1 INTRODUCTION

Future processor generations will provide capabilities to exploit both instruction and thread level parallelism and will be able to execute multiple threads concurrently. In this chapter, an approach to make use of such capabilities when there is a lack of non-speculative threads is presented: The technique consists of relaxing the constraints to parallelize applications in order to exploit speculative thread level parallelism. Finally, this chapter presents the main contributions of this thesis and details the organization of the document.

1.1. Motivation

It is well-known that the performance of processors depends on the amount of parallelism they can extract from the applications. As more tasks processors are able to correctly execute in parallel, the total time needed to fulfill the execution of the applications is drastically reduced. Parallelism can be exploited at different levels, such as instruction level, application level, etc., but in latest years, almost all the efforts and studies have been devoted to improve the ability to exploit fine-grain or instruction level parallelism. This kind of parallelism is characterized by the fact that different instructions are allowed to execute simultaneously each cycle with the only constraint that those imposed by the dependences, both control and data dependences. For instance, the main characteristic of superscalar processors[30], which are the most popular processor microarchitecture in the recent years, is their ability to exploit the instruction level parallelism that inherently exist in the programs.

To be able to exploit such parallelism, processors have to incorporate several mechanisms to detect whether an instruction can start their execution without affecting the correct behavior of programs as well as schemes to store those instructions that cannot execute until their operands are available. The cost of these mechanisms depends on how aggressively the processor tries to extract instruction level parallelism. In-order superscalar processors rely on the compiler to reorganize the code statically and then simplify the hardware requirements. These processors look for independent instructions in program order until a non-independent instruction is found. Then, the issue process stalls until the operands of this instruction are available and it can be issued. On the other hand, dynamically-scheduled (also known as out-of-order) superscalar processors fetch instructions in program order and store them into a buffer (which is referred to instruction window). The run-time scheduler is then responsible for issuing those instructions from the instruction window whose operands are available independently of the program order.

Despite the theoretical performance of this kind of microarchitectures in terms of instructions executed per cycle (IPC) can be high, in lots of applications, specially in non-numerical programs, the results achieved are quite far from the peak performance since the exploitation of the instruction level parallelism is much limited by several hurdles such as[85]:

- *Control hazards*: To exploit instruction level parallelism, a continuous flow of instructions has to be provided by the front-end of the processor to the issue mechanism. Conditional branch instructions affect the performance in the way that the front-end of the processor does not know exactly which would be their following instruction until the branch is solved.

- *Data hazards*: Data dependences are another important limitation for instruction level parallelism since instructions only can start their execution when they have all their operands available. In fact, the execution ordering imposed by data dependences strongly restricts the performance achieved by superscalar processors. Experiments with ideal machines[21] with unbounded resources and just the constraint of serializing data dependent instructions report an average IPC of just several tens of instructions committed per cycle for some integer programs.

In order to minimize the impact of the above hurdles to achieve higher degrees of instruction level parallelism, most of the processors use speculation techniques to reduce the penalties of both control and data dependences. Speculation techniques consist of predicting the result of instructions before executing them and using such results to start the execution of dependent ones. Speculation techniques can be applied for different purposes:

- Control speculation: To reduce the effect of conditional branch instructions, current processors predict the outcome of the branches and speculatively execute the instructions following the predicted path ([67] among others). When the branch outcome is solved, the prediction is verified and, and if a misprediction has occurred, the processor has to eliminate the incorrect instructions, recover the correct state, and continue the execution starting at that correct point. In this way, processors can maintain a continuous flow of instructions from the front-end of the processor to the instruction window and therefore, have more instructions to look for independent ones.
- Data value speculation: Relieve the penalties due to data dependences and minimize their impact on the performance of the processor by means of predicting the input/output operands of instructions ([37] among others). When a misprediction occurs, the processor has to squash the incorrect instructions and recover the correct state. Execution can continue starting at the point where the misspeculation happened or just reexecuting those instructions that depend on the mispredicted one (this technique is referred to as selective reissuing). In this way, data value prediction breaks the data dependence graph and instruction level parallelism can be improved since the number of eligible instructions for starting execution will be greater.

