# 2

# SPECULATIVE MULTITHREADED ARCHITECTURES

*In this Chapter, the execution model of the speculative multithreading paradigm is presented. This execution model is based on the identification of pairs of instructions in the program to spawn speculative threads. The first instruction of the pair, that is referred as spawning point is the instruction that when reached, fires the creation of a speculative thread. The second instruction of the pair, that is referred to as control quasi-independent point, is the instruction where the speculative thread starts its execution. The hardware requirements for supporting this execution model are also identified and the tasks involved in the spawn and the commit of the threads are discussed.*

## 2.1. INTRODUCTION

In the previous Chapter, the two main requirements for exploiting speculative thread level parallelism were pointed out: i) hardware support for executing multiple threads simultaneously and a partitioning mechanism to split the program into speculative threads. In this Chapter and the following one, the hardware requirements for supporting the speculative multithreaded execution model are investigated whereas Chapter 4 is focused on the spawning schemes.

The hardware support necessary for executing speculative thread-level parallelism is different depending on the type of parallelism that is to be exploited. Main differences come from the way values are initialized for the speculative threads, the way interthread data dependences are managed and the way speculative threads finish their execution.

Regarding these issues, eager execution and helper thread paradigms are very similar. In both cases, speculative threads are initialized with the values of the parent thread at the spawn time. Also, no hardware for forwarding dependent values is required since the speculative threads run independently. The main difference between these two paradigms resides on the lifetime of the speculative threads whereas in eager execution, speculative threads may become non-speculative or be squashed, in the helper thread paradigm speculative threads never become non-speculative, there is always a main non-speculative thread that executes all the instructions of the program.

Nevertheless, speculative threads in the speculative multithreaded paradigm behave significantly different to the previous schemes. In this paradigm, speculative threads are not initialized just with the values of the parent thread at the spawn time since such threads will execute instructions far away of the spawning point. Besides, interthread data dependences will be present and mechanisms to enforce such dependences are necessary. Finally, speculative threads work cooperatively and each one executes different parts of the dynamic stream. Thus, a speculative thread becomes the non-speculative one when all the previous threads have finished their execution.

Basically, the main requirements to implement a speculative multithreaded processor are:

- Separate contexts for the speculative threads: Speculative threads need to store local information such as their own instruction pointer and their own register and memory values. Besides, some of these values are shared among several speculative threads whereas some others are private. Thus, it is necessary to maintain multiple versions for some registers and memory locations.

- Storage for the speculative state: Speculative threads execute instructions that may belong to the wrong path or may use incorrect operands due to its speculative nature. Therefore, speculative threads are not allowed to modify the architected state of the processor until they become non-speculative and mechanisms to store and access the speculative state are needed.

- Interthread data dependence management: Speculative threads are control and data dependent on previous threads. In order to correctly execute them, such dependences have to be obeyed. So, mechanisms to detect such interthread dependences and forward the dependent values from the producer speculative thread to the consumer one are also required.

- Speculative thread order knowledge: An order relationship among threads have to be maintained to avoid that a logically younger thread uses a value produced by a logically older thread in program order. Therefore, it is necessary to provide the processor of mechanisms to keep track of the logical order among concurrent threads.

- Verifying the speculation: when the non-speculative thread finishes its work, it has to free its context and convert the next thread in logical order into the non-speculative one. Before that, the non-speculative thread has to verify that the speculation has been correctly performed, that is, that the input values of the following thread were correct.

In this Chapter, different proposals to deal with these requirements are presented and analyzed for different microarchitectural platforms, a clustered processor made up of several thread units and a centralized organization. The rest of the Chapter is organized as follows: Section 2 presents the execution model of the speculative multithreaded processors. Section 3 presents different microarchitectural platforms to execute speculative threads. Section 4 analyzes all the processes involved in the creation of speculative threads and Section 5 analyzes the processes involved in committing speculative threads and all the validation mechanisms to detect misspeculations. Section 6 presents some related architectures proposed in the literature and finally, Section 7 summarizes the main conclusions of this Chapter.

All topics related to the management of interthread data dependences will be thoroughly analyzed in Chapter 3.

## 2.2. EXECUTING SPECULATIVE THREADS

As it was mentioned in the previous Chapter, the execution model of the speculative multithreading paradigm is based on partitioning the code into threads that are executed concurrently by the multithreaded processor. The partitioning process will be thoroughly analyzed in chapter 4, but basically, it identifies

pairs of instructions in the dynamic instruction stream that are referred to as *spawning pairs*. The spawning pairs are made up of a *Spawning Point* and a *Control Quasi-Independent Point*. The spawning point is an instruction that when it is reached, it can fire the creation of a new thread. The control quasi-independent point is the instruction where the spawned thread starts its execution. The spawning and the control quasi-independent points can be any instruction in the program even though extensions to the instruction set architecture may be required to indicate these points.

