop this kind of architecture will not pose any problem, once the next generation of integrated circuits sets billions of transistors on a chip. Therefore, in a near future, problem number 3 will naturally work itself out.

This architecture also provides hardware support to process scheduling, and a new set of instruction that allows the operating system to manage processes with minimal CPU overhead. So, the waste time with both scheduling and context switching among processes will be insignificant, enhancing the performance of the whole system.

The main goal of this work is to propose and evaluate a new prefetch mechanism called "Process Prefetching" aiming at working out problem number 2. This mechanism makes use of SEMPRE specific features, and conceals the degradation of the L1 i-cache through prefetching instructions of various processes before they are needed. The proposed mechanism is evaluated as well as compared with a similar SMT architecture which does not prefetch instructions.

Section II presents an overview of related works about SMT and cache degradation. Section III describes the main parts of SEMPRE architecture and details the proposed prefetch mechanism. Section IV introduces briefly the use of Analytical Modeling and shows the models used in this work. Sections V and VI show the results and conclusions, respectively.

II. Related Works

The main object of a multithreaded architecture is to maximize processor utilization at the occurrence of high latency

*Work supported by CNPq and CAPES
† Phd. Student at PPGC/UFRGS; Professor at DIN/UEM
‡ Msc. Student at PPGC/UFRGS
§ Advisors at PPGC/UFRGS

operations, like those caused by i-cache misses or data dependencies ([LAU 94]). Unfortunately, such latencies can only be concealed if there are enough instructions available from other threads previously stored on L1 i-cache during the context switching.

Laudon, Gupta e Horowitz ([LAU 94]) had proposed a technique for multithreaded execution called Interleaving, that could be applied on traditional superscalar processors to allow the execution of monothreaded as well as multithreaded applications, without enlarging the hardware. Otherwise, Govindarajan e Nemawarkar ([GOV 92]) designed a multiprocessor called SMALL that was composed by many independent processing units.

The major part of multithreaded architectures is based on replication of both instruction paths and storage structures from conventional superscalars in order to support multithreading. Hirata et al. ([HIR 92]) developed a SMT that has multiple register banks, instruction queues and program counters for each thread. Also, Wallace, Calder e Tullsen ([WAL 98]) designed a SMT architecture with several register renaming tables to support the execution of both threads and paths simultaneously.

There are two approaches for multithreaded execution: concurrent ([TSA 96]) and simultaneous ([TUL 95]). The former allows interleaving of execution of different applications like the one made by operating systems, so just one process runs in the pipeline at a time. That technique hides high latency operations but doesn't maximize the utilization of functional units, once each application has specific features and requires functional units of different types more intensively. The second approach hides high latency operations and maximizes the hardware utilization by simultaneously dispatching instructions from different threads. This one is known as Simultaneous Multithreaded (SMT).

A lot of research is done on this field ([SIG 96], [AKK 98], [GOO 98], [HIL 98], [LO 98]) and many of them ([AGA 92], [CRA 93] e [THE 94]) have shown that during simultaneous multithreaded execution the instruction cache suffers so much degradation due to reduction of locality. On the experiments made by Tullsen [TUL 95], the i-cache conflicts were the most dominant factors to the occurrence of wasted cycles, for example, increasing the number of threads' from 1 to 8, the L1 i-cache miss rate increased from 1% to 14%. In subsequent work, Tullsen ([TUL 96]) concluded that significant performance could be reached using a suitable fetch policy. Even so, among all architectural parameters used in that experiment, the fetch width could be the main bottleneck of the SMT architecture.

Another important observation was made by Lo ([LO 98]). He showed that there are two kind of cache interferences: destructive interference, which occurs when the instructions/data from a thread are changed by the ones from another thread, increasing i-cache miss rate; and constructive interference, which occurs when the same instructions/data are required by many threads, reducing bus load.

Using a multithreaded processor, Gulati ([GUL 96]) showed that increasing the threads' number from 1 to 6 the average cache hit rate decreases from 97.33% to 76.21%, that implies in an increase of 21.7% on the cache miss rate.

Nemirovsky ([NEM 98]) also showed that when the number of streams increases the cache interference raises too reaching 200% in the worst case for 4 streams.

The main question about cache issue with SMT is whether there is enough parallelism to increase the occupancy of the processor resources ([BUR 99]) or not. Thus, the fetch unit can be a bottleneck. Even if the processor uses many memory levels, Rinker ([RIN 98]) showed that cache miss rate could be reduced by 1% only. In some cases, cache degradation could reach 50% of processor idleness. SEMPRE architecture can reduce these problems according to next sections.

