

---

# Contents

<b>I</b>	<b>This is a Part</b>	<b>7</b>
<b>1</b>	<b>Multithreading and Speculation</b>	<b>11</b>
	<i>Pedro Marcuello, Jesus Sanchez and Antonio Gonzalez</i>	
	Intel-UPC Barcelona Research Center; Intel Labs; Universitat Politecnica de Catalunya; Barcelona (Spain)	
1.1	Introduction . . . . .	11
1.2	Overview of Speculative Multithreaded Architectures . . . . .	13
1.3	Helper Threads . . . . .	17
1.3.1	Building Helper Threads . . . . .	18
1.3.2	Microarchitectural Support for Helper Threads . . . . .	18
1.4	Speculative Architectural Threads . . . . .	19
1.4.1	Thread Spawning Schemes . . . . .	20
1.4.2	Microarchitectural Support for Speculative Architectural Threads . . . . .	23
1.5	Some Speculative Multithreaded Architectures . . . . .	27
1.6	Concluding Remarks . . . . .	27
	<b>References</b>	<b>29</b>
	<b>References</b>	<b>29</b>



---

## *List of Tables*



---

## *List of Figures*

1.1	Helper Thread example . . . . .	15
1.2	Speculative Architectural Thread example . . . . .	16
1.3	Example of execution in a Speculative Multithreaded Processor	21



**Part I**

**This is a Part**



## Symbol Description

$\alpha$	To solve the generator maintenance scheduling, in the past, several mathematical techniques have been applied.		algorithms have also been tested.
$\sigma^2$	These include integer programming, integer linear programming, dynamic programming, branch and bound etc.	$\theta\sqrt{abc}$	This paper presents a survey of the literature
$\Sigma$	Several heuristic search algorithms have also been developed. In recent years expert systems,	$\zeta$	over the past fifteen years
$abc$	fuzzy approaches, simulated annealing and genetic	$\partial$	in the generator maintenance scheduling.
		sdf	The objective is to present a clear picture of the available recent literature
		ewq	of the problem, the constraints and the other aspects of
		bvcn	the generator maintenance schedule.



# Chapter 1

---

## *Multithreading and Speculation*

**Pedro Marcuello, Jesus Sanchez and Antonio Gonzalez**

*Intel-UPC Barcelona Research Center; Intel Labs; Universitat Politecnica de Catalunya; Barcelona (Spain)*

1.1	Introduction .....	11
1.2	Overview of Speculative Multithreaded Architectures .....	13
1.3	Helper Threads .....	17
1.4	Speculative Architectural Threads .....	19
1.5	Some Speculative Multithreaded Architectures .....	27
1.6	Concluding Remarks .....	27

---

### 1.1 Introduction

Moore's law states that the number of available transistors per chip is doubled at each process generation. Such continuous advance in technology has allowed processor designers to incorporate new and more powerful features in each processor generation to run faster the applications. Processor microarchitecture has evolved from single-issue in-order pipelined processors to current superscalar architectures that can run simultaneously multiple threads in the same processor core. These microarchitectures that can exploit both instruction-level parallelism (ILP) and thread-level parallelism (TLP) are known as *multithreaded processors*.

In previous chapters of this book, speculation techniques to extract higher degrees of ILP in dynamically-scheduled superscalar processors have been discussed. Branch prediction and complex fetch engines provide a continuous flow of instructions to the back-end; data speculation diminishes the penalties due to data dependences; and prefetching and precomputation techniques reduce the cost of high-latency operations such as memory instructions that miss in cache. We can conclude that the use of speculative instruction-level parallelism has become one of the main assets for the design of current microprocessors. However, the improvement in performance achieved by scaling up current superscalar organizations, even with the most complex speculation mechanisms, is decreasing and approaching a point of diminishing returns.

The evolution of the workloads that run in most computers together with the difficulties to further increase the exploitation of ILP have motivated researchers to look for alternative techniques to increase the performance of

the processors. An approach that has been reflected in some recent processors is based on the exploitation of different sources of parallelism, namely, fine-grain (instruction) and coarse-grain (thread) level parallelism. This has given rise to the so called multithreaded and multi-core processors.

Exploiting thread-level parallelism is an old idea and it has been widely studied in the past, especially in the context of multiprocessor architectures. Traditionally, thread-level parallelism has been exploited in a non-speculative manner, that is, parallel threads always commit, even though speculative instruction-level parallelism is exploited for each thread.

The main sources for thread-level parallelism have been basically two: 1) different applications that are run in parallel, and 2) parallel threads generated from a single program through the compiler/programmer support.

In the former case, threads run independently one from each other and no communication among them is required. The main benefit of this scheme is an increase in throughput (number of works finished per time unit). However, the individual execution time of a single application may increase when it is executed simultaneously with some others. This is due to the fact that the instructions of each application have to compete for some shared resources (e.g., cache memories, branch predictors, etc.) with instructions from other applications.

On the other hand, in the latter case, threads correspond to different pieces of the same program that are executed in parallel. In this case, partitioning the application into small parts and running them concurrently can significantly reduce the execution time of the application with respect to a single-threaded execution. However, unlike the previous model, threads are highly coupled and the processor has to provide support for communicating and synchronizing the parallel threads.

Partitioning applications into parallel threads may be straightforward for some programs, such as some numeric and multimedia applications, but it is very hard for many others.

Compilers often fail to find thread-level parallelism because they are conservative when partitioning a program into parallel threads, since they have to guarantee that their parallel execution will not violate the semantics of the program. This significantly constrains the amount of thread-level parallelism that the compiler can discover.

By allowing threads to execute in a speculative way and being able to squash them in case of a misspeculation, the scope for exploiting thread-level parallelism is significantly broaden. Speculative threads are not allowed to modify the architectural state of the processor until their correctness is verified. Then, if a thread's verification fails, the work done by the speculative thread is discarded and a roll-back mechanism is used to return the processor to a correct state. On the other hand, if the verification succeeds, the work performed by the thread is allowed to be committed. This parallelism exploited by speculative threads is referred to as *speculative thread-level parallelism*.