Nevertheless, speculation techniques are not enough to boost the performance of such processors since in some applications, the amount of instruction level parallelism is very limited. Branch prediction can help to maintain a large instruction window, but the number of useful instruction, i.e, the effective size of the instruction window, depends on the accuracy of the branch prediction since a single mispredicted branch prevents the instruction window from growing beyond the branch and the following instructions become useless. This phenomenon is due to the sequential nature of the branch prediction which introduces that the effective size depends on the number of consecutive branches that are correctly predicted. Even though the overall performance of commercial branch predictors is quite high, above 95%, for many applications, specially in non-numerical programs, which have lots of difficult-to-predict branches, the prediction rate is reduced causing a significant drop in the performance of the processor.

On the other hand, speculating on data values is harder than speculating on the outcome of branches since the range of values is huge compared with the branch prediction outcome, whose only possible values are taken or not taken. This wide range makes very difficult to design accurate value predictors. Moreover, recent studies have shown that the performance impact of this technique for superscalar processors is moderate, even though its impact on the performance may be higher on a more appropriate environment such as multithreaded architectures[22].

As a conclusion, the performance of superscalar processors, despite of the speculation techniques, are still far away from the peak performance these processors can obtain. Also, although technology evolution allows computer architects to use an increasing number of transistors[66], the performance improvement achieved just by scaling up the current superscalar microarchitecture is decreasing and approaching to diminishing return points. Wire delays, bypass logic, the issue and the rename logic among others have been identified as one of the most critical delays in such architectures[54].

These limits in instruction level parallelism have motivated researchers to look for alternative techniques to increase the performance of processors. Examples of these techniques are the multimedia extensions ([56][74] among others) to reduce the execution time of the ever increasing multimedia workloads (sound, video, etc...). Another approach considered by several vendors is combining the exploitation of different sources of parallelism, for instance exploiting both fine-grain (instruction level) parallelism and coarse-grain (thread-level) parallelism ([12][49][73][75] among others). With this technique, processors include some hardware to manage several threads and run them simultaneously. These threads can make use of those resources of the processor that otherwise would remain idle since, as it is mentioned above, instruction level parallelism is not enough to fulfill them (no independent instructions available, etc...) in lots of applications. In this thesis, these processors that are able to exploit both instruction level and thread level parallelism will be referred to as *multithreaded processors*.

Exploiting thread level parallelism is not a novel idea and it has been widely studied, specially in the field of multiprocessor architectures. Traditional sources of thread level parallelism that have been studied are basically two: executing different applications in the same processor core and parallelizing the applications with the compiler/programmer support. In the former case, each thread executed in the multithreaded processor corresponds to a different application in such a way the resources that would be idle in the sequential execution of those processes, now are used to execute instructions of other applications causing a beneficial increase of the throughput (number of processes finished per time unit) since the execution of the processes simultaneously will finish faster than the sum of the individual execution time of each application. In the latter case, programs are divided into threads that can run in parallel in the processor with some synchronization mechanisms. Here, the execution time of the parallelized application will be lower than if the application would be executed sequentially.

Nonetheless, these traditional techniques to exploit thread level parallelism have not completely solved the problem of boosting the performance of processors. Executing different applications simultaneously produces a higher throughput but the execution time of the individual applications is also increased if it is compared with its execution alone. This feature is due to the fact that now, instructions of an application have to compete, in addition to the instructions of the same application, with those instructions from the other applications for the same resources. Also, several subsystems of the processor can be negatively affected by the concurrent use of them by several processes, like cache memories and branch predictors.

On the other hand, partitioning applications into threads may be an straightforward task in regular or numerical applications. For those programs, current compiler technology can easily perform it efficiently. However, this task becomes much harder for irregular or non-numerical programs; compilers usually fail to discover the thread level parallelism that could be effectively exploited in this class of applications.

The benefits of thread level parallelism are impressive for those applications that can be parallelized, the execution time of them being drastically reduced. However, those applications that cannot be parallelized do not get any benefit at all. Compilers usually fail because they are quite conservative: partitioning is only performed on those parts of the program that at compile-time, the compiler is sure that their parallel execution will not affect the final result of the program. This fact causes that in irregular or non-numerical applications, compiler is almost not able to find any thread that accomplish such heavy criterias. This fact causes that in irregular or non-numerical applications, compiler is almost not able to find any thread that accomplish such heavy criterias.