III. SEMPRE Architecture

The SEMPRE architecture (Superscalar Execution of Multiple PRocEsses) was proposed in ([GON 98], [GON 98a]) and is similar to a traditional SMT, but it executes processes instead of threads. It adds new instructions that implement tasks usually done by the operating system. Its superscalar pipeline, showed in the Figure 1, contains 5 main stages (fetch, decode, execute, finish and conclusion) and provides usual techniques such as branch prediction (speculative fetch), simple register renaming and out-of-order execution for ready instructions.

The architecture has multiple slots to store fetched instructions from different processes. These processes are scheduled from FP (waiting process queue) that keeps a descriptor (identification, program counter, time-slice, status and so on) for every process waiting to be scheduled. Inside of each slot there is a RDP register, that contains both program counter and time-slice information for one process, and a FI queue (for fetched instructions). Also, there is a register bank (frame) to keep the contexts for each process created in the architecture. In to order to simplify its implementation, the i-cache is indexed by the real address, thus the translation of virtual to real address is done before the i-cache access. The functional units are shared by ready instructions which can be dispatched in-order from slots.

During instruction fetching, just one line is fetched from i-cache per cycle for each slot, in a round-robin way. After a slot has been selected, the fetching can be done using the program counter contained in the respective RDP register and the instructions are inserted in the respective FI. When it is necessary to switch context for a slot, other process must be scheduled from FP and the respective RDP must be updated.

The old descriptor must remain in the pipeline (in FA or FT queues) until the last instruction has been concluded, coming back to the FP queue. Usually, context switching shall be

done in three different situations: 1) on the occurrence of i-cache misses; 2) on the detection of one of the new instructions; and 3) after process time-slice has been finished. The new instructions are: create, kill, suspend, resume and run-now.

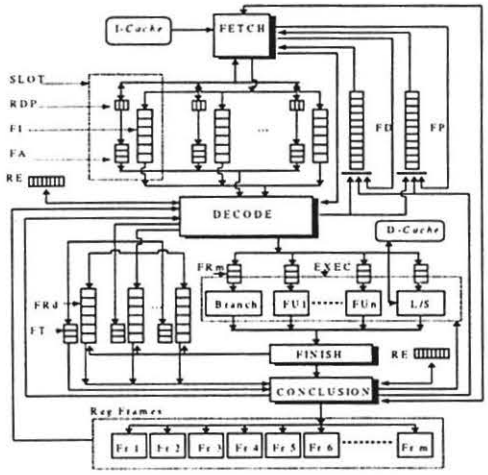

Fig. 1. Overview of SEMPRE Architecture

The decode stage executes 3 main activities (not necessarily in one single cycle): instruction scheduling from slots, decoding of those instructions (including simple renaming) and dispatching to issue buffers (FRm). The instructions located at the end of the slots (last instructions) must be decoded and just the ready instructions (without dependencies) are scheduled to be dispatched. This decision was taken because the out-of-order execution of not ready instructions in SMT couldn't be advantageous ([HIL 98]) besides to dirty the issue buffers with long latency instructions, but after the development of simulator other alternatives can be evaluated.

In the execution stage, each functional unit executes the instructions placed on its issue buffer. The instructions are removed in-order from FRm and after that the results are sent to finish stage which updates the correct entry of the reorder buffer (FRd) setting the "finished" tag. The conclusion stage performs an in-order checking of the last entry from each FRd and removes "finished" instructions from it. At this time, the results of those instructions are updated on the correct register frame. When the execution of the last instruction of a context is finished, its descriptor is removed from FT and put back in the FP queue. Also, the conclusion stage controls the accuracy of the speculative execution, exception and death of processes.

Like other SMT, one of the greatest challenges of SEMPRE project is the development of an efficient fetch mechanism, that must be able to fetch instructions from different streams and to sustain high bandwidth for the next stages of the pipeline. This mechanism is proposed in the next subsection.

A. *Prefetch Mechanism*

A simple alternative to reduce i-cache miss penalty on single-threaded processors is to use instruction prefetch mechanisms ([LEE 95]). One side effect of this kind of mechanism is the load of the bus, but the obtained speedup justifies the investments in bus bandwidth. The kind of misses present on that processors is known as intra-thread i-cache miss and a more detailed study about different prefetch policies for it was done by Tatiana ([TAT 99]). In SMT architectures, the great number of threads sharing the same cache forces another kind of miss: the inter-threads i-cache miss, that occur when a just scheduled thread causes the removal of the instructions from another thread that already was in the cache.

