Speculative Multithreaded Processors

Pedro Marcuello, Antonio González and Jordi Tubella
Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors
C/ Jordi Girona 1-3, Mòdul D-6
08034 Barcelona, Spain
E-mail: {pmarcue,antonio,jordit}@ac.upc.es

Abstract

In this paper we present a novel processor microarchitecture that relieves four of the most important bottlenecks of superscalar processors: the serialization imposed by true dependences, the instruction window size, the complexity of a wide issue machine and the instruction fetch bandwidth requirements. The new microarchitecture executes simultaneously multiple threads of control obtained from a single program by means of control speculation techniques that do not require any compiler/user support. In this way, it works on a large instruction window composed of multiple nonadjacent small windows. Multiple simultaneous threads execute different iterations of the same loop, which requires the same fetch bandwidth as a single thread since they share the same code. Dependences among different threads as well as the values that flow through them are speculated by means of data prediction techniques. The novel processor organization does not require any special feature in the instruction set architecture; its novel features are completely based on hardware mechanisms. The architecture is scalable in the sense that it consists of a number of processing units with separate hardware for issuing and executing instructions. The preliminary evaluation results show the potential of the new architecture to achieve a high IPC rate. For instance, a processor with 4 four-issue processing units achieves an IPC from 2.2 to 9.9 for the Spec95 benchmarks.

Keywords:

Data speculation, data dependence speculation, control speculation, dynamically scheduled processors, multithreaded processors.

1. Introduction

Several studies on the limits of the instruction-level parallelism (ILP) that current superscalar organizations can attain show that it is rather limited when a realistic configuration is considered. Four of the most important bottlenecks that cause this limitation are: the serialization imposed by data dependences, the instruction window size, the complexity of the logic required by a wide issue machine and the fetch bandwidth.

Whereas a lot of effort has been devoted to reduce the penalties caused by control and name dependences, techniques to relieve the serialization caused by data dependences (true dependences) have been practically ignored so far. Data value speculation techniques are emerging as a new family of techniques that can provide a significant boost in ILP [9][10][13][19][20][29]. Data value speculation is based on predicting either the source or destination operands of some instructions in order to execute speculatively the instructions that depend on them.

The amount of ILP that a superscalar processor can exploit is highly dependent on the size of the instruction window. However, increasing the window size poses new problems that limit its feasibility or its effectiveness. First, branch prediction accuracy limits the average window size. To go beyond a single basic block, superscalar processors rely on predicting the outcome of unresolved branches. However, this process is sequential in nature because the instruction window is composed of a contiguous region of the dynamic instruction sequence, which is called a *thread of control* (or thread for short) in this paper[1]. In consequence, a single mispredicted branch prevents the instruction window growing further until the branch is resolved. Second, the complexity and delay of the issue logic grow with the instruction window size. It has been shown in a recent study [16] that the issue and bypass logic are likely to be one of the most important hurdles to build a wide issue superscalar processor due to its impact on the clock cycle.

Finally, in addition to the branch prediction accuracy, the two main factors that limit the instruction fetch bandwidth are: the branch prediction throughput and the potential to fetch noncontiguous instructions.

In this paper, we propose a novel processor microarchitecture that relieves the four bottlenecks mentioned above.

First, the processor implements an effective large instruction window that is made up of several nonadjacent smaller windows. That is, the instructions that are in-flight in the processor at any point in time consist of several subsequences of the dynamic instruction stream such that there are instructions among consecutive subsequences that are not known (not fetched yet). The execution inside each small window uses conventional control speculation techniques whereas the creation of new small windows is based on speculating on highly predictable branches (e.g., branches that close loops). Each small window corresponds to a different thread of control of the same program. These threads, which are not necessarily independent, are created from a single sequential program completely by the hardware without compiler intervention and are executed by several thread units with distributed resources. In particular, the issue and bypass logic is local to each thread unit, which allows the processor to scale to higher issue widths by adding more thread units, without increasing the complexity of this logic and thus, without compromising the cycle time.

Second, the execution ordering constraints imposed by dependences among different threads (inter-thread dependences

[1] Regardless of the particular approach used to obtain it (i.e. the partition of a program into threads of control could be done by the hardware, as proposed in this work).

ICS 98 Melbourne (Australia)

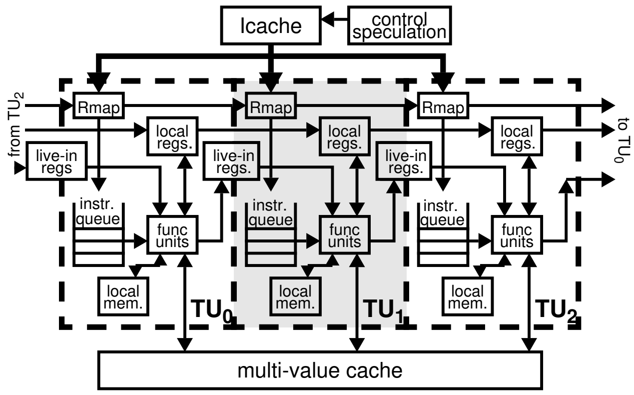

Figure 1. A speculative multithreaded processor with three thread units

for short) are avoided through the extensive use of data dependence and data value speculation. The proposed architecture speculates on both inter-thread data dependences and the data that flows through them. That is, for each new speculative thread it predicts which register and memory dependences it has with previous threads in the control flow and which values are going to flow through such dependences. The thread is then executed obeying the predicted dependences and using the predicted values to avoid waiting for the actual data.