In the recent years, several studies (this thesis among others) have proposed to relax these constraints to increase the benefits of thread level parallelism. The proposal is analogous to the way followed by the instruction level parallelism, exploiting it speculatively. Instruction level parallelism is enhance by speculative execution if the path or values have been correctly predicted by the speculation engine. In the same way, if the partitioning constraints are relaxed and the processor is able to speculatively spawn and execute threads that may be in the wrong path (control dependent) or executed with incorrect input values (data dependent), an extra gain could be achieved if the speculation engine correctly predicts the path and the input values. This technique is referred to as *speculative thread level parallelism* in opposition to the traditional techniques that exploit non-speculative thread level parallelism. Obviously, roll-back mechanisms that take the processor back to the correct state in case of misspeculation will be necessary in these processors.

1.2. Exploiting Speculative Thread-Level Parallelism

Speculative thread level parallelism can be defined as a technique targeted to reduce the execution time of applications by means of running in parallel several threads that belong to a single application. Threads spawned by this technique are speculative in the sense they could be data or control dependent on previous threads and their correct execution and commitment are not guaranteed, unlike the non-speculative thread level parallelism exploited by the traditional mechanisms.

Different strategies for exploiting speculative thread level parallelism can be considered depending on how the multithreaded processor try to reduce the execution time of programs. There are three big paradigms to achieve this goal, each one made up of lots of different variations depending on how programs are partitioned, when threads are spawned and the way data dependences among concurrent threads are managed by the processor. Previous strategies are not necessary to be considered alone. Processors can combine them to exploit speculative thread level parallelism in different ways. These three main strategies are:

- *Eager execution*: This technique is focused on minimizing the effect of branch mispredictions in the processor performance ([57] among others). As it was mentioned in the previous subsection, branch prediction is a speculation technique used to increase the instruction level parallelism by means of predicting the outcome of a branch before this branch instruction is executed. If the prediction is correct, instructions from the correct path have been fetched and may have started its execution. However, if the branch predictor fails, all the instructions from the wrong path have to be squashed and a correct state has to be recovered. All the tasks involved in recovering a mispredicted branch usually waste sever cycles that the processor cannot use to do useful work.

  Studies done in branch behavior have shown that most of the mispredictions are just provoked by a small number of static branches. The *eager execution* technique tries to reduce the impact of mispredictions of such branches by means of spawning two speculative threads when a branch is found, one thread starting at the target instruction of the branch (as if the branch were taken) and the other in the

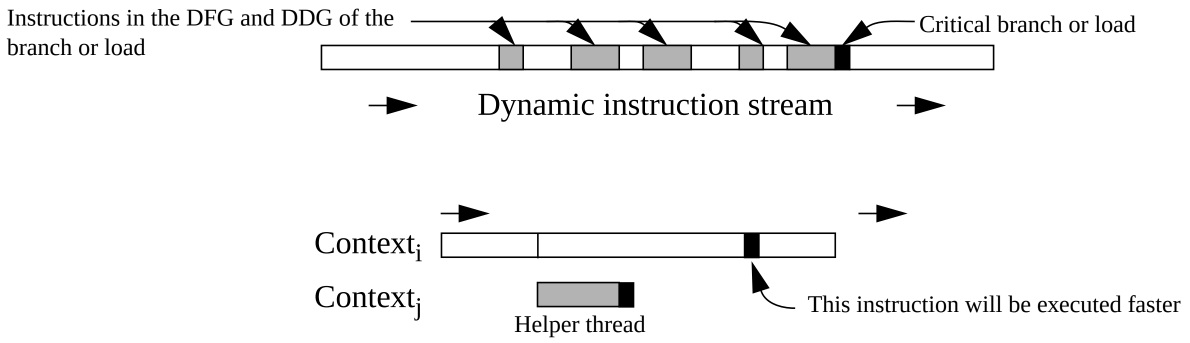
Instructions in the DFG and DDG of the branch or load
Critical branch or load
Dynamic instruction stream
Context$_i$
Context$_j$
Helper thread
This instruction will be executed faster

Figure 1.1. Execution model of a speculative multithreaded processor with helper threads.

following instruction in static order (as if the branch were not taken). Once the branch is solved in the main thread, the speculative threads corresponding to the wrong path are discarded and the execution continues with the speculative thread that has followed the correct path (which at this point becomes the non-speculative thread or the main thread). To avoid the exponential growth of the number of threads, this technique is just applied for those branches that are difficult to predict. With this technique, if the cost of managing several speculative threads is less than the average cost of recovering from mispredicted branches, then a potential benefit can be achieved.