Thus, programs in speculative multithreaded processors perform in the same way that in a traditional superscalar processor until a spawning point is reached. Then, a new thread is spawned in a free context of the processor. Once the spawning process finishes, both threads proceed in parallel. The spawner thread executes all the instructions between the spawning and the control quasi-independent point and the spawned thread executes instructions beyond the control quasi-independent point. In this example, the spawner thread is also known as the non-speculative thread and the spawned thread as the speculative thread.

Moreover, some parts of the context have to be initialized to correctly execute the speculative threads, such as the instruction pointer and the values that are to be consumed by the speculative thread. In the next Chapter, mechanisms to deal with those values consumed by the speculative thread that are not available at the spawn time will be analyzed.

When the non-speculative thread reaches the control quasi-independent point, the speculation has to be verified by checking whether the speculative thread was initialized with the proper values. If the speculation has been correctly performed, then the speculative thread becomes the non-speculative thread and the context of the committed thread becomes free for a future use of new speculative threads.

This model can be generalized so that any thread may spawn new speculative threads. In this more aggressive execution model, only one thread is non-speculative whereas the rest of them are speculative. The non-speculative thread is the thread that has not been spawned by any of the other active threads and it is the only one allowed to commit its instructions. Speculative threads will have to wait to become the non-speculative to commit their instructions and the values produced by them.

Active threads are ordered according to their program order. This order is the order in which the instructions of speculative threads would be executed if the program run single-threaded. Therefore, if the program order of threads A,B,C and D is A-B-C-D, thread A is the non-speculative thread, we say that thread B is older than thread C, and thread B is less speculative than thread C. Similarly, in this thesis, we

also consider that thread C is younger than thread B and thread C is more speculative than thread B. So, the terms older and younger are not referred to the time threads are created but the program order position regarding the sequential program execution. With this terminology, the non-speculative thread is always the oldest thread

There are different reasons to explain the performance improvement achieved by speculative multithreading. The main reason is that this execution model allows the processor to build a huge instruction window as it is illustrated in figure 2.1 In ILP processors, the effective size of the instruction window is limited by the branch prediction accuracy since the amount of correctly speculated control-flow instructions depends on the number of consecutive branches that have been correctly predicted. This is due to the sequential nature of the fetching mechanism of superscalar processors since a single mispredicted branch prevents the instruction window from growing beyond the branch. For non-numeric programs, which have many difficult-to-predict branches, this may be just by itself a very important limitation.

On the other hand, in this execution model instructions are not fetched in program order. Thus, the instruction window is not built by a sequential process but it consists of a collection of non-adjacent smaller windows ($W_1$, $W_2$, $W_3$ and $W_4$). Each small window is composed of a subsequence of the dynamic instruction stream, and it is build by sequentially speculating on branches.

As it is expected, the main problem of such execution model is the dependences among the different instruction windows, both control and data dependences. Such dependences do not only exist between the instructions already fetched, but among those instructions that have not been fetched yet.

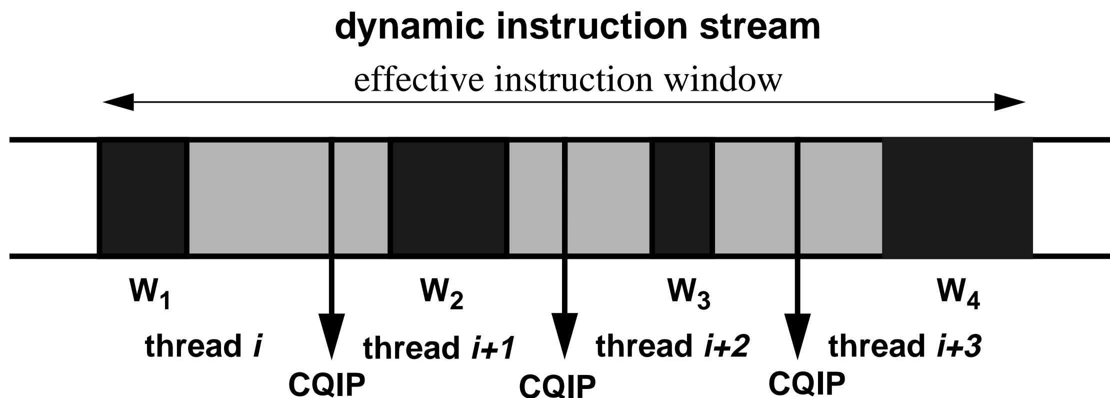


**Figure 2.1.** Effective instruction window managed by speculative multithreaded processors.

## 2.3. ARCHITECTURAL SUPPORT

Exploiting thread-level parallelism is not a novel idea and there are lots of microarchitectural proposals in the literature, especially in a non-speculative manner. Well-known examples of such architectures are for instance, the Simultaneous Multithreaded Architecture proposed by Tullsen *et al.*[79] and multiprocessors.