Third, since the multiple threads of control are obtained by speculating on loop-closing branches, simultaneous active threads of control share the same code (the loop body), and thus, a simple fetch engine can feed all the threads with the same fetch bandwidth as that is required by a single thread.

The new processor microarchitecture, which is called *Speculative Multithreaded (SM)* architecture, does not require any modification in the instruction-set architecture: ordinary programs compiled for a superscalar implementation can run in this new processor architecture.

The rest of this paper is organized as follows. The SM processor microarchitecture is presented in section 2. Performance figures of the SM architecture are analyzed in section 3. Section 4 reviews the related work. Finally, section 5 summarizes the main contributions of this paper.

2. Speculative multithreaded processor microarchitecture

The microarchitecture of a *Speculative Multithreaded Processor (SM)* is shown in Figure 1. It consists of several thread units (TU) that execute concurrently different threads of a sequential program. These threads are dynamically obtained by a control speculation mechanism based on identifying loops and executing speculatively different iterations of a loop [25] (not necessarily an innermost loop) and they do not need to be independent. Thread units are interconnected through a ring topology and iterations are allocated to thread units following the execution order. Each thread unit has its own physical register file (local registers in Figure 1), register map table, instruction queue, functional units, local memory and reorder buffer in order to execute multiple instructions out-of-order. In this way, the issue bandwidth of a SM processor is scalable and the main bottlenecks observed in superscalar processors (wakeup, select and bypass logic) are avoided [16]. High issue rates can be achieved by increasing the number of thread units, which has no impact on the cycle time.

An important feature of SM processors is the aggressive use of speculation techniques. As pointed out in the introduction, data dependences are one of the main barriers for the exploitation of instruction level parallelism. This problem is addressed in SM processors through speculation mechanisms that can be classified into three categories:

- *Control speculation* is used at two levels. On the one hand it is used to obtain threads from a sequential program. On the other hand, it is used to speculate on individual branches inside each thread like superscalar processors do.
- *Data dependence speculation*: Dependences among different threads (inter-thread dependences) are predicted by means of address prediction. When a thread executes a memory instruction, it is speculatively disambiguated against other memory instructions in previous threads by using the predicted addresses of those instructions if they have not yet been executed. The memory dependence speculation scheme is implemented by means of the *multi-value cache.* Inter-thread register dependences are managed by identifying at run time which registers hold live values at the beginning of an iteration (*live-in registers*) and the number of writes that each iteration performs to them. Live-in registers whose value is not predictable are read from the *live-in register file* after being produced by the previous thread.
- *Data value speculation*. Inter-thread data dependences, either through registers or memory, do not cause a serialization between the producer and the consumer in SM processors, provided that the values that flow through such dependences are predictable. Previous works have shown that many of such values are predictable, including the results of arithmetic instructions, the values read from memory and the register values used to compute the effective address of memory instructions [9][13][19][20][29].

Another important feature of SM processors is that they may exploit a large amount of instruction level parallelism with a simple instruction fetching mechanism. This feature is based on the observation that the majority of iterations of the same loop follow the same path. In particular, we have evaluated that the most frequent path of each loop represents about 85% of the total number of iterations for the Spec95. This suggests that if thread units are devoted to execute iterations of the same loop with the same control flow, the instruction fetching mechanism can be shared by all the threads. A single fetch engine fetches a single instruction stream from the instruction cache using a conventional branch predictor. The instructions fetched in each cycle are broadcast to all the thread units where their registers are renamed using a different register map table and then, they are dispatched to their corresponding instruction queue. In this way, the processor peak performance can be equal to the actual fetch bandwidth multiplied by the number of thread units. This organization overcomes one of the most important hurdles of multithreaded architectures. In those machines, the processor is required to fetch from different program counters, simultaneously or alternatively, which makes the fetch engine to be one of the critical parts of such architectures [24]. In SM processors, instructions are always fetched from a single program counter. Each thread validates the predicted intra-thread control flow by executing branch instructions and comparing the result with the prediction. In case of misprediction, this thread and the following ones are squashed[1].

Finally, precise exceptions are supported by SM processors.

[1] Alternative ways of dealing with branch misspeculation, which are based on a more selective squashing may also be implemented but they are not considered in this paper.

Instructions of the same thread are retired in order by means of a local reorder buffer. Besides, memory values that are produced by each thread are kept in the multi-value cache and will not update the next level of the memory hierarchy until the thread is committed (i.e., until the thread becomes non speculative). Each thread unit has a *local memory* to store the predicted live memory values at the beginning of the corresponding iteration (*live-in memory values*) and also to speedup the access to data produced by itself or reused several times.

Below the main parts of the SM microarchitecture are described in more detail.

2.1 Inter-thread control speculation

A program is executed out-of-order by means of a large instruction window that consists of several non contiguous small windows, each one corresponding to a different thread of control. The multiple threads of control are built at run-time through the control speculation mechanism proposed in [25]. In this section we just outline the main features of this mechanism.

The idea of such mechanism is to identify loops at run-time and to execute concurrently several iterations of the same loop, even if they are not independent. Among all the threads that proceed in parallel at any given time, there is only one that is not control dependent on previous threads, which is called the *non speculative thread*. The remaining ones are called *speculative threads*.