In order to exploit speculative thread-level parallelism, some requirements are needed. First, hardware and/or software support for executing speculative threads is necessary. This hardware/software should include mechanisms to hold the speculative state until it can be safely committed, mechanisms for communicating and synchronizing concurrent threads, and verification and recovery schemes. In particular, inter-thread data dependence management has a significant impact on the design and performance of these processors.

In addition to that, a strategy for deciding which speculative threads are to be spawn is also necessary. What instructions speculative threads are to execute, at which point such threads are to be spawned and conclude their work, how a misspeculation on these threads is to affect the processor and the steps required for recovery dramatically impact on the performance of the processor too. Besides, the hardware requirements may be different depending on the speculation strategy. The combination of the hardware/software support and the speculation strategy defines a *speculative multithreaded architecture*.

In this chapter, different proposals for speculative multithreaded architectures are discussed and their main hardware/software requirements are analyzed. We distinguish two main families of architectures. In the first family, speculative threads do not alter the architectural state of the processor but they just try to reduce the cost of high-latency instructions through the side-effects caused by speculative threads on some subsystems of the processor, such as prefetching on caches or warming up branch predictors. The speculative threads executed by this model are referred to as *Helper Threads* (HT). In the second family, each speculative thread executes a different part of the code and computes its architectural state that is only committed if the speculation is correct. In this execution model, the speculative threads are referred to as *Speculative Architectural Threads* (SAT).

The rest of the chapter is organized as follows. Section 1.2 introduces basic concepts on speculative thread-level parallelism and outlines the main features of both families for exploiting it. Section 1.3 presents the Helper Thread family and analyzes its main features. The Speculative Architectural Threads family is presented in Section 1.4 and the main requirements for this model are analyzed. Some examples of proposed speculative multithreaded architectures are presented in Section 1.5. Finally, Section 1.6 summarizes the chapter and points out some future issues for this execution paradigm.

---

## 1.2 Overview of Speculative Multithreaded Architectures

Speculative Multithreading is a technique based on speculation that tries to reduce the execution time of an application by means of running several speculative threads in parallel. Threads are speculative in the sense that

they can be data and control dependent on other threads and their correct execution and commitment are not guaranteed, unlike the traditional non-speculative multithreading exploited conventionally. Speculative thread-level parallelism is the additional parallelism obtained by these speculative threads on a speculative multithreaded processor.

To reduce the execution time of applications, several approaches based on speculative threads can be considered. The main differences among them reside on the strategy to decide which speculative threads are spawned, the requirements to concurrently execute such speculative threads (i.e., if they require to communicate or to synchronize among them) and how speculative threads are spawned and committed or squashed. These features can be combined to exploit speculative thread-level parallelism in different ways.

There are two main approaches to exploiting speculative thread-level parallelism that have been widely studied: Helper Threads and Speculative Architectural Threads.

The Helper Thread paradigm ([8] [20] [28] [38] [39] among others) is based on the observation that the cost of some instructions severely hurt the performance of the processor, such as loads that miss in cache or branches that are incorrectly predicted. This paradigm tries to reduce the execution time of the applications by means of using speculative threads that cut down the latency of these costly operations.

The Helper Thread paradigm consists of a main non-speculative thread that executes all the instructions of the code and some speculative threads that help to decrease the execution time of the main thread by means of some side effects. For instance, these speculative threads may be used to prefetch memory values in order to reduce cache misses when they are required by the main thread, or to precompute some branches that are hard to predict.

There are several manners to build the speculative threads as it is described in the next section of the chapter, but basically a speculative thread contains previous dynamic instructions in the control-flow and data-dependence graph of a long-latency instruction. The speculative thread executes these instructions before the non-speculative thread does. It does not change the architectural state of the processor but modifies the microarchitectural state (e.g., warms up the cache or the branch predictor) so that when the main thread executes the critical instruction, it is executed faster.

Figure 1.1 shows an example of the Helper Thread paradigm. Assume that the bold load always misses in cache and then, the dependent instructions (the add) is delayed until this instruction is satisfied. Below the code, the helper thread for that load is presented. The speculative thread is executed in a different context (thread unit 1) and performs the load before, so that when it is executed by the main thread, it does not miss in cache since it has been prefetched. To do that, the speculative thread consists of the instructions that the load depends on. The performance of this strategy will depend on the ability to spawn the helper thread with enough anticipation to perform the prefetch and keep the requested line in cache until the load is executed.

**Assembly Code:**

```
mul R3, R1, R5
store R3, (R1)
mul R1, R2, R9
add R2, R4, R4
div R3, R1, R4
load R8, (R2) ← Cache Miss
add R8, R8, R3
```

**Helper thread for the load:**

```
add R2, R4, R4
load R8, (R2)
```

**TU0**

```
mul R3, R1, R5
store R3, (R1)
mul R1, R2, R9
add R2, R4, R4
div R3, R1, R4
load R8, (R2) ← Cache hit
add R8, R8, R3
```