In fact, any architecture that is capable of executing multiple threads is suitable for exploiting speculative multithreading. To do that, it is necessary to add to such architectures some elements for supporting speculative threads. These extra requirements are necessary due to the speculative nature of the threads since the execution of such threads is control and data dependent on the previous threads. Basically, the processor has to provide support for:

- Storing the speculative state of the speculative threads: Speculative multithreaded processors have to provide storage for holding the speculative versions of the register and memory values until such threads become the non-speculative ones.
- Knowing the usage of the contexts of the processor: The processor has to know the availability of the contexts for allocating new speculative threads when a spawning point is reached.
- Maintaining the program consistency: Speculative threads can be created out of the program order. Therefore, the processor has to provide mechanisms for knowing which is the program order among the speculative threads in order to avoid data dependence violations. Thus, values produced by a thread should be visible to younger threads but not to older threads.
- Managing interthread data dependences: Speculative threads execute future instructions in the dynamic instruction stream and data dependences among different threads may occur. Mechanisms for detecting such dependences as well as for detecting data dependence violations are needed.

In this section, different implementations for these requirements are investigated excepting the management of interthread data dependences, which will be analyzed in the next Chapter. The study of the architectural requirements is done for two architectural platforms. The first one is a totally centralized multithreaded processor. The second one is based on an on-chip multiprocessor with two different interconnection networks, an unidirectional ring and a fully-interconnected scheme based on a crossbar.

### 2.3.1. Centralized Speculative Multithreaded Processor

Figure 2.2 shows the design of a centralized speculative multithreaded processor. This architecture is very similar to a simultaneous multithreaded processor[79]. Almost all the subsystems of the processor are

shared among the active threads. Sharing may benefit the performance of the processor since may increase the usage of such resources even though it may also cause some contention or degradation on the performance of those resources.

On the other hand, there are some elements of the processors that are replicated for each context such as the individual program counter of each of the threads, the reorder buffer and the register map table. Thus, different contexts have different versions for the architected registers for their own use and for holding the speculative state of the thread.

Regarding the register file, different implementations can be considered. One solution is based on replicating the register file for the different contexts. Different threads use different register locations. This partition can be done physically or logically. In the latter case there is a huge register file, but a context can only access to a given range of physical registers and the ports of the register file are shared among the contexts.

However, the major drawback of having a partitioned register file is the initialization cost. When a speculative thread is spawned and allocated to a context, it is necessary to copy the register values and the register map table from the parent thread to the spawned one and this copying process may cause a lot of traffic.

Moreover, this naïve scheme may waste resources and time. For instance, some register values that are not to be consumed by the spawned thread are copied anyway. In the same way, some values are shared by all the speculative threads and having different physical registers with the same value results in a waste of space and the time of copying.

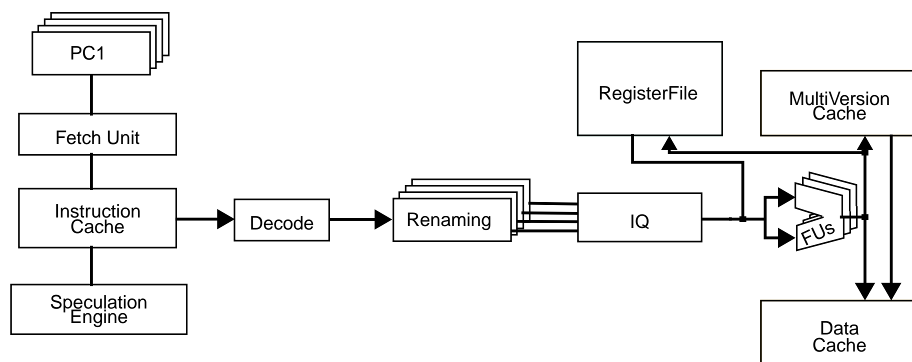


**Figure 2.2.** Centralized Speculative Multithreaded Processor.

Considering a smarter mechanism for initializing the register values of the speculative threads may reduce the time of copying. In fact, it is only necessary to maintain different versions for those registers that are to be written by the spawned thread. And, for this group, it would only be necessary to copy the value for those logical registers that are to be read by the speculative thread. However, instead of copying the value, both threads may share the physical location until the speculative thread has to write in it. Then, the speculative thread will create a new version for that logical register. Handling interthread register dependences will be investigated in Chapter 3.

To implement this new register allocation, a new organization of the register file is required as well as a mechanism (hardware or software) to know which logical registers are to be used by the spawned threads. To implement this approach, a large register file fully shared by all the contexts is necessary. Thus, when a thread is spawned, instead of copying all the register values from the parent thread to the spawned one, only the register map table is copied. In this way, the physical locations of all the architected registers are shared. Different versions of the logical registers are supported by means of maintaining different register map tables. When a speculative thread writes in a register, the renaming mechanism supplies a new physical register which is stored in the local register map table to prevent the parent thread from accessing the value produced by a latter thread.

Since a physical location may be accessed by different threads, it is not possible that a given thread frees a register location if it is still visible for any other active thread. So, each entry of the register file requires a counter that indicates the number of threads that can access to such physical register. Thus, when a thread frees a physical register, the counter associated to such entry is decreased and it is only freed when the counter becomes 0.