The i-cache misses on SMT architectures don't cause so much prejudices than on single-threaded architectures, because they can be hidden by the context switching. But, this benefice can not be possible once there are not ready thread stored in L1 i-cache during a context switching. So, it is necessary the development of a prefetch mechanism that be able to anticipate the fetching of instructions from L2 to L1 i-cache for a thread before that it is scheduled for execution.

When the scheduling is done by operating system or by another program, the inter-threads i-cache miss can't be detected and a thread can just be fetched after i-cache miss. However, when the architecture has knowledge about the scheduling policy, such as SEMPRE does, the prefetching can work according to that in order to anticipate the storing of threads in L1. This prefetch mechanism can eliminate the inter-threads interference as well as hide the intra-thread i-cache misses.

The fetch stage of the SEMPRE pipeline was redesigned to support that prefetch mechanism, as shown in Figure 2. The fetch unit transfeer instructions from L1 i-cache for those processes located in the slots, looking to the program counter in RDP registers, and putting the instructions in the FI queues. Simultaneously, the prefetch unit makes the fetching of instructions from L2 cache of those processes located in the FP queue, and puts their instructions in the L1 i-cache, looking on the program counter in the correct fields from FP. To control this mechanism, each entry of FP queue has a bit called "miss-status" that shows whether a process is or is not in the L1 i-cache.

Once a process is prefetched its miss-status flag is set to 1. Every time a process is stalled because i-cache miss it is switched, going back to FP queue. Then its miss-status flag is reset to 0. The performance of this technique depends on the availability of "prefetched" threads in the FP queue. Note that a process can be stalled by I/O or time-slice leaving its

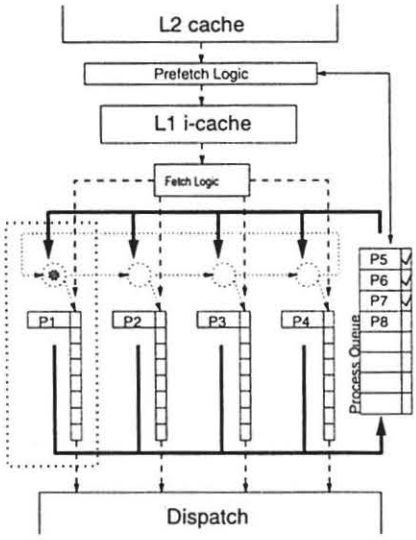

Fig. 2. Simplified Scheme of Prefetch Model

miss-status flag unchanged. The SEMPRE architecture uses round-robin scheduling algorithm but that prefetch technique isn't limited to it. Also, in the Figure 2, there is a token ring linking all slots only to represent the round-robin scheduling and to help on the good understanding of the analytical model.

A full simulator for SEMPRE architecture is being developed but will be available only at the beginning of the next year. So, to make easy its implementation as well as predict its behavior the fetch unit was modeled analytically. That is described in the next section.

IV. Analytical Modeling

Usually, two techniques can be used to estimate the performance of a computer architecture before its implementation ([JAI 91], [SAG 99]): computational simulation and analytical modeling. The former is the most widely used technique, once it allows the implementation of the model with more details, but it takes too much programming and computing time to implement and produce useful results. On the other hand, the analytical models can be implemented and solved in much less time, but their results are not so accurate as those ones obtained by simulation. That happens because the over simplification of the models.

The research activity on analytical modeling field is quite intensive and many computer systems (and architectures) are being modeled and evaluated using such methodology ([MAR 84], [SAA 90], [YAM 94], [JAC 96] e [KAN 97], [CAR 97], [COU 91], [GUN 98], [ROB 94], [SAH 96], [SIL 92]). In [YAM 94], the results obtained by an analytical model differs only by 4% of the ones obtained by conventional simulation showing that even simple models can produce good results.

In this work, a tool called DSPNExpress [LIN 98] was used to create and solve the analytical models of the fetch and prefetch mechanisms. The mathematical formalism used was the DSPN (Deterministic and Stochastic Petri Net) which is a kind of extension of the original Petri nets ([PET 62], [LIN 98]). Its application to model and evaluate performance of computer architectures has been successfull in many other works ([SAA 90], [LIN 98], [MOR 98], [SAG 99a]).

The modeled prefetch scheme works on the waiting processes queue prefetching their instructions from the L2 cache to the L1 i-cache independently of the L2 cache-miss delay. Note that intra-thread i-cache misses continue to occur, but it guarantees the hiding of such delay by the prefetching of another process. The created models and the obtained results are shown on the next section.