**TU1**

```
add R2, R4, R4
load R8, (R2) ← Prefetch
```

Main thread  
is executed faster

**Execution with the helper thread**

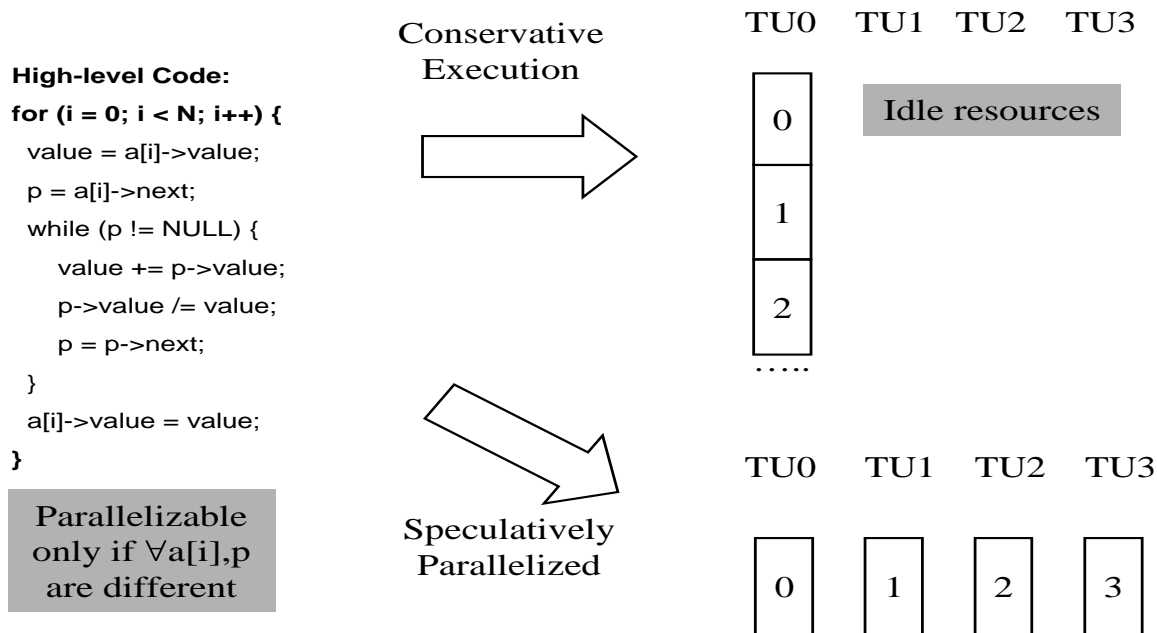
**FIGURE 1.1:** Helper Thread example

In this execution model, the communication among threads is straightforward since speculative threads do not modify any architectural state of the processor. They may reuse values from the non-speculative thread and modify microarchitectural structures also used by the non-speculative thread as pointed out above.

In the Speculative Architectural Thread paradigm ([1] [6] [22] [31] among others) there is also one non-speculative thread at any point in time and speculative threads, but unlike the previous approach, all threads contribute to compute architectural state. In other words, the program is parallelized in the conventional sense, but in a speculative manner. The speculative nature of these threads may cause that the work done by a speculative thread is useless since it may execute with incorrect inputs. In this case, this thread is squashed and is not allowed to modify the architectural state of the processor.

When the non-speculative thread reaches the first instruction of a running or terminated speculative thread, a verification process checks whether such speculative thread is correct (that is, it has not violated any data nor control dependence). If so, the non-speculative thread finishes, its resources are freed for future use by another thread and such speculative thread becomes the new non-speculative one. If the verification fails, then the speculative thread is squashed and the current non-speculative thread continues executing instructions in a normal way. Note that the verification has not to be totally performed at the point when the non-speculative thread reaches a speculative one. Part of the verification can be performed before, as the threads are executed, to anticipate the detection of misspeculations.

Figure 1.2 shows an example of the Speculative Architectural Thread ap-



**FIGURE 1.2:** Speculative Architectural Thread example

proach. The outer loop cannot be parallelized since the compiler cannot assure that the pointers in vector  $a$  and the corresponding linked lists are always different. This fact may be critical since the inner loop modifies the field value of the linked elements. However, the outer loop can be speculatively parallelized by spawning a speculative thread for every iteration assuming there will not be dependences among iterations. If a misspeculation occurs due to a data dependence violation, then the work done by the speculative thread is discarded. Note that, if this loop is executed on a conventional processor, the iterations will be executed serially as it is shown in the figure and some resources will be wasted. However, if the speculation is correct, several iterations can be performed in parallel and the execution time of the application is reduced.

In both speculative multithreaded models, there are two main factors that have a significant impact on the performance of the processor. One of them is the thread spawning scheme. Speculative threads are obtained from applications using different techniques (compiler support with/without profiling, run-time mechanisms, etc.). The features of the selected threads will dramatically impact on the performance of the processor. For Helper Threads, speculative threads should alter the architectural state of the processor before the main thread executes the target instruction. On the other hand, for Speculative Architectural Threads, features such as control and data independence, data predictability, load balance and coverage are key for performance.

In addition to that, hardware/software support for executing speculative threads is necessary. The requirements for spawning, verifying and committing threads as well as providing support for storing the speculative state have a significant impact on the cost and performance of these systems. In particular, the way inter-thread data dependences are managed strongly affects the required hardware/software and the resulting performance.

---

### 1.3 Helper Threads

A helper thread consists of a set of instructions such that, when executed in parallel with the main thread, they are expected to produce a beneficial side effect on the latter. A helper thread does not compute a part of a program, but executes a piece of code that may improve the execution of the main thread. All instructions of the program must still be executed by the main thread. The main objective of helper threads is to speedup some long latency instructions such as loads and branches. In the case of loads, a helper thread prefetches the data to avoid a long latency memory access in the main thread. For branches, a helper thread may help by precomputing the branch outcome. Other possible uses of helper threads are feasible but loads and branches have been the main target of the proposals so far.

The benefit of such technique depends on which loads or branches are selected to be affected by helper threads. Trying to help any load or branch in a program in a blind manner would usually mean a too high overhead that may even offset the potential benefits, in addition to significant increase in power budget. That results in a reduction in the efficiency of the technique and in a waste of resources and power. Fortunately, caches and branch predictors work quite well for the majority of loads and branches respectively. Thus, helper threads should be only used to help those cases where the standard mechanisms are inefficient. We will refer to these instructions as critical instructions.

There are several ways to identify critical instructions: based on heuristics [20], based on profiling [8], or based on dynamic history of the instructions [39]. Though it is not necessary, the first two approaches are usually employed when helper threads are constructed statically by either the compiler or an assembly/binary optimizer, whereas dynamic detection of critical instructions is used when helper threads are built at run time.

The typical execution model for helper threads consists of triggering the execution of the helper thread from the main thread some time before the critical instruction is actually executed. This trigger may be an explicit instruction inserted statically or an event detected by a hardware mechanism. In any case, the time ahead at which the trigger is launched is an important

decision since the goal is to solve the critical instruction before its actual execution. If that is done earlier than needed, the result must be kept in some place, wasting storage, or may be lost. On the other hand, if the trigger is performed too late, then the benefit may be reduced or even disappear.