On the other hand, when a speculative thread is spawned and the register map table is copied from the parent thread to the spawned thread, all the counters of the physical registers of the parent thread are increased. In fact, if the processor knows which of such logical registers are to be consumed by the spawned thread, it can only increase the associated counters of such physical registers. Besides, the entries of the register map table that corresponds to those logical registers that are not to be consumed, have to be initialized with a special value such as NIL.

Regarding memory values, the different versions of the memory locations are stored in a special first level cache which is referred to as MultiVersion Cache. This cache also detects interthread memory dependence violations and will be thoroughly studied in the next Chapter.

Finally, in figure 2.2 appears an additional block that is the Speculation Engine Block. Among other things, this block keeps track of the availability of the contexts as well as the logical order among the current active threads.

In summary, the main advantages of the centralized design lie on the low communication latency among the current active threads and a higher resource utilization. However, the requirements to increase the size of several parts of the processor such as the register file, may increase its access time which may affect the cycle time of the processor.

### 2.3.2. Clustered Speculative Multithreaded Processor

As it was mentioned before, centralized processors require to increase the size of several systems of the processor and it may affect its access time. A very common technique to solve this problem is clustering ([15][24] among others). Clustering splits a structure into multiple faster and smaller blocks. A possible approach for clustering is to physically separate the contexts, that is, having different small processors for each of the contexts of the multithreaded processor similar to an on-chip multiprocessor.

Thread units are similar to a superscalar core and they have their own register file, their own instruction window and their own functional units. The resource contention among different threads is eliminated in this way.

In addition to the local register file, the local instruction window and the local functional units, thread units also have a local cache to store those values that are produced and consumed by this thread. However, the values produced by a speculative thread cannot update the main memory until this thread becomes the non-speculative one.

The different versions for the registers and memory values are supported by means of the local register file and the local memory. Similarly to the partitioned scheme in the centralized version, when a thread is spawned and a thread unit is assigned, the corresponding local register file has to be initialized with the values of the parent thread. As sharing values may be very difficult, all the register values have to be copied from the parent thread to the spawned one through the interconnection network. In this thesis we have analyzed two topologies: an unidirectional ring topology and a fully-interconnected one. Next subsections will present both schemes.

Independently of the selected topology, it is also necessary to have a centralized speculation engine for keeping track of the availability of the thread units as well as the thread ordering of the current active threads.

The main advantage of a clustered design is the scalability since introducing more thread units does not significantly affect the latency of the different blocks of the processor. However, the communication latency to forward values from one thread to another is larger than for the centralized processor since values have to travel through an interconnection network.

### 2.3.2.1. Unidirectional Ring Clustered Speculative Multithreaded Processor

Figure 2.3 illustrates a 3-thread unit clustered speculative multithreaded processor interconnected by means an unidirectional ring topology. In this implementation of the clustered processor, a thread unit can only communicate with the adjacent thread units. Moreover, this communication is unidirectional, that is, a thread unit only receives values from the previous thread unit and only can send values to the following one. Therefore, if a speculative thread needs a value from a non-adjacent thread, such value has to travel through the intermediate thread units until reaching the consumer one.

This kind of interconnection favours the execution model in which speculative threads are created in program order. In this execution model, the logical order of the speculative threads matches with the physical order of the thread units in the ring and thus, the communication latency to initialize and to forward values from the producer to the consumer thread is faster since values only have to be communicated to the following thread.

When a speculative thread is created, the register file and the register map table are initialized with the contents of the register file and the register map table of the parent thread. In the same way as for the cen-