A. Models

Owing to all slots being dependent upon the same prerequisites for utilization, they will maintain an equivalent behavior, which fact makes easy the modelling of the prefetch mechanism, as only a generic model for one slot must be developed. So, the joint behavior of the prefetch and fetch mechanisms can be reached by resolving that simple model. The reduced model of the prefetch mechanism is presented in Figure 3, That model if composed of 3 essential parts: 1) the left most part that represents the complete model of one slot; 2) the right-most part, that represents all the other slots of the architecture; and 3) the FP queue (or "process queue"), which is located in the middle of the model and interacts with all slots (left and right sides of the model). Figure 4 details the left-most part of this model.

In Figure 3 there is no instructions of the waiting processes in L1 i-cache in the initial state. Thus, there is an initial loss of cache performance, while the instructions are not fetched. When those instructions are fetched, the process initiates its execution on the slot until the occurrence of an i-cache miss. After that, the missed process goes back to the "process queue" and waits for the prefetch mechanism. Note that, in this mechanism, L1 i-cache doesn't care about requesting the missed lines to the L2 cache, because this work will be done by the prefetch mechanism.

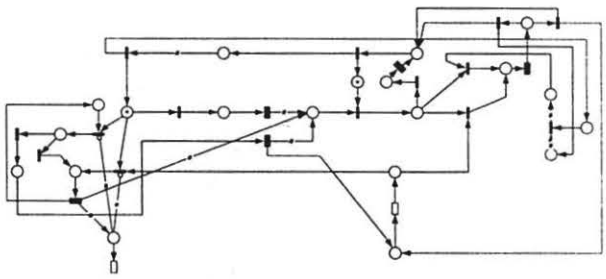

Fig. 3. Petri Net of the Fetch Mechanism

In this paper, we consider "dispatch throughput" as the number of instructions available for being dispatched at each cycle, that means the number of instructions present on the last positions of the slots' instruction queue and ready for dispatching. Due to the main goal of this work is to evaluate the performance of the proposed prefetch mechanism, the dispatch throughput can be used as a good performance metric. The stages after the fetching were not modeled because they are the same for all the studied cases ("best", "SEMPRE" and "no-prefetch") and it's not the goal of this paper to measure the IPC of that architecture.

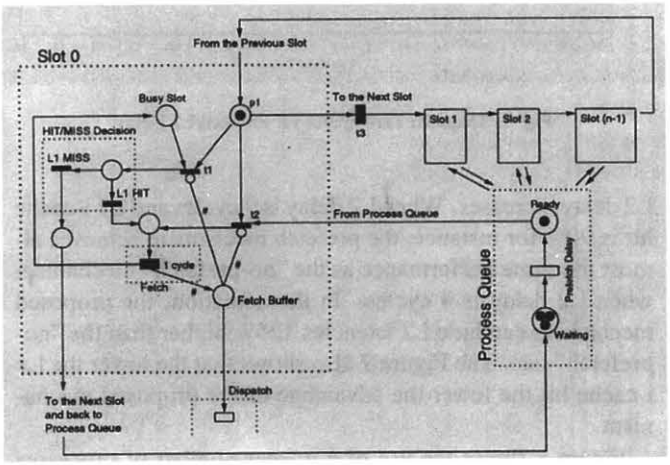

Fig. 4. DSPN Model of One Slot

In Figure 4 it is shown the complete model for one slot. Place *p1* represents the marked place of Figure 2 and controls the round-robin fetch mechanism. Only one slot has its *p1* place marked at a time, indicating the slot that is being fetched at that cycle. If place *p1* is marked and the slot is busy (place *"Busy Slot"* marked), then transition *t1* fires.

The region labeled *HIT/MISS Decision* simulates the behavior of the L1 i-cache. Both transitions *L1 MISS* and *L1 HIT* are enabled at the same time, but only one is fired.

If the requested line is present in the L1 i-cache, the token enables the *Fetch* transition and after 1 cycle the corresponding *Fetch Buffer* is fulfilled with the fetched instructions. Then this slot is marked as "busy" (*Busy Slot*=1) and a token is sent to the next slot. If a L1 i-cache miss occurs then after 1 cycle a token is sent to the "process queue" and another one is sent to the next slot, leaving this slot "idle" (*Busy Slot*=0).

If the slot is idle and there is at least one process ready to be scheduled on the *Process Queue*, then transition *t2* is fired and the slot is filled with that process (token). After that, the usual fetch mechanism takes place. If there are no ready processes, the transition *t3* is fired and the fetch control is sent to the next slot (at the next cycle).

The processes waiting on the *Process Queue* are represented by tokens on the *Waiting* place and they have to wait for as many cycles as is the L2 cache delay plus 1. That is the delay of the prefetch mechanism as stated on Section III.