### 1.3.1 Building Helper Threads

Another important issue is how helper threads are built. There are two main approaches proposed in the literature. The first one is based on identifying the backward slice of a critical instruction [37] and selecting a subset of it that is referred to as a pre-computation slice. This subset usually represents the minimum dependence sub-graph that is needed to correctly execute the critical instruction most of the times (to compute the effective address and access memory in case of a load, or to compute the register/s implied in a conditional branch in case of the branch). There are also different ways to identify the static instructions that make up the speculative thread. The two most common cases consists of: 1) marking in the original binary the instructions that correspond to the helper thread (this means the addition of a field to each instruction to indicate whether it belongs to a given helper thread or not), or 2) copying instructions from the main thread to a new section in the program. In any case, the length of the pre-computation slice will, among other issues, determine how ahead the helper thread trigger must be inserted. As the execution of the helper thread will not affect the correctness of the main thread (it may just speed up its execution), the pre-computation slice could be speculatively optimized. Memory disambiguation and branch pruning have been proposed among others optimizations.

A different approach to build a helper thread can be found in the Subordinate Multithreading framework [3], in which the instructions that form a helper thread do not necessarily belong to the program. A subordinate thread code consists of micro-code in the internal ISA of the machine. This micro-code may contain several routines, each of them implementing a certain algorithm to be applied on specific events (for instance, caches misses or branch missprediction). This allows the application to use powerful and adaptable algorithms to deal with such events that would be very costly to implement in hardware.

### 1.3.2 Microarchitectural Support for Helper Threads

Since the execution of a helper thread does not modify the architectural state of the processor, no verification or recovery mechanisms are needed, no matter if the work done by helper thread is successful or not.

Regarding implementation issues, the processor must include some mechanism to pass data between the main thread and the helper threads. For instance, in the case of helper threads for data prefetching, the data cache can be shared between the main and the helper threads. Simultaneous mul-

tithreaded architectures [34] (e.g., Intel's Hyperthreading [26]) are good candidates for implementing such technique.

---

## 1.4 Speculative Architectural Threads

In the Speculative Architectural Thread paradigm, the main idea is to execute different sections of a program in parallel and speculatively, in such a way that if the execution is correct, the values computed by speculative threads can be committed and the code executed by them does not need to be re-executed by the main thread. Thus, the objective of this technique is to parallelize a program as opposed to speeding up the main thread. In any case, this technique is orthogonal to that of helper threads and they can be combined.

The problem of parallelizing a program (or a section of it) has been extensively studied in the past. Traditional parallelizing compilers try to find independent sections of code (typically loop iterations) or sections of code with few dependences, and add the corresponding synchronization and communication operations to honor such dependences. Though this technique has shown to be effective for scientific codes, in which most of the time the program is executing loops that manipulate matrix-like structures, it is much less effective for other types of codes. For example, the abundance of pointers and the use of irregular memory access patterns make it very difficult for the compiler to disambiguate most memory dependences and many opportunities to parallelize code may be lost.

If we relax this constraint of generating code that honors any possible dependence, the opportunities to parallelize a code greatly increase. In front of a possible dependence, one way to speculate is to ignore the dependence if it is likely not to occur. Another way to speculate is to try to predict or precompute the values that will be produced through these dependences, if they are likely to occur.

The potential of Speculative Architectural Threads is greater than that of Helper Threads since side effects that can be achieved through the latter can also be achieved by the former, and only the former can parallelize a code. On the other hand, Speculative Architectural Threads require a more complex hardware.

In a Speculative Architectural Thread architecture there is a non-speculative (or main) thread always running, whereas the rest of the threads (if any) are speculative. Threads have a predecessor-successor relationship among them that reflect the order in which their instructions would be executed in a single-threaded processor. Only the non-speculative thread, which corresponds to the oldest thread, is allowed to commit its state. In some implementations,

the only thread allowed to spawn speculative threads is the non-speculative one [40], whereas in other implementations, any thread can spawn others [1] [22].

During its execution, a speculative thread goes through different stages. First, it is created and some time is spent to allocate the resources for the new thread and initialize the different structures needed (e.g., register file, etc.). Afterwards, depending on the type of speculative multithreading, the next stage consists of identifying the inter-thread dependences and computing the thread live-ins. After that, the thread starts the execution of the body code until the end of the thread is found. After that, the thread is validated and, if it results to be correct, its produced values are committed.

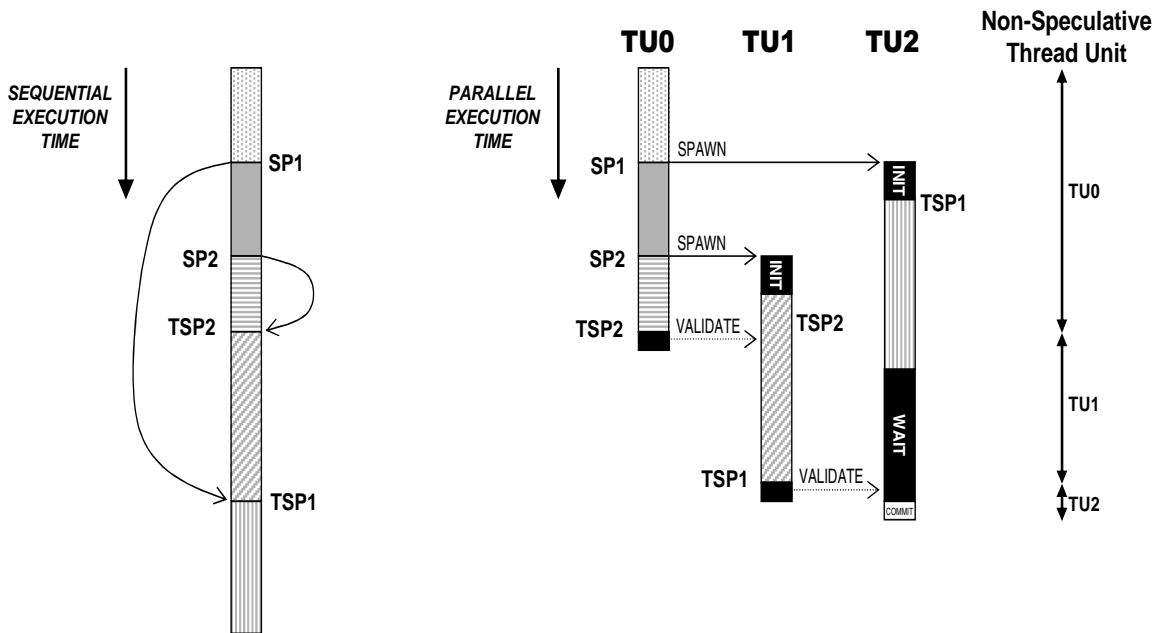
In the next subsections, different schemes for partitioning the applications into speculative threads are presented as well the basic support for executing speculative architectural threads.