Initially, there are not speculative threads. When the non speculative thread starts a new iteration of a loop, a number of speculative threads may be created and allocated to execute the following iterations. When a speculative thread reaches the closing branch of its iteration, it is suspended and waits to be either committed or squashed. When the non speculative thread finishes an iteration of a loop, all the speculative threads of this loop are squashed if the branch is not taken. Otherwise, the thread allocated to the next iteration is committed and becomes the new non speculative thread.

A small table, which is called *loop execution table*, can be used to predict the number of iterations of a loop based on previous history. A 16-entry table that implements a stride predictor has a hit ratio of 92% for the Spec95 benchmarks [25].

2.2 Data dependence and data value speculation

Inter-thread dependences (which correspond to loop carried dependences) and the values that flow through them are predicted by means of a history table that is called *loop iteration table*. Each entry of this table contains information about the last iteration of a different loop. The loop iteration table (see Figure 2) is indexed with the loop identifier (the target address of backward branches) and it contains the following fields:

- Register dependences. This field stores for each logical register the number of writes performed by the last iteration of the corresponding loop, whether it contained a live value at the beginning of the iteration, the value at the beginning, the difference between the values of the last two iterations (val_str) and a field (conf) that indicates whether such value is predictable (for instance a 2-bit saturating counter can be used to assign confidence to the predictions).
- Memory dependences. For each store, this table stores the logical register identifier that was used to compute the effective address and the offset of the effective address in relation to the initial value of the register at the beginning of the iteration. Besides, information regarding live-in memory values could be added in order to predict such values. However, this is not considered in this paper.

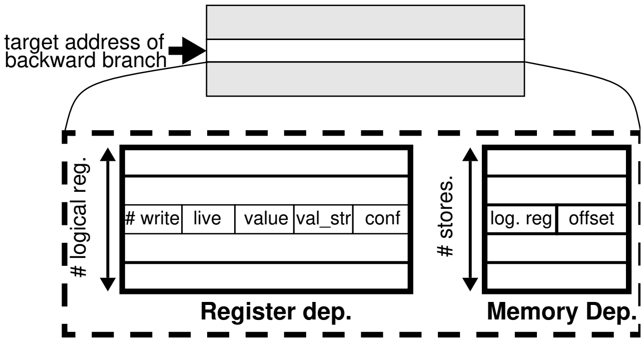

Figure 2. The loop iteration table

A new entry in this table is allocated every time that a new loop execution is started. The entry corresponding to the loop with the least recently started excution is chosen for replacement.

Each entry of the iteration table is relatively large. However, very few entries are enough to obtain significant benefits since programs usually spend large intervals of time in the same few loops. Figure 3 shows the percentage of iterations that find their history in the iteration table for a number of entries ranging from 2 to 16 averaged for the whole Spec95 benchmark suite. This percentage is 85% for 2 entries and 90% for 4 entries.

2.2.1 Dependences through registers

When a speculative thread is created, its local register file and its register map table are copied from its predecessor. Then, for each register R_i that is live and predictable, the instruction queue of the thread unit is initialized with the instruction `add R_i,R_i,stride`. These instructions are not in the static code but they are inserted in the instruction queue by the hardware. In this way, the registers are initialized with the predicted values. If a register R_i is live but not predictable, then the *i*-th entry of the register map table is set to point to the *i*-th entry of the live-in register file. This implies that the size of the live-in register file is equal to the number of logical registers. In fact, the number of live-in registers per iteration is much lower as it is shown in section 3.2 (see Figure 10). Thus, a shorter live-in register file could suffice but then, a map table would be necessary to indicate the physical register in the live-in register file allocated to each live-in logical register.

Besides, each thread unit has another table, which is called register write table, that contains for each logical register the number of remaining writes to that register. This table is initialized with the #writes field of the loop iteration table. When an instruction with destination register R_i is retired from its thread unit, the corresponding entry of the register write table is

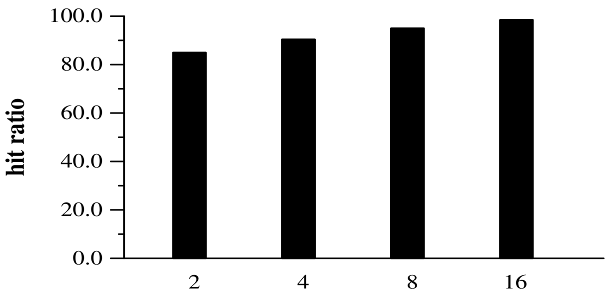

Figure 3. Hit ratio of the loop table for 2, 4, 8 and 16 entries.

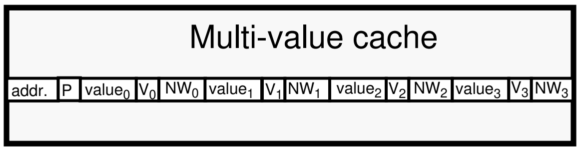

Figure 4. The multi-value cache for a SM processor with four thread units.

decreased and if it becomes zero, the result of this instruction is also written in the *i*-th entry of the live-in register file of the succeeding thread unit. At this point, the instruction of the next thread that was stalled waiting for such value, if any, is woken up. Since all the simultaneously active speculative threads follow the same path, the number of register writes is known. If different control flows were allowed as suggested in the extensions to the architecture discussed in section 3, the number of writes would not be known but it could be predicted and all the actions could be performed based on this prediction.