The region labeled as *Dispatch* represents the next stages of the pipeline. The dispatch throughput is measured at this point. Once the behavior of all slots is the same, the total throughput of the model can be obtained by multiplicating the throughput of one slot by the number of slots.

Two other models were based on the SEMPRE model. They represent the best and worse cases and have little modifications compared to the original one. The best case is considered to the one where there are always prefetched processes waiting to execute. So the slot never waits to be filled by a new process. This model eliminates the "prefetch delay" and "inter-threads interference" effects. The case where the prefetch mechanism does not exist and the slot waits for the L1 i-cache miss resolution every time a L1 i-cache miss takes place is the worst case. In this model, the missed line is fetched on the L2 cache only when the miss event takes place, so there is no work being done on the L2 cache if there is no miss (only data accesses).

V. Results

All Figures present an index that describes the parameters used on the models. For each model, there is an 5-uple that describes it: (fetch width, #slots, L1 i-cache hit, prefetch/L2 delay, #processes). The basic configuration uses realistic parameters: (8, 8, 80%, 4, 16), which represents an architecture that fetches 8 instruction per cycle, has 8 fetch slots, L1 i-cache hit of 80%, prefetch delay of 4 cycles (3 for L2 plus 1 for prefetch logic compensation [SHA 97]); and 16 waiting processes in the process queue initially.

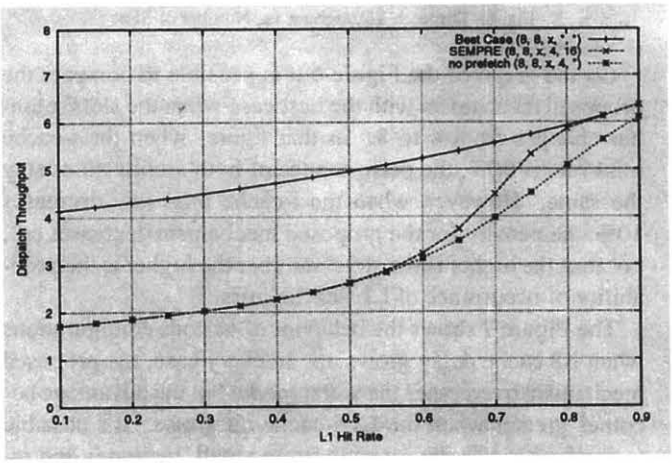

Fig. 5. Dispatch Throughput vs. Hit Rate

The behavior of the three models with the L1 i-cache hit variation is depicted in the Figure 5. When a L1 i-cache hit higher than 80%, the proposed prefetch mechanism takes the same dispatch throughput as the best case. It happens because the hit rate is not sufficient to overcome the throughput

of the prefetch mechanism, so the number of missed processes per cycle is lower than the number of prefetched processes. That keeps the process queue always with 1 or more prefetched processes.

When the L1 i-cache hit is too low, the proposed mechanism behaves just like the worst case. However, such hit rates (lower than 55%) are not so usual and can be easily solved improving the L1 i-cache configuration. Obviously, the best case overcomes the other ones in more than 100%, because context switching does not effect this model.

The best performance obtained by the proposed method is achieved when the L1 i-cache hit is between 70% and 85%, which corresponds to a gain of 9% to 18% over the worst case. Those are satisfactory results, once low i-cache hit rates (like those obtained on the experiments) are expected when the number of threads grows ([TUL 95], [TUL 96]).

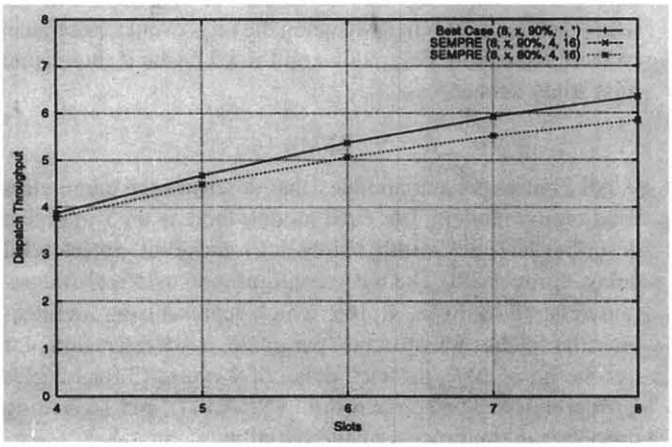

Fig. 6. Dispatch Throughput vs. Number of Slots

On the graph of the Figure 6 it is possible to compare the proposed mechanism with the best case when the slots' number changes from 4 to 8. In that figure, when the i-cache miss rate is 90%, the performance of both architectures stay the same. However, when the i-cache miss rate decreases 10%, the penalty for the proposed mechanism decreases too, because the higher is the slots' number the higher is the probability of occurrence of L1 i-cache miss.