#### 1.4.1 Thread Spawning Schemes

A thread spawning mechanism tries to identify which parts of the programs are the most suitable to be executed by speculative threads. That is, at which points of the program speculative threads should be spawned, which part of the code is to be executed by speculative threads and when the speculation should be validated. The different ways these issues are handled result in different spawning schemes.

For any speculative thread, two basic points in the program stream must be determined: when the spawn of it is triggered, and where the new thread starts. Hereinafter, we will refer to these two points as *spawning point* and *thread start point* respectively. A *spawning pair* is the spawning point and the thread start point of a given thread.

Figure 1.3 illustrates the execution model. Assume that two spawning pairs have been found in this application. These pairs are identified as (SP1,TSP1) and (SP2,TSP2). The program starts executing in Thread Unit 0. When this thread finds a spawning point, then a new speculative thread is allocated in a free thread unit, in this case, thread unit 2. The spawned thread will execute instructions starting from the corresponding thread start point, that is, TSP1. However, there is an initialization overhead that includes the setup of all the structures of the thread unit. A discussion of this initialization is presented in following subsections. Meanwhile, the non-speculative thread continues executing instructions and both threads finally proceed in parallel. When the non-speculative thread reaches a new spawning point (SP2), a new speculative thread is spawned at thread unit 1 and these three threads run in parallel. When the non-speculative thread reaches the thread start point of any active thread, in that case TSP2, it stops fetching instructions. Then, a validation process starts to verify that the speculation was correct. If so, the non-speculative thread is committed and the thread unit is freed for future use. Afterwards, the speculative thread on thread unit 2 reaches the end of



**FIGURE 1.3:** Example of execution in a Speculative Multithreaded Processor

the program. As it is not the non-speculative thread, it has to wait until all previous threads finish and validate their speculation. Then, when the non-speculative thread (running on thread unit 1) reaches TSP1, it verifies the speculation and commits, and the thread that is stalled at thread unit 2 is allowed to commit.

Which sections of code are executed by speculative threads can be determined either at compile time or run time. In the former case, the compiler performs an analysis of the code and inserts special instructions in the program that determine the spawning pair of each speculative thread. In the latter, a specialized hardware is responsible for analyzing run-time statistics and detect effective spawning pairs.

The effectiveness of a thread spawning scheme may be computed in different ways: coverage, ratio of successful speculations, average active threads per cycle, etc. However, the metric that really determines the goodness of a spawning scheme is the execution time. If an application with a given spawning policy is executed faster than with other, the former is better. It is possible that a policy with lower coverage and less active threads per cycle outperforms others in execution time. For instance, if speculative threads are data dependent on previous ones, they may have to wait for the computation of the dependent values whereas if speculative threads are almost independent they may proceed in parallel most of the time. Certain properties that may

improve the quality of the speculative threads are:

- **Control independence:** The probability of reaching the thread start point from the spawning point must be very high. Otherwise, this thread will be cancelled at some time in the future. It will not contribute to increase performance but it will consume energy and may prevent other useful threads from being executed.
- **Thread size:** The distance between the spawning point and the thread start point should not be too small nor too large to keep the thread size into a certain limit. Small threads result in too much spawning overhead and large threads may require large storage for speculative state.
- **Data independence:** Threads may have data dependences among them. Instructions after the thread start point (the instructions that will correspond to the speculative thread) should have few dependences with instructions between the spawning and the thread starting point. The more dependences, the more difficult to predict/precompute the thread input values, and the more likely to fail due to a misspeculation.
- **Workload balance:** The ideal scenario is when the maximum number of threads are always active and doing useful work. This may not happen due to several reasons: a spawning point is not found immediately after finishing a thread, a thread is waiting for some event (for instance, to be validated), a speculative thread is doing some initializations, or a speculative thread is misspeculated and squashed.

The previous list is not exhaustive. Depending on the particular speculative multithreaded architecture, some other criteria can be considered such as data predictability if we use data value speculation to deal with data dependences.

Some thread spawning schemes are based on simple heuristics. In some proposals speculative threads are assigned to well-known program constructs that may provide certain of the desired features mentioned above. The three most studied schemes are loop iterations, loop continuation and call continuation [1] [9] [27] [23] [24] [33]. Spawning on loop iterations tries to parallelize different iterations of the same loop. For spawn on loop continuation the spawning point is the beginning of a loop and the thread start point is the exit of the loop. Similarly, for spawn on call continuation the spawning point is the beginning of a subroutine and the thread start point is the instruction after the return. A similar approach to this is spawning threads at module-level instead of subroutine-level and it was studied in [36]. The common feature of these schemes is high likelihood of reaching the thread start point after the spawning point is reached. There are some other heuristics also based on cache misses such as presented in [6].

Other thread spawning schemes are based on a more sophisticated analysis of the code in order to quantify some relevant metrics, as the one described

above, that can help to identify the sections of the code that can most benefit from being speculatively parallelized. This schemes can be implemented in the compiler [25] [35] and generate special code for speculative multithreaded processors or at run-time [4]. This last technique includes the re-compilation of Java applications.

### 1.4.2 Microarchitectural Support for Speculative Architectural Threads

The design space for Speculative Architectural Threads architecture is huge. The main required feature for supporting this execution model is to provide mechanisms to deal with multiple threads simultaneously and some special extensions to deal with the special nature of the speculative threads. These additional features are related to the communication and synchronization among speculative threads, how speculative threads are spawned and committed and how the speculative state is kept.