• *Helper threads*: Some types of instructions have a significant impact on the performance of the processor. Load instructions from memory can spend many cycles if the value requested is far away in the memory hierarchy and it will prevent dependent instructions to start their execution until this instruction is completed. On the other hand, conditional branches may prevent the processor from fetching instructions beyond this instruction until it is solved. Speculation techniques can be used to predict the value of the load or the outcome of the branch, but it is still important to execute these instructions as soon as possible to early detect misspeculations and start the roll-back mechanisms.

Helper threads are focused on reducing the execution time of such critical instructions ([7][11][61] among others). An instruction needs to have all their operands available to start its execution so it has to wait for the completion of all the previous instructions in their data and control dependence graph. These instructions have also to compete with others instructions of the program for using the resources of the processor. The technique of *helper threads* tries to give priority to these instructions in the data and control dependence graph of the critical instructions. Figure 1.1 shows how this strategy works. A critical instruction is detected by different methods (compiler, profiling, dynamically at run time, etc...) and the previous instructions in the control and data dependence graph of this critical instruction are identified. Then, the processor proceeds as follows, context *i* executes the main thread until it

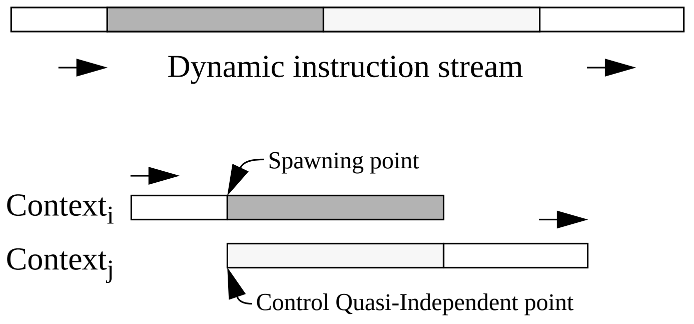

Figure 1.2. Execution model of a speculative multithreaded processor with speculative multithreading.

detects that it will soon execute a critical instruction. Then a *helper thread* is created in an idle context. This speculative thread only executes those instructions in the control and data dependence graph of the critical instruction. Meanwhile, the main thread continues executing sequentially the whole program. When the critical instruction is executed by the main thread, since it was executed before in the *helper thread*, their execution will be faster (for instance, if the instruction is a load, the *helper thread* may have prefetched the value to cache, or if it is a branch, prediction may not be necessary since it has already been computed).

• *Speculative Multithreading*: In previous techniques, speculative threads are used to reduce the impact of several types of instructions in the overall performance of the processor, but in none of them speculative threads reduce the amount of instructions that the main thread has to execute. On the other hand, in the technique of *speculative multithreading* each speculative thread is responsible for executing a part of a program in parallel with some other threads ([16][44] among others).

Programs are partitioned into speculative threads by different techniques (compiler techniques, profiling, dynamically, etc.). This partitioning process tries to identify pairs of instructions (that will be referred as *spawning pairs*) that will provide the best performance. A spawning pair is made up by two instructions, the *Spawning Point*, which is the instruction that will fire the creation of the speculative thread and the *Control Quasi-Independent Point* which is the instruction where the speculative thread will start its execution. Figure 1.2 shows how a speculative multithreaded processor works: Programs run sequentially in a single hardware context until a *Spawning Point* is reached. Then, a speculative thread is spawned in an idle context. The thread starts its execution at the *Control Quasi-Independent Point* an both, the non-speculative and the speculative threads proceed in parallel until the non-speculative reaches the *Control Quasi-Independent Point*. Then, the non-speculative thread finishes and the following speculative thread in logical order becomes the non-speculative one. This model can be generalized and more speculative threads can be created even by other speculative threads. In this case, when a speculative thread reaches the *Control Quasi-Independent Point* of a future speculative thread does not commit until it becomes non-speculative.

Processors that are able to exploit speculative thread level parallelism will be referred to as *speculative multithreaded processors*. Spawning instructions may be added by the compiler or inserted at run-time. When a thread finishes, if no misspeculations have occurred, the thread is committed and its context freed for future use of other speculative threads. Otherwise, the thread is discarded and a roll-back mechanism is used to recover a correct state.