The Figure 7 shows the behavior of various configurations when L2 cache delay grows up. In every case, the proposed mechanism overcomes the worst model but the advantage becomes greater when the L1 i-cache hit grows. It's possible to verify that relative speedup begin small, increases and reduces again. The best performance is achieved when the L2 delay is 12 cycles and the L1 i-cache hit is 90%. In that case, the dispatch throughput for the proposed mechanism is 25% greater than the "no-prefetch" mechanism.

The graphs show the prefetch mechanism keeps the throughput high even with high L2 delays, while the performance of the "no-prefetch" mechanism decreases when the

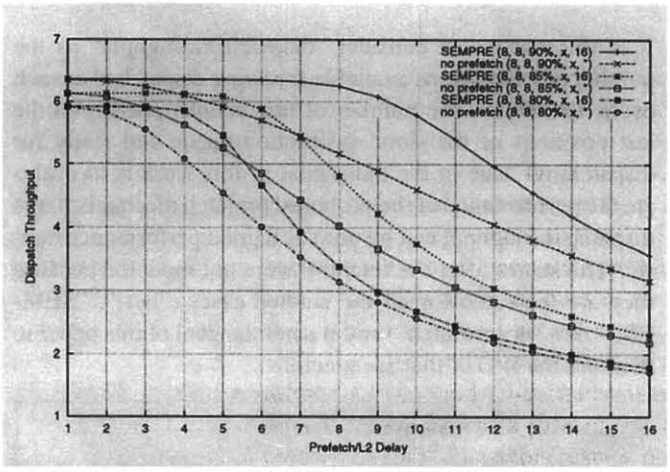

Fig. 7. Dispatch Throughput vs. Prefetch/L2 Delay

L2 delay increases. When L2 delay is 9 cycles and L1 i-cache hit is 90% for instance, the prefetch mechanism achieves almost the same performance as the "no-prefetch" mechanism when L2 delay is 4 cycles. In that situation, the proposed mechanism can hide L2 latencies 125% higher than the "no-prefetch" one. The Figure 7 also shows that the lower the L1 i-cache hit the lower the advantage of the proposed mechanism.

Figure 8 shows the use of a greater number of processes guarantees that a fetch width of 8 instruction on the proposed mechanism behaves as the best case for an architecture with 8 slots.

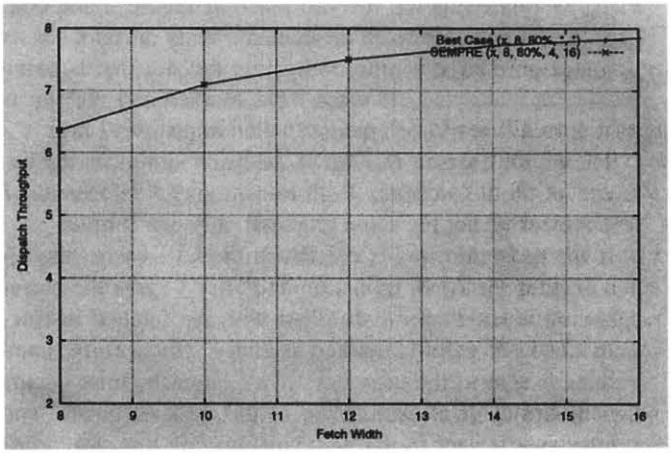

Fig. 8. Dispatch Throughput vs. Fetch Width

VI. Conclusions

This work proposes a prefetch mechanism for a simultaneous multithreaded architecture called SEMPRE that has additional skills to schedule and execute processes instead of threads. This mechanism was modeled analytically and the

results are shown.

That modeling allows us to conclude that the variation of both fetch width and slots' number don't have so much influence on the performance of the proposed prefetch mechanism. Nevertheless, the architectural parameters that most constrains the performance of that simultaneous multithreaded architecture are the L1 i-cache hit rate as well as the L2 delay. These techniques produce good results, but should be seen just as tools for the behavioral analysis of the modeled system.

The proposed mechanism proves to be efficient handling those problems and improves the dispatch width by 25% in some cases. Also, it tolerates L2 delays up to 9 cycles when L1 i-cache hit rate is 90% with the same performance of a similar architecture with L2 delay of 4 cycles, that is an advantage of 125%.

The results show that the proposed prefetch mechanism improves the performance of a SMT fetch mechanism that uses a round-robin algorithm. However, in future works other scheduling algorithms must be evaluated.