Speculative Architectural Threads have been studied in different platforms such as centralized or monolithic processors and clustered processors. Centralized speculative multithreaded processors are similar to simultaneous multithreaded processors and almost all the subsystems of the processor are shared among concurrent threads. Sharing has the benefit of low communication latencies and better resource usage, but it may hurt the performance due to resource contention or degradation of the performance of several systems such as cache memories.

On the other hand, clustered processors reduce resource contention, and the size of clustered structures, such as register files, caches, which may reduce their latency. However, clustering increases the communication and synchronization latency.

In the next subsections, some insight in the most important features of a speculative multithreaded processor, such as the spawning and the committing process, the storage of the speculative state and the management of inter-thread data dependences are discussed.

#### 1.4.2.1 Spawning Process

A speculative thread may be spawned through a special instruction inserted in the binary by the compiler or when some hardware or run-time system detects a certain event (e.g., a call to a subroutine or a cache miss).

The spawning process includes several non-negligible tasks such as looking for an idle context, computing the logical position of the thread among all the active threads and initializing the values for the spawned thread.

Assigning a context for a spawned thread is an easy task if there are available contexts. In that case, the new speculative thread is spawned at one of them. If there are no free contexts, different solutions can be taken: 1) not to spawn the thread, 2) wait for a free context and 3) squash one thread and

spawn the new one at the recently freed context.

Since there may be data dependences among speculative threads and it is possible that values flow from one speculative thread to another, the logical order among threads must be known by the processor. When a speculative thread is created, it is placed in the correct order in the list of threads. The approach to identify the order may depend on the spawning scheme. If we consider a processor where only the most speculative thread is allowed to spawn threads, the placement function is straight-forward since the new spawned thread is always the most speculative one. However, in a processor where all active threads are allowed to spawn new threads, it is not as easy. In that case, the processor may implement a thread order predictor based on previous history.

Finally, in order to perform useful work, speculative threads have to be executed with the correct input values. Thus, it is necessary to initialize the register and memory values that are to be used by the spawned thread. If such values are available at the spawning point, they can be directly copied from the spawner thread to the new spawned thread. If the values have to be produced by a preceding thread, they may be predicted as described in later subsections or a synchronization may be required.

#### **1.4.2.2 Committing Process**

A thread may finish when it finds a special instruction or when it reaches the thread start point of any other active thread. When this occurs, this thread stops fetching instructions and stalls until its control and data speculation is verified. That is, it waits until this thread becomes the non-speculative.

Thus, when the non-speculative thread reaches the thread start point of any other active thread, it verifies that it corresponds to the next thread in sequential program order. If this is not the case, an order misspeculation has occurred. This situation may have caused that a value produced by a younger threads was incorrectly forwarded to an older one. In addition to the thread order, all the input values used by a thread have to be validated (i.e., those consumed but not produced by itself). The verification of the input values can be done either when the non-speculative thread has reached the thread start point, or can be done on-the-fly as values are produced.

The validation process should include the comparison of all register and memory values of the speculative thread when it was spawned and the final values of the non-speculative thread. In fact, it is only necessary to check that the input values that have been consumed by the speculative thread match with the final values.

In case of misspeculations, the speculative thread is cancelled and the produced state must be discarded. A recovery mechanism is needed in order to roll back the processor to a safe state, which can be implemented through the same techniques used to recover from other types of speculation such as branch or value missprediction. In general, either full squash of the misspeculated

thread and its successors or a selective reissue scheme can be implemented.

Note that, when a misspeculation is detected, all threads more speculative than the offending one may also incur in misspeculations. Then, recovery actions in all threads more speculative than the offending one are required.

#### 1.4.2.3 Storage for speculative state

The fact that all threads work on the same register and memory space implies that each variable may have potentially a different value for each of the threads running concurrently. This requires a special organization of the register file [2] [17] and the memory hierarchy [11] [13] [17] [32] [29], which represents the main added complexity of this paradigm.

In both cases, two solutions may be considered: 1) separate structures for each thread (physical partitioning), that is, different register files or different caches for the speculative threads; and 2) a large shared structure but with separate locations for the different versions (logical partitioning), that is, a large register file and separate map tables or a large cache that supports multiple values for every location. In the first case, the access time is usually lower but communicating values is more costly. On the other hand, sharing reduces the communication latency but affects the scalability and increases the access time.

#### 1.4.2.4 Dealing with Inter-Thread Dependences

Ideally, speculative threads should be both control and data independent. In this way, the concurrent execution of these speculative threads would not require any communication/synchronization. However, as it is expected, such kind of speculative threads are hard to find for many applications. Therefore, speculative multithreaded processors have to provide mechanisms to deal with inter-thread dependences. The way such dependences are managed will strongly affect the performance of such processors.

A speculative thread is data dependent on a previous thread if it consumes any value produced by it. Thus, an inter-thread data misspeculation may occur when a speculative thread reads a variable before it is produced by the previous thread.

The values that are consumed by a speculative thread are normally referred to as thread inputs or thread live-ins. Those thread live-ins that are not available when the thread is spawned are those that are critical for performance.

Dealing with inter-thread data dependences includes several activities: 1) identifying the thread live-ins of a speculative thread; 2) providing the dependent value to the consumer thread; and 3) detecting when a speculative thread has consumed a wrong value.

The identification of the thread live-ins is relatively straightforward for register values and can be easily performed with compiler or run-time techniques. However, it is more difficult for memory values. Compilers usually are unable to statically determine the memory locations accessed and often take a

conservative approach. To predict thread memory live-ins, data dependence prediction techniques such as those explained in Chapter 9 can be used.

Regarding how speculative threads obtain the dependent values, we can distinguish two different families of solutions: 1) synchronization mechanisms, and 2) value prediction schemes. These families will also impact on how misspeculations are detected. Obviously, a hybrid scheme that combines both can also be considered.

Synchronization techniques stall the execution of dependent instruction of a speculative thread until the value is forwarded from the producer thread. This scheme requires to identify the producer instruction of each thread live-in and hardware support for forwarding the dependent values from the producer to the consumer.