Speculative multithreaded processors include support for multiple hardware contexts, each one with its own local values (register and memory values). Additional hardware for initializing and committing threads is also required. Moreover, synchronization mechanisms are also required in case that values produced by one thread are to be consumed by other thread. Finally, some parts of the processor must be carefully designed in order to avoid important penalties. For instance, the performance of the memory subsystem and the branch predictor may be affected by having multiple threads running in parallel.

The task of detecting the best points to spawn threads can be done at run-time by the processor or by the compiler support. The strategy used to partition a program into threads will be referred to as the *thread spawning policy.*

1.2.1. Related Work

Multithreaded architectures have been studied for long but so far the focus has been the improvement of throughput by executing several independent threads or dependent threads with the necessary synchronization added by the compiler. Examples of architectural paradigms that exploit this kind of non-speculative thread level parallelism are multiprocessors and simultaneous multithreading processors[79][90] among others.

Multistream Processors[90] and SMT (Simultaneous Multithreading Processors)[79] were the pioneer works on simultaneous multithreading. They presented a hardware organization to provide support for managing several threads (streams) simultaneously sharing the resources of the processors among the active threads. In both works, the microarchitecture is just an extension of a superscalar processor with some components being replicated: the register map table, the program counter and the return stack, etc... Instructions are fetched for each thread from a different program counter, renamed separately and stored in a shared instruction window. The wake up and selection logic is also shared. The main benefit of such architectures is a higher throughput due to the better usage of the computer resources.

Regarding the exploitation of speculative thread level parallelism, several works have been performed in the last years for the alternative strategies outlined above.

Eager execution is an old paradigm. Pioneer work was presented by Riseman and Foster in 1972[57] as a technique to extract high degrees of parallelism. In that work a multi-level eager execution that spawns threads for every branch is analyzed. This approach has an exponential cost in terms of the number of simultaneous threads required. Later on, Uht *et al.*[88][82] proposed the Disjoint Eager Execution scheme. In that work, resources are assigned depending on the probability of the path in such a way that only the most likely paths at each moment are being executed. On the other hand, Heil and Smith[27] proposed to use eager execution only for those branches difficult to predict, based on a confidence estimator. In that work, only two threads can proceed in parallel. Klauser *et al.*[33] generalized the previous work for more than two threads. The baseline architecture considered in that work was the PolyPath architecture which is quite close to the architectural model of the simultaneous multithreading processor. Similar work are the Threaded Multiple Path Execution[86] presented by Wallace *et al.* and the Multipath Execution[1] presented by Ahuja *et al.* As in previous works, threads are spawned just for difficult-to-predict branches but only the thread in the most likely path can spawn new speculative threads.

Early work on *helper threads* is the Simultaneous Subordinate Microthreading[7], proposed by Chappel *et al*. In that work, microthreads are spawned and run in parallel with the main thread to help in their execution. These microthreads are implemented in microcode and perform a better or more complex branch predictions and/or data prefetching.

Other recent techniques that exploit speculative thread level parallelism by means of helper threads are based on identifying dynamic sequences of instructions that could potentially have a high impact on performance and assign them to speculative threads. These works use a simultaneous multithreading processor as the underlying hardware support for speculative threads. Hong *et al.* [11][36] proposed to reduce the impact of loads that miss in cache by executing on a speculative thread the instructions that belong to its data dependence graph. In this way, the data was prefetched before the main thread requires it. Luk[39] attacked the same problem in a similar manner. The main differences lie on the way loads are selected (Hong *et al*. use a profile mechanism to detect them, whereas Luk uses locality analysis) and how speculative threads are built. Roth *et al.* [61][69] proposed Speculative Data-Driven Multithreading which is based on executing the instructions that belong to the data dependence graph of some critical instruction (loads and branches) on a helper thread. Zilles *et al.*‘s speculative slices[91][92] uses the same idea but speculative threads contain control flow instructions in addition to the data flow instructions.