2.2.2 Dependences through memory

Inter-thread memory dependences are enforced by means of the *multi-value* cache. Besides, this structure provides the required support for speculating on inter-thread memory dependences by means of address prediction.

Notice that the SM processor stores multiple states of the registers, each one corresponding to a different point in time of the execution. In the same way, multiple states of the memory are supported by the multi-value cache (see Figure 4). It stores for each address as many different data words as number of thread units. For each replicated word the multi-value cache contains two additional fields: the number of writes that the corresponding thread is expected to perform (NW) and a flag indicating whether the data corresponding to that thread unit has been produced (V). Finally, each entry contains a single presence bit (P) that indicates whether those entries with a valid flag actually contain the data or it has to be searched from the next memory level. This is intended to implement a “delayed copy” policy to bring data to the multi-value cache, which reduces the pressure on the next memory level at initialization time.

This cache is initialized when a speculative thread is created. A new entry is allocated for each store whose base register is predictable. This is done by inserting in the instruction queue a special instruction that adds the register with the corresponding offset (provided by the loop iteration table). Such instruction, which is not in the static code but it is inserted by the hardware, computes the effective address and initializes the corresponding multi-value cache entry as follows.

For each predicted write address, a line is allocated in the multi-value cache if not present. If some thread has not enough entries in the multi-value cache, it and its successors are not created. When a new cache line is allocated, the NW field of the corresponding thread is set to 1, the P flag is reset and the V flags of the thread and the preceding ones are set whereas the V fields of succeeding threads are reset. If the line is already in the multi-value cache, the NW field of the thread is increased and the V bits of the succeeding threads are reset.

Store instructions update both the local memory and the multi-value cache. The local memory is just updated to store the new value, regardless of whether such address was in the local memory. If the line corresponding to the written address is in the multi-value cache, its V flag is set and its NW field is decreased. If it becomes zero, the data is copied to all succeeding threads until the next one that is expected to produce (but has not yet produced it) a different value for the same address. This implies that it is copied from the next thread to the first one that has either NW or V different from zero. The data is copied into this latter thread only if it has the V bit reset. The V bits of the threads where the produced value is copied are set. If NW becomes negative, a misspeculation checking mechanism is activated.

Misspeculations are handled by broadcasting the store effective address to the succeeding threads. Each thread checks in its load/store queue for a matching load and if it finds any, it is re-executed as well as those instructions that depend on it through the selective re-issuing approach proposed elsewhere [17][26].

When a thread performs a store and the corresponding line is not in the multi-value cache, the misspeculation checking mechanism is also activated. Besides, a new line is allocated into the multi-value cache with all the V bits set and all the NW fields equal to zero.

When a thread executes a load instruction, the local memory is checked first and in case of miss, the multi-value cache is looked up. If the corresponding data line is in the multi-value cache, it will contain a different value for each thread. If the data corresponding to that thread has its V bit set, then this value is available. The presence bit indicates whether it is present in the multi-value cache or it has to be read from the next level of the memory hierarchy. In this later case it is copied in the multi-value cache and the presence bit is set. If the V flag is not set, the load is cancelled and stored in a *load wait* queue. Loads from this queue are tried again in idle cycles of the multi-value cache. If the corresponding cache line is not in the multi-value cache, the data is read from the next memory level. Optionally, any read data can be copied into the local memory to speedup further references to the same data.

When a thread finishes (or is squashed), if the corresponding NW field of any line of the multi-value cache is greater than zero, it is reset to zero, and the value is propagated to succeeding threads as in the case when the counter becomes zero. This occurs when some predicted store did not actually occur. In this case, all dependences have been obeyed but there may be loads of succeeding threads waiting for a non existent write that must be woken up.

A line of the multi-value cache can be considered for replacement only if all its NW fields are equal to zero (this will always be the case when all the speculative threads have finished). If the line is dirty, it is considered for replacement if in addition there are not speculative threads. This ensures that the next memory level is only updated with committed values. Deadlock is guaranteed not to happen since new lines are only necessary to be allocated at speculative thread creation.

Finally, note that the local memory is not necessary for a correct functioning of the system. It is used for performance reasons, to exploit locality in the values created by the same thread. It could be also used to store live-in memory values using a value prediction scheme like the one proposed for registers. This feature will be studied in future extensions of this work.

3. Performance evaluation

In this section we present the results of a preliminary evaluation of the SM microarchitecture. The objective is to demonstrate the potential of the new architecture to exploit ILP. Evaluation of different configurations as well as the tuning of critical parameters of the architecture like the prediction scheme, size of the cache, size of register file, etc., are beyond the scope of this paper.

3.1 Experimental framework

The SM architecture has been evaluated through trace-driven simulation of the Spec95 benchmark suite. The programs have

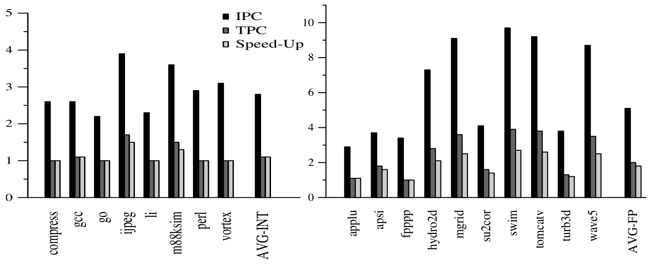

Figure 5. IPC and TPC (threads per cycle) of the SM processor and speedup when compared with a superscalar processor.