The simplest synchronization scheme is to stall the execution of a speculative thread until all live-ins are produced and validated. These schemes have been shown to significantly constraint the performance that can be obtained through a Speculative Architectural Thread architecture since in most cases, a thread cannot know all the values of its inputs until all previous threads have practically completed [24]. Code reorganization may help by moving up the producer instructions in the producer thread and moving down the consumer instructions in the consumer thread [33].

In order to obtain some benefit, synchronization schemes should accurately identify when dependent values can be forwarded in order to avoid unnecessarily stalls in the speculative threads. Similarly to the identification of the thread live-ins, identifying when a register value is available to be forwarded is relatively easy to do statically, but harder for memory values. In that case, data dependence prediction techniques may also help.

In the case of inter-thread memory dependences, synchronization mechanisms based on data dependence speculation have been widely used. Hardware examples of that for speculative architectural threads are Address Resolution Buffer for Multiscalar architectures [11], the Speculative Versioning Cache [13], the Memory Disambiguation Table [17], the Thread-Level Data Speculation [32] among others. On the other hand, there are some software approaches to deal with inter-thread data dependences [29] that consists of associating to each shared variable a data structure to allow multiple threads to access to it.

For synchronization mechanisms the performance we can expect from Speculative Architectural Threads is limited by the critical path of the data dependence graph. In order to further boost performance, a mechanism that breaks the serialization imposed by data dependences is required. Thus, the second family of techniques to deal with data dependences is based on value prediction [19] [30]. Data value speculation is a technique that can relieve the cost of serialization caused by data dependences and boost the performance of superscalar processors as it has been shown in Chapter 9. Predicting the input/output operands of instructions before they are available allows the processor to start the execution of those instructions and their dependent ones

speculatively. The study in [12] concludes that the potential of that technique in platforms such as multithreaded processors is very promising.

Both hardware and software data predictors has been used so far for speculative multithreaded processors. Hardware schemes use similar type of predictors to those proposed for superscalar (see Chapter 9), though predictors specialized for multithreading have also been proposed [23]. A different approach is to perform the prediction in software. Distilled programs are an example on this type of value prediction [40]. The idea is to include in the binary some code that quickly computes the live-ins. This code is executed before the speculative thread starts.

The detection of misspeculated predicted values requires to store the value obtained by the prediction in order to compare them with the current ones once they are available.

---

## 1.5 Some Speculative Multithreaded Architectures

There are several proposals in the literature for speculative multithreaded architectures:

For instance, examples of Helper Threads that build slices from the applications are Data-Driven Multithreaded Architecture and the Backward Slices [28] [38] [39], the Delinquent loads [8] [18], and Luk's work [20] among others. On the other hand proposals where helper threads are not built from the applications are the Simultaneous Subordinate Microthreading [3].

For Speculative Architectural threads, pioneer work was the Multiscalar [10] [31]. After that, other schemes have been proposed such the Superthreaded [33], the SPSM architecture [9], the Dynamic Multithreaded Processor [1] and the Clustered Speculative Multithreaded processors [21] [23] among others. Moreover, schemes for exploiting speculative thread-level parallelism on a on-chip multiprocessor such as the Atlas [6] [7], the Stanford's Hydra [4] [14] [15] [27], the IACOMA project [17] [5], the Agassiz project [33] [16], the STAMPede project [32] and the Chalmers University project [29] [36] among others have been proposed.

---

## 1.6 Concluding Remarks

A shift on the main paradigm exploited by microprocessors has occurred in the past about every decade. During the 80's, RISC was the main paradigm, followed by superscalar in the 90's, and multithreaded in the current decade.

Speculative Multithreaded processors may become the main paradigm for the next decade. Speculative multithreading is targeted to exploit thread-level parallelism in order to increase performance. Thread-level parallelism is not new, but the novelty comes from exploiting it in a speculatively way. This opens a lot of new opportunities in terms of performance potential and challenges in terms of complexity and power. In this chapter we have reviewed some of the most relevant schemes proposed so far. However, speculative multithreading still requires significant research efforts before becoming a mainstream technology.

Two main families of speculative multithreaded processors have been investigated so far: Helper Threads and Speculative Architectural Threads. The former requires a much simpler support at the microarchitectural level but the latter has a much higher potential to increase performance. Each one represents a different trade-off between cost and performance that may be attractive for different workloads and/or market segments.

A main challenge to be further investigated is the scheme to identify and spawn speculative threads, which can rely on compiler technology, run-time schemes or a combination of both. Another important aspect to further investigate is the approach to dealing with inter-thread dependences. In the Speculative Architectural Threads model, another important challenge is the design of a complexity-effective memory organization for managing the multiple versions of each variable, some of them speculative, that are alive simultaneously. Solutions to these challenges already exist but these are the key areas where more innovation is required in order for this paradigm to become a mainstream technology.

Overall, the various schemes proposed so far suggest that speculative multithreading may be the next paradigm to keep performance growth on the Moore's curve.

---

## References

- [1] AKKARY, H. AND DRISCOLL, M.A. "A Dynamic Multithreading Processor", *Procs. of the 31st Int. Symp. on Microarchitecture, 1998*
- [2] BREACH, S., VIJAYKUMAR, T.N. AND SOHI, G.S. "The Anatomy of the Register File in a Multiscalar Processor", *Procs. of the 25th Int. Symp. on Computer Architecture, pp. 181-190, 1994*
- [3] CHAPPEL, R.S. ET AL. "Simultaneous Subordinate Microthreading (SSMT)", *Procs. of the 26th Int. Symp. on Computer Architecture, pp. 186-195, June 1999*
- [4] CHEN, M. AND OLUKOTUN, K. "TEST: A Tracer for Extracting Speculative Threads", *Procs. of the Int. Symp on Code Generation and Optimizations, pp. 301-312, 2003*
- [5] CINTRA, M., MARTINEZ, J.F. AND TORRELLAS, J. "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems", *Procs. of the 27th Int. Symp. on Computer Architecture, 2000*
- [6] CODRESCU, L. AND WILLS, D. "On Dynamic Speculative Thread Partitioning and the MEM-slicing Algorithm", *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 40-46, 1999*
- [7] CODRESCU, L., WILLS, D. AND MEINDL, J. "Architecture of Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications", *IEEE Transaction on Computers, vol 50(1), pp. 67-82, January 2001*
- [8] COLLINS, J.D. ET AL. "Speculative Precomputation: Long Range Prefetching of Delinquent Loads", *Procs. of the 28th Int. Symp. on Computer Architecture, 2001*
- [9] DUBEY, P.K. ET AL. "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grain Multithreading", *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 109-121, 1995*
- [10] FRANKLIN, M. AND SOHI, G.S. "The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism", *Procs. of the 19th Int. Symp. on Computer Architecture, pp. 58-67, 1992*
- [11] FRANKLIN, M AND SOHI, G.S. "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", *IEEE Transaction on Computers, vol 45(6), pp. 552-571, May 1996*