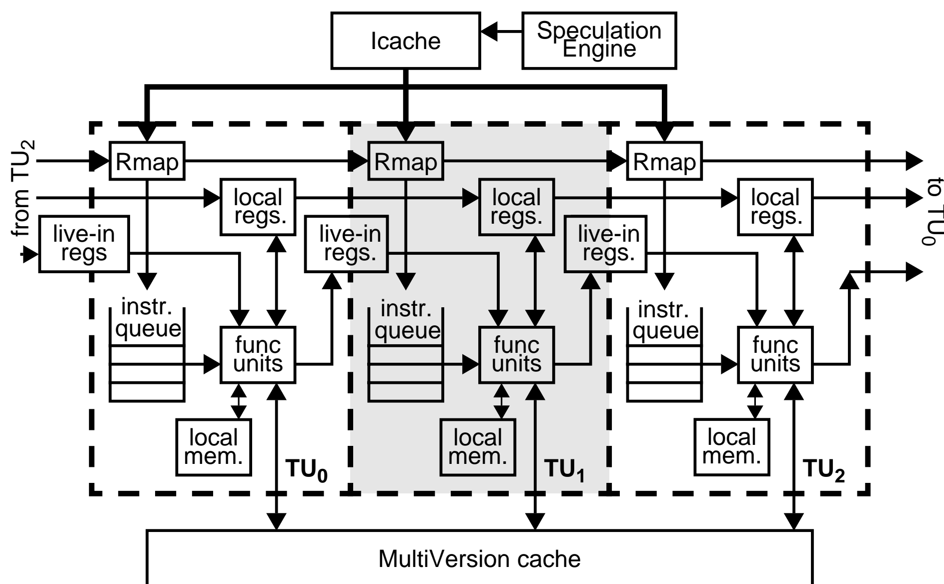


**Figure 2.3.** Clustered Speculative Multithreaded processor with three thread units.

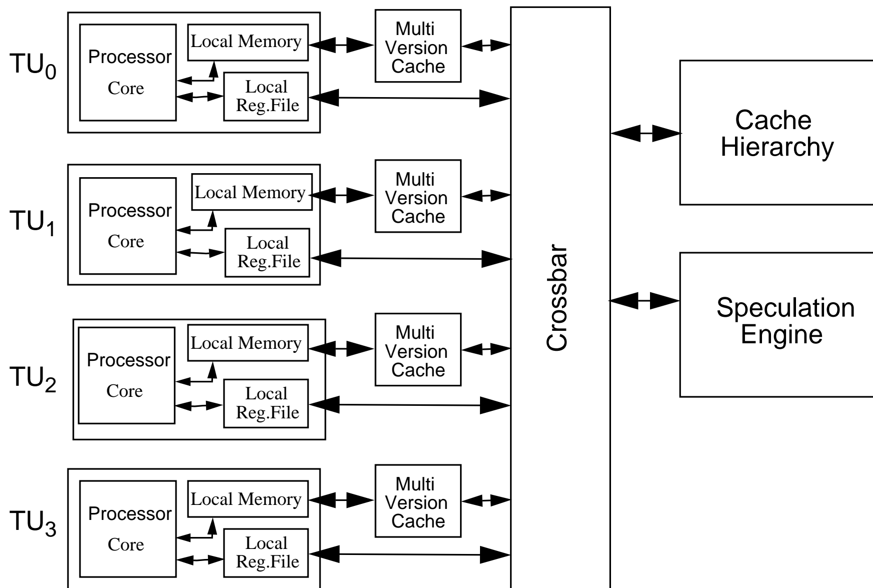


**Figure 2.4.** A clustered speculative multithreaded processor with four thread units fully interconnected.

tralized version, it is not necessary to copy everything in the register file but only those register values that are to be consumed by the spawned thread.

To forward interthread dependent register values, a special register file which is referred to as live-in register file is used. Its management will be detailed in the next Chapter.

Finally, in the same way as for the centralized speculative multithreaded processor, it is necessary to keep track of the different versions of the memory values and to detect any possible misspeculation. Here, the same special first level cache as the one of the centralized processor, a MultiVersion cache, may work. Note that this cache can be implemented either centralized or clustered.

#### 2.3.2.2. Fully-Interconnected Clustered Speculative Multithreaded Processor

The main problem for the unidirectional ring architecture is that the communication latencies to forward dependent values are variable and depend on the physical location of the threads, that is, how far is the thread unit where the consumer speculative thread is allocated. To avoid that, a different interconnection network can be considered. For instance, a crossbar that interconnects all the thread units of the processor.

Figure 2.4 shows a fully-interconnected clustered speculative multithreaded processor. It is made up of several thread units which are similar to a superscalar core. Different versions of the register and memory

values are obtained by means of local register files and local memories. A MultiVersion cache is also needed.

## 2.4. SPAWNING SPECULATIVE THREADS

When a thread reaches a spawning point, a new speculative thread can be created. This spawning process includes several non-negligible tasks such as checking for an idle context, evaluating which is the logical order position regarding the other active threads and initializing the values for the new speculative thread. In this section, different approaches to implement some of these tasks are discussed. The way interthread dependent values are initialized and how such dependences are enforced will be analyzed in the next Chapter.

### 2.4.1. Spawning models

In previous sections, it has been mentioned that any active thread is allowed to spawn new speculative threads. This spawning model is referred to as *unrestricted thread ordering*.

The greatest benefit provided by this spawning model is that it allows the processor to speculate at different loop levels. However, it is difficult to exactly determine which is the logical order of a spawned thread. In this thesis, we propose a thread order predictor.

When a thread is spawned by any other thread, it is obvious that the new spawned thread is younger than the spawner thread. So, it is only necessary to determine the order with respect to the threads that are younger than the spawner one. Therefore, if the youngest thread -that is, the most speculative thread- is the only one allowed to spawn speculative threads, the thread order predictor is not required since the order position of the spawned thread is obvious. This spawning model is referred to as *sequential thread ordering*.

As it is expected, the potential of this spawning model is lower than the one offered by the unrestricted thread ordering model. However, the main advantage of this model is that fits very well into an unidirectional ring clustered processor since the order of the threads can match with the physical order of the thread units.

### 2.4.2. Thread Unit Availability

The first step of the spawning process is to look for an idle context where the new spawned thread will be allocated. If there are no thread units available, different solutions can be taken:

- The new speculative thread is not spawned.
- The spawner thread is stalled until there is a thread unit available. This solution is not very good since it may cause deadlocks if the non-speculative thread is stalled. Moreover, it does not seems worthy to stall the execution of a thread just for creating a speculative thread that may be never reached.
- Compute the order of the new speculative thread and, if there is any active thread more speculative than it, squash the most speculative and place the new spawned one at that thread unit.