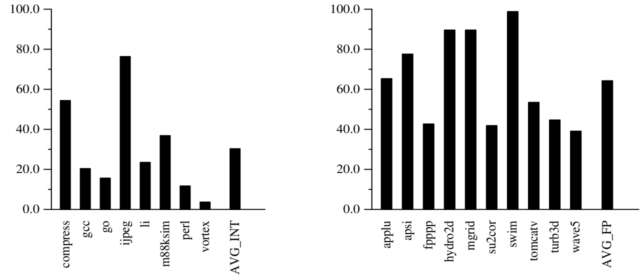

Figure 6. Percentage of dynamic instructions in innermost loops.

been compiled with the DEC Fortran and C compilers for a DEC AphaStation 600 5/266 with full optimization, and instrumented by means of the Atom tool. A cycle-by-cycle simulation is performed in order to obtain accurate timing results.

We have assumed a SM processor with 4 thread units, an issue bandwidth of 4 instructions per cycle for each thread unit, 4 entries in the loop iteration table, a multi-value cache with 128 entries (4-KB capacity) and 4 local memories with 64 entries each (512-byte capacity). The latency of these memories is 2 cycles and the next level of the memory hierarchy is assumed to have a latency of 2 cycles and an infinite capacity. The number of functional units per thread unit is (latency in brackets): 2 simple integer (1), 1 integer multiplication (2), 2 simple FP (1), 1 FP multiplication (4) and 1 FP division (17). Every thread unit has a local reorder buffer with 64 entries. The fetch bandwidth of the single fetch engine is just up to 4 consecutive instructions (i.e., no more than one taken branch). Branch prediction is performed through a branch history table with 2048 2-bit entries.

3.2 Performance figures

Figure 5 shows the average number of committed instructions per cycle (IPC) of the SM processor for the Spec95 benchmarks. It also depicts the average number of active threads per cycle (TPC) that are correctly speculated. The TPC is a measure of the thread level parallelism exploited by the SM processor. Figure 5 also includes the speedup of the SM processor over a superscalar processor with the same fetch bandwidth as the SM processor and the same resources as one of the thread units. It can be observed that the speedup of the SM processor is quite correlated with the TPC, which confirms that the TPC is a rough estimation of the additional parallelism exploited by the novel features of the SM microarchitecture. These results correspond to 100 million of instructions for each benchmark after skipping the initial part that corresponds to initialization of data structures.

The IPC of FP programs is quite high. Notice that in spite of a very simple instruction fetching scheme whose bandwidth is bounded by 4 instructions per cycle, the processor can achieve in many cases an IPC that is about twice the fetch bandwidth. For comparison, the performance of a superscalar processor is always lower than the fetch bandwidth. Increasing the fetch bandwidth is hard since it involves the prediction of multiple branches and the fetch of non consecutive code. Moreover, to achieve the same level of performance a superscalar processor should also increase the issue bandwidth. However, using the results of a study recently published by Palacharla et al. [16], in a 0.18 µm process, which is expected to be used in a few years, the worst case delay is reported to increase from 578ps for a four-issue processor to 1056ps for an eight-issue processor, that is, the cycle time would be increased by 83%. Under such circumstances, the maximum speedup that may be obtained by moving from a four-issue processor to an eight-ssue processor is bounded by 1.09. In consequence, it could not reach the performance level attained by the SM processor. In fact, the results in [16] suggest that the route for higher ILP should be based on microarchitectures that have a scalable design in most of its parts, especially in the issue logic. This is the approach taken by SM processors.

The performance of the SM processor for integer programs is significantly lower than for FP programs. It can be observed in Figure 5 that the main reason for such difference is their low TPC. Whereas the TPC for FP codes is quite high, it is very low for integer codes. There are mainly two reasons for such low TPC:

- Most of the loops in integer codes have a very low number of iterations. In average, their number of iterations per loop is 5.56 (geometric mean) [25]. Moreover, due to the use of stride predictors, thread speculation does not take place until the third iteration of the loop. Thus, if SM processors are limited to speculate just in one loop at the same time, as assumed here, the amount of thread level parallelism is very limited. Besides, when just one loop is speculated at the same time, this loop is usually (although not necessarily) the innermost loop of a nest. Figure 6 shows the percentage of dynamic instructions in innermost loops. Whereas innermost loop instructions represent 64% of the total executed instructions in the FP benchmarks, they only account for 30% for integer codes (the results in these graphs, as well as those in the remaining figures of the paper, correspond to the first 10^9 instructions of each program).
- Intra-thread control instructions are much more frequent in integer than in FP codes. Besides, such control instructions exhibit a more variable behavior in integer codes. In consequence, we have that the probability that one iteration follows the same control flow as the previous one is quite high in FP codes but it is low in some integer ones (see figure 7). This has a direct consequence in the number of misspeculated threads due to the fact that they follow a path different from that of than the non speculative one.

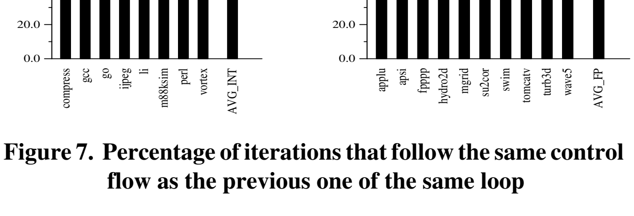

Figure 7. Percentage of iterations that follow the same control flow as the previous one of the same loop

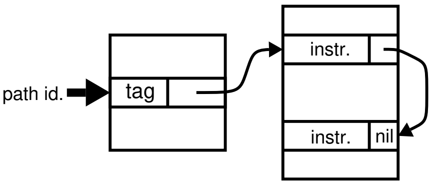

Figure 8. A loop cache. The figure shows a loop that occupies two lines.