- [12] GONZALEZ, J. AND GONZALEZ, A. "The Potential of Data Value Speculation to Boost ILP", *Procs. of the 12th Int. Conf. on Supercomputing*, pp. 21-28, 1998
- [13] GOPAL, S. ET AL. "Speculative Versioning Cache", *Procs. of the 4th Int. Symp. on High Performance Computer Architecture*, 1998
- [14] HAMMOND, L. ET AL. "The Stanford Hydra CMP", *Micro IEEE*, vol. 20 (2), pp. 6-13, March 2000
- [15] HAMMOND, L., WILLEY, M. AND OLUKOTUN, K. "Data Speculation Support for a Chip Multiprocessor", *Procs. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998
- [16] KAZI, I.H. AND LILJA, D.J. "Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors", *Procs. of the 12th Int. Conf. on Supercomputing*, pp. 93-100, 1998
- [17] KRISHNAN, V. AND TORRELLAS, J. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip Multiprocessor", *Procs. of the 12th Int. Conf. on Supercomputing*, pp. 85-92, 1998
- [18] LIAO, S.S. ET AL. "Post-Pass Binary Adaptation for Software-based Speculative Precomputation", *Procs. of the Int. Symp. of Programming Languages, Design and Implementation*, 2002
- [19] LIPASTI, M.H., WILKERSON, C.B AND SHEN, J.P. "Value Locality and Load Value Prediction", *Procs. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, 1996
- [20] LUK, C. "Tolerating Memory Latency Through Software-controlled Pre-execution in Simultaneous Multithreading Processors", *Procs. of the 28th Int. Symp. on Computer Architecture*, pp. 40-51, 2001
- [21] MARCUELLO, P., TUBELLA, J. AND GONZALEZ, A. "Speculative Multithreaded Processors", *Procs. of the 12th Int. Conf. on Supercomputing*, pp. 76-84, 1998
- [22] MARCUELLO, P. AND GONZALEZ, A. "Clustered Speculative Multithreaded Processors", *Procs. of the 13th Int. Conf. on Supercomputing*, pp. 365-372, 1999
- [23] MARCUELLO, P. AND GONZALEZ, A. "Value Prediction for Speculative Multithreaded Architectures", *Procs. of the 32nd Int. Symp. on Microarchitecture*, pp 230-236, November. 1999
- [24] MARCUELLO, P. AND GONZALEZ, A. "A Quantitative Assessment of Thread-Level Speculation Techniques", *Procs. of the 15th Int. Symp. on Parallel and Distributed Processing*, 2000

- [25] MARCUELLO, P. AND GONZALEZ, A. "Thread Spawning Schemes for Speculative Multithreaded Architectures", *Procs. of the 8th Int. Conf. on High Performance Computer Architecture*, 2002
- [26] MARR, T. ET AL. "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology Journal*, vol. 6(1), 2002
- [27] OPLINGER, J., HEINE, D. AND LAM, M. "In Search of Speculative Thread-Level Parallelism", *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 303-313, 1999
- [28] ROTH, A. AND SOHI, G.S. "Speculative Data-Driven Multithreading", *Procs. of the 7th Int. Symp. on High Performance Computer Architecture*, pp.37-48, 2001
- [29] RUNDBERG, P. AND STENSTROM, P. "A Low Overhead Software Approach to Thread-Level Data Dependence on Multiprocessors", *Chalmers University, Sweden, TR-00-13, July 2000*
- [30] SAZEIDES, Y. AND SMITH, J.E. "The Predictability of Data Values", *Procs. of the 30th Int. Symp. on Microarchitecture*, 1996
- [31] SOHI, G.S., BREACH, S. AND VIJAYKUMAR, T.N. "Multiscalar Processors", *Procs. of the 22nd Int. Symp. on Computer Architecture*, pp. 414-425, 1995
- [32] STEFFAN, J. AND MOWRY, T. "The Potential of Using Thread-Level Data Speculation to Facilitate Automatic Parallelization", *Procs. of the 4th Int. Symp. on High Performance Computer Architecture*, pp. 2-13, 1998
- [33] TSAI, J.Y. AND YEW, P-C. "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation", *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35-46, 1996
- [34] TULLSEN, D.M., EGGERS, S.J. AND LEVY, H.M. "Simultaneous Multithreading: Maximizing On-chip Parallelism", *Procs. of the 22nd Int. Symp. on Computer Architecture*, pp. 392-403, 1995
- [35] VIJAYKUMAR, T.N. "Compiling for the Multiscalar Architecture", *Ph.D. Thesis, University of Wisconsin at Madison*, 1998
- [36] WARG, F. AND STENSTROM, P. "Limits on Speculative Module-level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms", *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 221-230, 2001
- [37] WEISER, M. "Program Slicing", *IEEE Transaction on Software Engineering*, vol. 10(4), pp. 352-357, 1984

- [38] ZILLES, C.B. AND SOHI, G.S. "Understanding the Backward Slices of Performance Degrading Instructions", *Procs. on the 27th Int. Symp. on Computer Architecture*, pp. 172-181, 2000
- [39] ZILLES, C.B. AND SOHI, G.S. "Execution-Based Prediction using Speculative Slices", *Procs. of the 28th Int. Symp. on Computer Architecture*, pp. 2-13, 2001
- [40] ZILLES, C.B. AND SOHI, G.S. "Master/Slave Speculative Parallelization", *Procs. of the 35th Int. Symp. on Microarchitecture*, pp. 85-96, 2002