In this thesis the last approach has been considered. However, if the speculation mechanism is very reliable, the solution that provides better performance may be the first one since it does not eliminate speculative threads that are correctly executing.

### 2.4.3. Thread-Order Prediction

When a new speculative thread is spawned, it is necessary to know which is the order of this thread with respect to all the other active threads. This is necessary not only to know which is the next non-speculative thread when the current one commits, but also to early detect data misspeculations. A data misspeculation occurs when an older thread has consumed a value that has been produced by a younger thread. The way to detect such data misspeculations will be detailed in the next Chapter.

For the sequential thread ordering, computing the order of a new spawned thread is straightforward since the spawned threads always are the new most speculative ones.

This task becomes difficult for the unrestricted thread ordering model. Here, any active thread is allowed to spawn a new speculative thread and such thread may be located at any order position between the spawner and the most speculative thread.

Naive solutions that consist in assuming that the new speculative thread is always the most speculative or just the next to the spawner thread can be considered. However, the processor can take benefit from the fact that speculative threads are usually executing with the same other speculative threads and that the existent order among them is usually the same. This suggests a simple thread order predictor which predicts for a new spawned thread the same order that in its previous execution with the same threads.

Regarding the way of representing the order information, the order relationship between two threads only has two possible outcomes, younger or older. So, to represent the order relationship between two threads, it is only necessary to keep one-bit information, that is, if the new spawned thread will be committed before (1) or after (0) the other active thread.

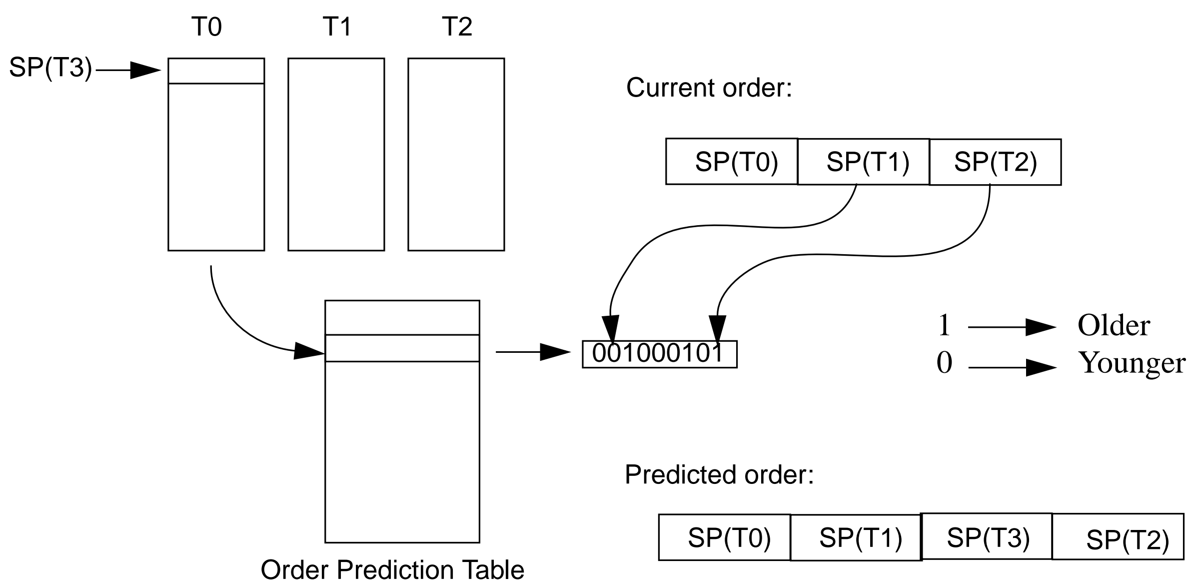


**Figure 2.5.** The thread order predictor.

Then, every spawning pair has a table that contains all the order relationships with respect to all the other speculative threads. This table can be tagged, that is, each entry contains the identifier of the other speculative thread, or non-tagged. Each entry of the table can contain just one bit indicating which was the last order relationship among them or a $n$-bit saturated counter.

The behavior of the thread order predictor is shown in figure 2.5. The speculative multithreaded processor is executing three threads and the non-speculative thread reaches a spawning point. If there is an available thread unit, the speculative thread is created and the logical order with the other two active threads has to be determined. There are three possibilities, the new speculative thread is older than both, it goes between them or it is the most speculative.

With the thread identifier of the new spawned thread, which may be any hash function combining the instruction pointers of the spawning and the control quasi-independent point, the order predictor table is accessed. Each entry of this table contains the order relationships for this speculative thread. Then, this entry is accessed with the thread identifiers of the other active threads to determine the logical order of the new speculative thread. In this example, the spawned thread is predicted to be younger (more speculative) than thread T1 and older (less speculative) than T2.