We are currently investigating extensions to the SM microarchitecture to deal with these two issues. Regarding loops with few iterations and nests with few instructions in the innermost loop, the solution could be to allow to speculate on more than one loop simultaneously. This mainly implies that the processor should be allowed to create speculative threads from other speculative ones. Consider for instance a two deep loop nest. If the outer loop is speculated, each thread could be forked when it reaches the inner loop into different threads corresponding to different iterations of the inner loop.

The problem of a variable intra-thread control flow could be handled by not restricting all the concurrent iterations of the same loop to follow the same control flow. Although the required fetch bandwidth would increase, it could be supported by a special single-ported cache, which we call *loop cache* (see Figure 8), that has some similarities with the trace cache [18]. This cache is indexed by a path identifier and each entry corresponds to the dynamic sequence of instructions executed by a particular loop iteration, which is called a *path*. A path identifier consists of a loop identifier plus a particular control flow of the branches of one iteration. Since a path may consist of many instructions, it is split into several linked lines. Multiple fetch engines would access the loop cache alternatively, in different cycles, to get one line of a different path per cycle and broadcast it to those thread units that are executing such path. Notice that very few paths and very few fetch engines are required since a few paths account for most of the executed instructions. This can be observed in Figure 9, which shows that the 2 most frequent paths of each loop cover 91% and 96% of the total number of iterations for integer and FP codes respectively.

It should be pointed out that the problem in integer codes is not due to the data speculation technique. Figure 10 shows the average number of live-in registers and live-in memory values per iteration. In average, an iteration of the Spec95 suite has 4.3 live-in values in registers and 3.5 live-in values in memory for integer codes and 10.9 and 12.8 respectively for FP programs. Figure 11 shows the percentage of live-in registers, live-in memory values and addresses of live-in memory values that are predictable with a stride-based predictor. It can be seen that in general this percentage is high and that there is not significant difference between integer and FP codes. Obviously, the performance of SM processors could

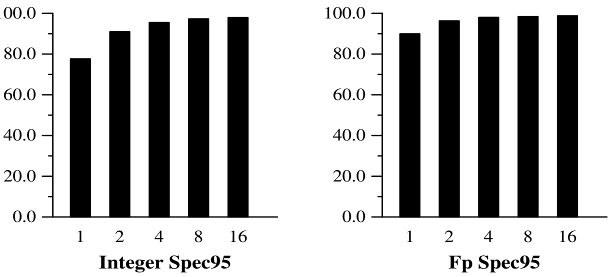

Figure 9. Percentage of iterations versus number of different paths.

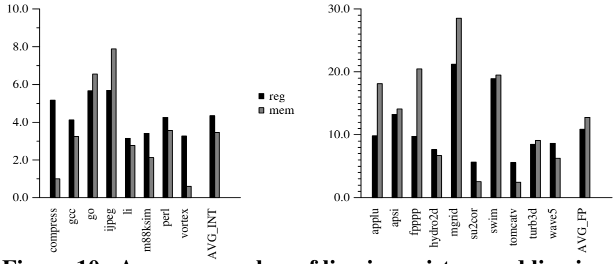

Figure 10. Average number of live-in registers and live-in memory values per loop iteration

be improved with more sophisticated predictors. We believe that this is an interesting topic for further research.

The main objective of this preliminary evaluation is to confirm the potential benefits of the SM microarchitecture. More exhaustive evaluations are required to evaluate the benefits for different configurations, like branch, value and dependence predictors, and in general to identify critical design parameters and propose alternative solutions.

4. Related work

Multithreaded architectures have been studied for long but so far the focus has been the improvement of throughput by executing several independent threads or dependent threads with the necessary synchronization added by the compiler in order to obey all dependences. On the other hand, in this paper we focus on multithreaded architectures that try to reduce the execution time by dynamically speculating (on control dependences, data dependences and data values) on multiple threads of control from a single sequential application.

Control speculation has been extensively researched. Proposed schemes for superscalar processors are based on predicting branches in the sequential order of the program. This means that a single mispredicted branch will cause the squash of every instruction fetched after it. In these schemes, branches that are difficult to predict may prevent the processor from speculating beyond them, even if successive branches are highly predictable. To obtain multiple threads of control, SM processors speculate only on highly predictable branches, and therefore they have more potential to build a large instruction window. On the other hand, speculating on non contiguous branches results in a non contiguous instruction window that is more complex to manage.

Data dependence speculation is used by some processors in the memory disambiguation stage. Both the address resolution buffer [7] and the time-sequence cache [2] of the Multiscalar, the Speculative Versioning Cache that is based on a snoop cache coherence protocol [11], as well as the address reorder buffer of the HP PA8000 [12] are examples of such mechanism. However, these techniques assume that any load is independent of all the previous stores whose addresses are unknown. More sophisticated

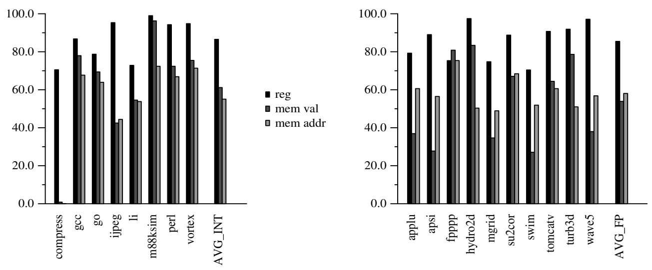

Figure 11. Percentage of predictable live-in registers, live-in memory values and addresses of live-in memory values.

approaches have recently been proposed in order to predict more accurately the existence of a data dependence through memory and avoid some of the expensive recovery actions that are required by misspeculations [8][9][15]. Data dependence speculation has also been proposed to improve the parallelization of code for single-chip multiprocessors [22].