ACKNOWLEDGMENTS

The authors of this paper would like to thank Brazilian financial agencies CAPES and CNPq which have supported this work.

REFERENCES

[AGA 92] *Performance Tradeoffs in Multithreaded Processors* IEEE Transactions on Parallel and Distributed Systems, 3(5):525-539, September, 1992.

[AKK 98] Akkary, H.; Driscoll, M. A.: *A Dynamic Multithreading Processor* Proceedings of the MICRO-31: ACM/IEEE International Symposium on Microarchitecture, Dallas, Texas, December, 1998.

[BUR 99] Burns, J.; Gaudiot J.-L.: *Exploring the SMT Fetch Bottleneck* Proceedings of the MTEAC'99 (in conjunction with HPCA-5), Orlando, Florida, 1999.

[CAR 97] Carvalho, L.: *Uma Ferramenta para Modelagem de Sistemas de Comunicação, Computação e Confiabilidade.* Msc. Thesis, COPPE/UFRJ, Brazil, 1997.

[COU 91] Couvillion, J.; Freire, R.; Johnson, R.; Obal II, W.D.; Qureshi, M.A.; Rai, M.; Sanders, W.H.; Tvedt, J.E.: *Performability Modeling with UltraSAN* IEEE Software, vol. 8, no. 5, Sept. 1991, pp. 69-80.

[CRA 93] McCrackin, D.C.: *The Synergistic Effect of Thread Scheduling and Caching in Multithreaded Computers* COMPCON Spring, pages 157-164, 1993.

[GON 98] Goncalves, R. A. L.; Navaux, P. O. A.: *SEMPRE: Uma Arquitetura SuperEscalar com Multiplos PRocessos em Execucao* Anais, X SBAC-PAD, Buzios, Brazil, Sep, 1998.

[GON 98a] Gonçalves, R. A. L.; Navaux, P. O. A.: *Proposta de uma Arquitetura Multi-Threading Voltada para Sistemas Multi-Processos* IV Congresso Argentino de Ciencia da Computação - CACIC'98, Neuquen, Argentina, Oct, 1998.

[GOO 98] Goosens, B. T.: *The Threads Processor* Proceedings of the MTEAC'98, Workshop on Multithreaded Execution, Architecture and Compilation: held in conjunction with HPCA-4, Las Vegas, Nevada, February, 1998.

[GOV 92] Govindarajan, R.; Nemawarkar, S. S.: *SMALL: A Scalable Multithreaded Architecture to Exploit Large Locality* Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, Dec, 1992.

[GUL 96] Gulati, M.; Bagherzadeh, N.: *Performance Study of a Multithreaded Superscalar Microprocessor* Proceedings of the HPCA-2, California, February, 1996.

[GUN 98] Gunther, N. J.: *The Practical Performance Analyst: Performance-by-Design Techniques for Distributed Systems* McGraw-Hill, 1998.

[EGG 97] EGGERS, Susan J. et al: *Simultaneous Multithreading: A Platform for Next-Generation Processors* IEEE Micro, V.17, n.5, Sep/Oct 1997.

[HIL 98] Hily, S.; Seznec, A.: *Out-of-Order Execution May Not Be Cost-Effctive on Processors featuring Simultaneous Multithreading* IRISA (Institut de Recherche en Informatique et Systemes Alatoires, Publication Interne 1179, March, 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, 1992.

[JAC 96] Jacob, B. L.; Chen, P. M.; Silverman, S. R.; Mudge, T. N.: *An Analytical Model for Designing Memory Hierarchies* IEEE Transactions on Computer, Vol. 45, No. 10, Oct/1996.

[JAI 91] Jain, R.: *The Art of Computer Systems Performance Analysis* John Wiley and Sons, New York, 1991.

[KAN 97] Kant, L.; Sanders, W.H.: *Analysis of the Distribution of Consecutive Cell Losses in an ATM Switch Using Stochastic Activity Networks* Special Issue of International Journal of Computer Systems Science & Engineering on ATM Switching, vol. 12, no. 2, March 1997, pp. 117-129.

[LAU 94] Laudon, J. et al: *Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations* Proceedings of the International Conference on ASPLOS, Oct, 1994.

[LEE 95] Lee, D. et al: *Instruction Cache Fetch Policies for Speculative Execution* Proceedings of the 22th International Symposium on Computer Architecture (ISCA'22), Italy, 1995.

[LIN 98] Lindemann, C.: *Performance Modeling with Deterministic and Stochastic Petri Nets* John Wiley and Sons, 1998.

[LO 98] Lo, J.L. et al: *An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors* Proceedings of the 25th Annual International Symposium on Computer Architecture, June, 1998.