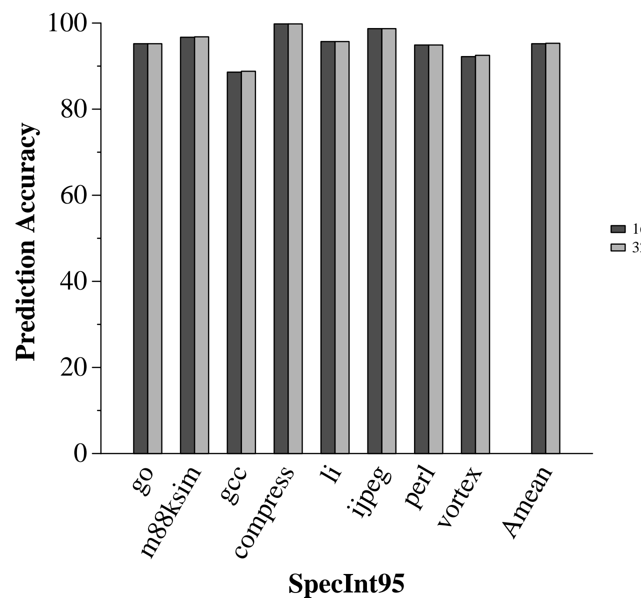


**Figure 2.6.** Prediction accuracy of the thread order predictor.

Inconsistent predictions, that is, the outcome predicts that the new thread is older than thread A but younger than thread B and thread A is older than thread B can be eliminated just computing sequentially the order from the less speculative to the most one until the prediction is “older”.

The update is done when a misspeculation occurs, that is, the non-speculative thread has reached a control quasi-independent point of an active thread different to the expected one and when a thread is committed. The best approach to updating is to store the current order and the prediction at the moment of the thread creation and make the update for the threads that exist at the spawning time. The update can also be done at the commit-time. This solution may produce misspeculations since it updates entries of the table that were not accessed at the spawn-time.

#### 2.4.3.1. Performance evaluation

Figure 2.6 shows the prediction accuracy obtained by the order predictor for the SpecInt95. The assumed spawning scheme will be presented in Chapter 4. The size of the order prediction table is 4K entries and each entry contains 16 or 32 2-bit saturating counters, and the access to the second level table is non-tagged. It can be observed that the order information is quite redundant and simple predictors achieved very high hit rations, greater than 95% in both cases.

### 2.4.4. Other Considerations: Branch Prediction

A centralized branch predictor may not be adequate for speculative multithreaded processors since it would require a large number of ports. In addition, branches are not fetched in sequential order, and thus the history information, especially global history registers, would be significantly degraded by this.

Each thread unit could have its local branch predictor that could be copied from the table of the parent thread at initialization. This may imply additional traffic from the spawner to the spawned thread. Alternatively, branch prediction tables could be not initialized at thread creation. Instead, when a new thread is started in a thread unit, it simply inherits the prediction table as it was left by the previous thread executed in that unit.

Predicting the outcome of branches only based on the history of branches executed in the same thread unit may cause negative effects in the accuracy of the predictor and therefore, in the overall performance of the processor.

Figure 2.7 compares the branch prediction accuracy of branch predictors initialized from the parent at thread creation and that of a non-initialization policy considering the fully-interconnected clustered speculative multithreaded processor with 16 thread units and perfect register value prediction. In addition, it also shows the prediction accuracy of a centralized predictor that processes all branches in sequential order as a superscalar microprocessor does, as a baseline for comparison. Observe that the degradation suffered when the copy mechanism is implemented is very low (only 1% for a loop-iteration spawning policy and 4% for

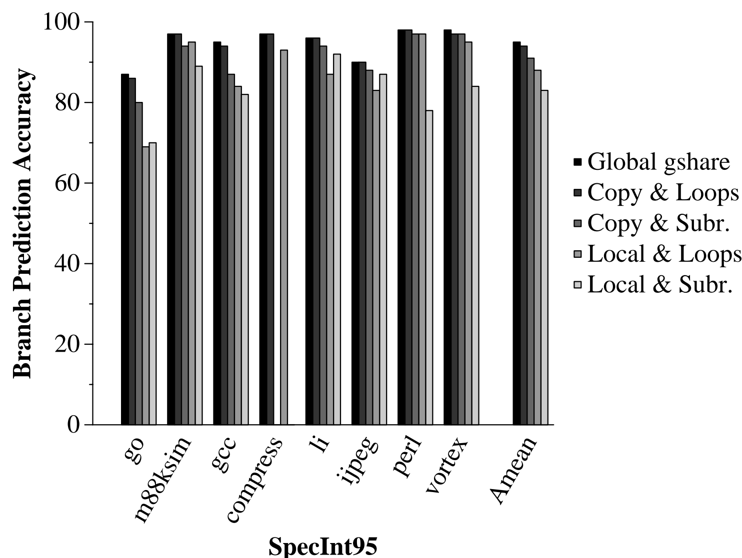


**Figure 2.7.** Branch prediction accuracy.

the subroutine-continuation one), but it is significant when predictors are independently managed (higher than 10% on average).

The degradation in the overall performance of the speculative multithreaded processor for both schemes will be shown in Chapter 5.