There are few proposals in the literature dealing with the dynamic management of a large window that consist of several threads of control not necessarily independent among them obtained from a sequential program. Pioneer work on this area was the Expandable Split Window paradigm [6] and the follow-up work on Multiscalar processors [21]. Other proposals are the SPSM architecture [5]; the Superthreaded architecture [23]; the Multithreaded Decoupled architecture [4]; the Dependence Speculative Multithreaded Architecture (DeSM) [14], and Trace processors [17]. There are important differences between the SM microarchitecture and those previous proposals:

- The Multiscalar, SPSM, Superthreaded and Multithreaded Decoupled architectures require some addition/extension to the ISA. On the other hand, the SM architecture uses speculation techniques to obtain and manage multiple threads of control dynamically from a sequential conventional object code without any support from the user/compiler. Moreover, in those architectures data dependences are always enforced by executing the producer instruction before the consumer one. On the other hand, the SM architecture uses data value speculation to deal with inter-thread dependences, both through registers and memory. In this way, the producer and consumer instructions can be executed in any order as though they were independent, provided that the predicted values are correct.

- Data value speculation has been used in previous proposals, mainly in the context of a superscalar processor with a single thread of control [9][10][13][19][20]. Data speculation is also used by Trace processors. However, there are significant differences between Trace processors and SM processors. First, SM processors speculate on inter-thread data dependences by predicting all memory references at the beginning of a thread, whereas Trace processors execute speculatively memory instructions based on the predicted value of source operands but these instructions compete with the other instructions for securing issue slots. Thus, when a SM processor disambiguates a load instruction all the addresses of previous stores have been predicted (assuming that they are predictable) whereas this is not the case of Trace processors. In consequence, Trace processors will experience a much higher number of memory dependence misspeculations, that is, the data dependence speculation mechanism of SM processors is more accurate than that of Trace processors. The second important difference is that Trace processors require a global register file that may become a bottleneck for the scalability of the system, whereas SM processors have all the register files completely distributed. Third, unlike SM processors, the multiple instructions windows simultaneously managed by Trace processors are adjacent. Finally, the approach to build the instruction window is different: whereas in Trace processors it is based on a trace cache [18], in SM processors it is based on a loop prediction technique [25].

- Data dependence speculation is used by the Dependence Speculative Multithreaded architecture, but it does not perform data value speculation. Data dependence speculation is also used by the Multiscalar and SPSM architectures. However, they just implement an "always-independent" prediction scheme, whereas the approach used by the SM architecture is more powerful since it is based on memory address prediction. Memory addresses have been shown to be highly predictable. Previous works have used this fact to implement hardware prefetching ([3] among many others), to reduce the memory latency perceived by the processor ([1] among others), or to implement data value speculation [9][19] in the context of a superscalar processor.

Improving the instruction fetch bandwidth has been the target of some recent works. However, all of them are oriented towards a processor that supports a single thread of control. Some proposals have focused on improving the branch prediction throughput ([30] among others), whereas others have in addition addressed the problem of non contiguous instruction fetching [18]. The SM architecture takes a different route: it reduces the fetch bandwidth requirements by taking advantage of the fact that simultaneously active threads process the same code with different data. A similar feature is exploited by the CONDEL architecture [27] and the dynamic vectorization approach proposed in [28]. However, those approaches are more restrictive than the one used by SM processors since the former is limited to loops whose static body does not exceed the implemented instruction window, whereas the latter is feasible only if the dynamic sequence of instructions executed by the loop are the same for all the iterations and they fit into a single instruction cache line (the instruction cache organization that they use is the trace cache [18]).

5. Conclusions

We have presented a novel processor microarchitecture, which is called Speculative Multithreaded (SM). A novel feature of such architecture is its ability to dynamically extract and execute multiple threads of control from a single sequential program written in a conventional ISA without requiring any compiler support. Multiple concurrent threads execute different iterations of the same loop. These threads are not necessarily independent (usually they are dependent) but inter-thread data dependences are resolved by speculation techniques: both dependences and values that flow through them are predicted. In this way, loops that are not parallelizable by the compiler can be executed in parallel if data dependences and data values are correctly predicted. The second main feature of the architecture is that the additional instruction level parallelism due to inter-thread parallelism hardly increases the fetch bandwidth requirements since multiple threads share the same code. Once a new instruction is fetched, it is copied into the instruction register of every thread. Then, its operands are renamed using a different register map table for each thread and afterwards, the renamed instructions are dispatched to their respective instruction queues.