[MAR 84] Marsan, M.; Baldo, G.; Conte, G.: *A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems* ACM Transactions on Computer Systems, May 1984.

[MAR 98] Marcuello, P., Gonz lez, A.: *Control and Data Dependence Speculation in Multithreaded Processors* Proceedings of the MTEAC'98 (In conjunction with HPCA-4), Las Vegas, Nevada, February, 1998.

[MOR 98] Moreno, E. D.; Kofuji, S. T.: *Um Modelo RPDE para Busca Antecipada de Dados num Multiprocessador Baseado em um Simples Nó SMP* X SBAC-PAD. Anais. Búzios, RJ, 28-30 Sep, 1998.

[NEM 98] Nemirovsky, M., Yamamoto, W.: *Quantitative Study on Data Caches on a Multistreamed Architecture* Proceedings of the MTEAC'98 (In conjunction with HPCA-4), Las Vegas, Nevada, February, 1998.

[PAT 90] Patterson, D. A.; Hennessy, J. L.: *Computer Architecture: A Quantitative Approach* Morgan Kaufmann Publishers, 1990.

[PAR 91] Park, W.; et al: *Performance Advantages of Multithreaded Processors* Proceedings of the International Conference on Parallel Processing, 1991.

[PET 62] Petri, C. A.: *Kommunikation mit Automaten* Ph.D. Thesis, University of Bonn, Germany, 1962.

[RIN 98] Rinker, R.E, Tamma, R., Najjar, W.: *Evaluation of Cache Assisted Multithreaded Architecture* Proceedings of the MTEAC'98 (In conjunction with HPCA-4), Las Vegas, Nevada, February, 1998.

[ROB 94] Robertazzi, T. *Computer Networks and Systems: Queuing Theory and Performance Evaluation* Springer-Verlag, 1994.

[SIG 96] Sigmund, U.; Ungerer, T.: *Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor* Proceedings of the EUROPAR'96, Lyon, August, 1996.

[SAA 90] Saavedra-Barrera, R. H.; Culler, D. E.; von Eicken, T.: *Analysis of Multithreaded Architectures for Parallel Computing* 2nd Annual ACM Symposium on Parallel Algorithms and Architecture; Crete, Greece; July, 1990, pp. 169-178.

[SAG 99] Sagula, R. L.; Diverio, T.; Navaux, P. O. A.: *Modelagem Analítica: Formalismos e Ferramentas.* Trabalho Individual 789, PPGC/UFRGS, 1999.

[SAG 99a] Sagula, R. L.; Gonçalves, R. A. L.; Diverio, T. A.; Navaux, P. O. A.: *A Utilizaç ao de Modelagem Analítica no Projeto de Arquitetura de Processadores.* CLEI/PANEL'99, Assunción, PY, Sept/1999.

[SAH 96] Sahner, Robin A.; Trivedi, Kishor S.; Puliafito, A.: *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package* Kluwer Academic Publishers, 1996.

[SHA 97] Shanley, T.: *Pentium Pro and Pentium II System Architecture.* Addison-Wesley, 1997.

[SIL 92] Silva, E.; Muntz, R.: *Métodos Computacionais de Solução de Cadeias de Markov: Aplicações a Sistemas de Computação e Comunicação* VIII Escola de Computação, Gramado-RS, 1992.

[TAT 99] Serra, T.; Bampi, S.: *Mecanismos de Pré-Busca em Máquinas RISC* Trabalho Individual 829, PPGC/UFRGS, 1999.

[THE 94] Thekkath, R.; Eggers, S.J.: *The Effectiveness of Multiple Hardware Contexts* Proceedings, Sixth International Conference on Architectural Support for Programming Languages and Operating Systems pages 328-337, October, 1994.

[TSA 96] Tsai, J.-Y. & Yew; P.-C.: *The Superthreaded Architecture: Thread Pipelining with Run-Time Data* Dependence Checking and Control Speculation Proceedings of the Conference on Parallel Architectures and Compilation Techniques - PACT96, October, 1996.

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

[TUL 96] Tullsen, D.M. et al: *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor* Proceedings of the 23rd ISCA, Philadelphia, PA, May, 1996.

[WAL 98] Wallace, S.; Calder, B.; Tullsen, D. M.: *Threaded Multiple Path Execution* Proceedings of the 25th International Symposium on Computer Architecture, June, 1998.

[YAM 94] Yamamoto, W.; Serrano, M.; Talcott, A.; Wood, R.; Nemirovsky, M.: *Performance Estimation of Multistreamed, Superscalar Processors* Proceedings of the Hawaii International Conference on Systems Sciences, January, 1994.