## 2.5. COMMITTING SPECULATIVE THREADS

When a speculative thread reaches the control quasi-independent point of any other active thread, it stops fetching instructions and stalls until its control and data speculation is verified, that is, it waits until the processor is assured that it has executed correct code with the correct instructions and becomes the non-speculative thread.

On the other hand, when the non-speculative thread reaches the control quasi-independent point of any other active thread, it verifies that the control quasi-independent point corresponds to the next thread in program order. If not, an order misspeculation has occurred. An order misspeculation may cause that a value produced by a younger thread has been forwarded to an older one. In this case, different actions can be taken, the most conservative one is to squash all the threads and continue with the execution of the non-speculative one. A more aggressive approach is to squash the intermediate threads and verify the values of the next thread. If a data misspeculation has occurred, then the correct values are forwarded to such thread and all dependent instructions and selectively reissued.

If the order prediction is correct, then the non-speculative thread has finished its work and has to verify that the speculation done is correct. Verification schemes for the data input values of a thread will be analyzed in the next Chapter.

If the verification is correct, the next thread becomes the non-speculative one and the thread unit in which the previous non-speculative thread was allocated is freed. Before freeing the thread unit, the processor has to ensure that the values are committed. Memory values stored in the local caches of the non-speculative thread can be written to the main memory or just marked as committed. In the latter case, when a speculative thread is allocated to that thread unit, it will update the main memory when it replaces such committed lines. This model eliminates the burst traffic at the commit time of the non-speculative thread but it may complicate the management of memory dependences.

## 2.6. Related work

The Multiscalar[16][68] architecture is a clustered processor made up of several execution units interconnected by means of an unidirectional ring topology. Besides, the logical order of the tasks matches with the physical order to minimize the communication latency of dependent values. Such execution units are similar to a superscalar core and each of them executes only one task. When an execution unit becomes free, the task predictor predicts the next task to be executed and is allocated to a free thread unit. The Multiscalar processor also has a head and tail pointer that moves through the ring indicating the non and the most speculative task respectively.

Trace processors [59] have a clustered design and the instructions executed in each cluster correspond to traces built by the trace cache. It also uses trace prediction to determine which is the next trace to be executed when a processing element becomes free.

Another clustered design is the Superthreaded Architecture[76], which uses a unidirectional ring for interconnecting the Thread Processing Units. The logical and the physical order of the threads is the same.

Some other works have studied the performance of different speculative multithreaded models in an on-chip multiprocessor such as the Atlas[10], the Hydra[24], the TLDS[71] and the IACOMA[34] groups among others. The Atlas multiprocessor consist on 8 processors interconnected by means of a bidirectional ring and the spawning model used is the sequential thread ordering in order to match the physical and the logical order of the threads.

Stanford's Hydra is an on-chip multiprocessor made up of several MIPS processors fully interconnected. In this design, it does not exist direct interconnection among the processors in such a way that register values are forwarded through memory (by means of the register passing buffer). Similarly, the IACOMA and the TLDS works present an on-chip multiprocessor with capabilities to support speculative multithreading. In all the cases, the order of the spawned threads is known at the spawning time since they spawn threads at well-known program constructs and their logical order can be easily detected.

Finally, the Dynamic Multithreaded Processor[2] uses a centralized design similar to a Simultaneous Multithreaded Processors. Such processor implements a certain order predictor to order the new spawned thread with respect to the existent ones. It also uses a preemptive spawning model if there is no thread units available for spawning new threads. Thus, if there are no context available and the thread to be created is older that any active thread, the most speculative thread is squashed and the new thread is spawned at such context.

## 2.7. SUMMARY

In this chapter, the main hardware requirements to support the speculative multithreaded execution model have been presented. This execution model is based on spawning threads at spawning pairs. The spawning point is the instruction that when reached, fires the creation of the speculative thread. The control quasi-independent point is the instruction where the spawned thread starts its execution.

Different architectural platforms have been presented such as a centralized and a clustered design with different interconnection topologies. Support for speculative multithreading includes a multiversion register file and a multiversion cache and the schemes for spawning and committing threads.

The spawning process has been described and it includes the identification of available contexts to spawn the new speculative thread, the identification of the logical order relationship of the spawned thread with respect to the other active threads and the initialization of the context of the speculative thread. A thread order predictor has been presented and it achieves very high hit ratios (higher than 95%). Moreover, the penalties due to fetching instructions out of program order have been studied and two proposals have been evaluated. The first one is based on copying the branch prediction information from the parent thread to the spawned thread. This mechanism achieves results close to an idealized branch predictor which manages branches in program order. The second one does not initialize the branch prediction tables when a new thread is spawned. This latter mechanism achieves lower hit ratios but the cost in hardware is much lower than the previous one.

Finally, the committing process is described. In this kind of architectures, only the non-speculative thread is allowed to commit its values whereas the speculative threads have to wait to become the non-speculative thread to do it. In case of misspeculation, a simple squash of the speculative thread or a selective reissue of the misspeculated instructions can be implemented.