A preliminary evaluation of a SM processor has shown that it can achieve a high IPC for FP programs, which can be even much higher than the fetch bandwidth. Besides, since the architecture is based on a scalable design, it does not suffer from the cycle time penalties that wide issue superscalar processors are expected to experience. For integer codes the thread level parallelism is much lower. Several extensions to the architecture have been proposed to alleviate such problem: speculating on multiple loops simultaneously and speculating on iterations of the same loop with different control flow. A loop cache has been proposed as an extension to support the increased fetch bandwidth required by these extensions.

In summary, we have shown that the combination of data value speculation, data dependence speculation, multiple speculative threads of control and a distributed design is a promising alternative to relieve the most critical bottlenecks of current superscalar microprocessors: data dependences, the instruction window size, the complexity of a wide issue machine and the limited instruction fetch bandwidth.

6. Acknowledgements

This work has been supported by the Spanish Ministry of Education under contract CYCIT 429/95 and by the grant AP96-52274600. The research described in this paper has been developed using the resources of the Center of Parallelism of Barcelona (CEPBA).

7. References

[1] T.M. Austin, G.S. Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency", *Proc. of Int. Symp. on Microarchitecture,* pp 82-92, 1995.

[2] S.E. Breach, T.N. Vijaykumar, S. Gopal, J.E. Smith and G.S. Sohi, "Data Memory Alternatives for Multiscalar Processors", technical report # CS-TR-97-1344, University of Wisconsin, 1997

[3] T-F. Chen and J-L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes", *Proc of the Int. Symp. on Computer Architecture,* pp. 223-232, 1994.

[4] M.N. Dorojevets and V.G. Oklobdzija, "Multithreaded Decoupled Architecture", *Int. J. of High Speed Computing*, 7(3), pp. 465-480, 1995.

[5] P.K. Dubey, K. O'Brien, K.M. O'Brien and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading", *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 109-121, 1995.

[6] M. Franklin and G.S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism", *Proc. of Int. Symp. on Computer Architecture*, pp. 58-67, 1992.

[7] M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", *IEEE Transactions on Computers*, 45(6), pp. 552-571, May 1996.

[8] J. González and A. González, "Memory Address Prediction for Data Speculation", *Proc. of EURO-PAR 97 Workshop on ILP*, pp.1084-1091,1997.

[9] J. González and A. González, "Speculative Execution via Address Prediction and Data Prefetching", *Proc of 11th. ACM Int. Conf. on Supercomputing*, pp. 196-203,1997.

[10] J. González and A. González, "The potential of Data Value Speculation to Boost ILP", *Proc. of Int. Conf. on Supercomputing*, 1998.

[11] S.Gopal, T.N. Vijaykumar, J.E. Smith and G.S. Sohi, "Speculative Versioning Cache",Proc. of the 4th Int. Conf. on High-Performance Computing Architecture, Feb. 1998.

[12] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000", *Proc. of the CompCon'95,* pp. 123-128, 1995.

[13] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, "Value Locality and Load Value Prediction", *Proc. of the 7th. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, Oct. 1996.

[14] P. Marcuello and A. González, "Control and Data Dependence Speculation in Multithreaded Processors", *Proc. of the Workshop on Multithreaded Execution, Architecture and Compilation* held in conjuction with HPCA-4, 1998

[15] A. Moshovos, S.E. Breach, T.N. Vijaykumar and G.S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences", *Proc. of Int. Symp. on Computer Architecture*, pp. 181-193, 1997.

[16] A.S. Palacharla, N.P. Jouppi and J.E. Smith, "Complexity-Effective Superscalar Processors", *Proc. of Int. Symp. on Computer Architecture*, pp. 206-218, 1997.

[17] E. Rotenberg, S. Bennett and J.E. Smith, "Trace Processors", *Proc. of the 30th. Int. Symp. on Microarchitecture,* Dec. 1997.

[18] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", *Proc. of the 29th. Int. Symp. on Microarchitecture,* Dec. 1996.

[19] Y. Sazeides, S. Vassiliadis and J.E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing", *Proc. of the 29th. Int. Symp. on Microarchitecture,* pp. 238-247, Dec. 1996.

[20] Y. Sazeides and J.E. Smith, "The Predictability of Data Values", *Proc. of the 30th. Int. Symp. on Microarchitecture*, pp. 238-247, Dec. 1997.

[21] G.S. Sohi, S.E. Breach and T.N. Vijaykumar, "Multiscalar Processors", *Proc. of the Int. Symp. on Computer Architecture,* pp. 414-425, 1995.

[22] J.G. Steffan and T.C. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization", *Proc. of the 4th Int. Conf. on High-Performance Computing Architecture*, Feb. 1998.

[23] J-Y. Tsai and P-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation", *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35-46, 1996.

[24] D.M. Tullsen, S.J. Eggers and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proc. of the Int. Symp. on Computer Architecture,* pp. 392-403, 1995.

[25] J. Tubella and A. González, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection", *Proc. of the 4th Int. Conf. on High-Performance Computing Architecture*, Feb. 1998

[26] G. S. Tyson and T.M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming", *Proc of the 30th. Int. Symp. on Microarchitecture,* 1997

[27] A. K. Uht, "Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream", *IEEE Trans. on Computers*, vol 41, July 1992.

[28] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences", Proc. *of the Int. Symp. on Computer Architecture,* pp. 1-12, 1997.

[29] K.Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", *Proc. of the 30th. Int. Symp. on Microarchitecture*, Dec. 1997.

[30] T-Y. Yeh, D.T. Marr and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and an Address Cache", *Proc. Int. Conf. on Supercomputing,* pp. 67-76, 1993.