Finally, on speculative multithreading, pioneer work on this topic was the Expandable Split Window Paradigm[16] and the follow-up work, the Multiscalar[68]. This microarchitecture has a clustered design and it is made up of several execution units which are responsible for executing the speculative threads (tasks). This execution units are interconnected among them by means of an unidirectional ring in such a way that the communication is done from one thread to the following one in the logical order. Tasks are statically created by the compiler based on several heuristics that try to minimize the data dependences among active threads or maximize the workload balance, among other compiler criteria. Related work was the Multithreaded Decoupled Architecture presented by Dorojevets and Oklobdzija[13]. Its execution model is quite similar to the Multiscalar processor and programs are split into threads by the compiler, but here, speculative threads are constrained not to contain branches.

Other architectures that exploit speculative multithreading has been proposed in recent years. Dubey *et al.* presented the Single-Program Speculative Multithreading architecture[14] (the SPSM architecture). In this proposal, programs are divided into threads at compile-time and the execution of a thread is not started until the dependent values from previous threads that it needs have been produced. Tsai and Yew´s Superthreaded architecture[76] also relies on the compiler to split the program into threads. In this architecture, speculative threads are assigned to different loop iterations. To deal with inter-thread data dependences, the Superthreaded architecture uses the thread pipelining model. In this model, the code of the loops where the speculative threads are to be spawned are reorganized in such a way that the instructions that calculate the input values for the following iterations are promoted to the beginning of the thread to be early available for the consumer speculative threads. Inter-thread data communication is done in this architecture explicitly by new instructions introduced in the instruction set.

A related approach is the scheme used by Trace Processors[59][83]. This architecture splits the sequential program into almost fixed-length traces with the help of the trace cache[60] and executes them into different processing elements, each one similar to a superscalar core. This fact maximizes the workload balance among the different processing elements. Another important feature of Trace Processors is that inter-thread dependent register values are predicted when the speculative thread starts its execution whereas inter-thread memory dependences are assumed not to occur among concurrent threads. To detect misspeculations, a Trace Processor provides a mechanism to detect them and fire a recovery task to restore the correct state by reexecuting only those instructions dependent on the mispredicted value.

Dynamic Multithreaded Processors[2] rely only on hardware mechanisms to divide a sequential program into threads. In this architecture, speculative threads are created in those parts of the program that usually exhibit control independence. In particular, speculative threads are created for every subroutine call and their execution starts at the following instruction in static order. These speculative threads execute the code that normally will be executed when the flow of the program returns from the subroutine. Also, speculative threads are created for every backward branch (since a backward branch is the instruction that usually closes a loop). Speculative threads start at the following instruction in static order and execute the continuation of the loop. The Dynamic Multithreaded Processor has a centralized design similar to the simultaneous multithreaded processor and shares the most of its resources.

Warg and Stenstrom[89] presented an approach to spawning speculative threads similar to that of the Dynamic Multithreading. Here, instead of creating threads at subroutine and loop continuations, threads are created when an invocation of a module is done (procedures, functions, methods, etc...). Another significant difference between these works is that the considered microarchitecture was an on-chip multiprocessor instead a centralized scheme.

Zilles´s *et al* Master/Slave Speculative Parallelization[93] represents a different scheme to exploit speculative thread level parallelism by means of distilled programs. The distilled program is a small subset of instructions of the program that calculate de input values of the threads. In the execution model of this processor, the distilled program runs as a master thread and when the input values of a speculative thread are calculated, then it is spawned on an idle context while the master thread starts calculating new input values for the next thread. When the least speculative slave thread finishes, input values of the next slave thread are verified and if misprediction has occurred, following speculative threads are discarded and the main thread rolls back to the correct state.

Some other works performing speculative thread-level parallelism on multiprocessor platforms have been done. The Altas multiprocessor proposed by Codrescu and Wills[9][10] is an on-chip multiprocessor whose different processing elements are interconnected by means of a bidirectional ring topology and to which support for thread and value speculation has been added. Speculative threads are obtained by means of the MEM-Slicing algorithm[9], which, instead of spawning threads at points of the program with high control independence, spawn threads at memory-access instructions. Similarly, the Hydra processor[24][26][54] is an on-chip multiprocessor with some capabilities that allow it to exploit speculative thread level parallelism by providing mechanisms for data dependence speculation. Here, threads are obtained at compile time, but the constraints imposed by data dependences are relaxed in such a way that more threads can be created and higher degrees of parallelization can be achieved.

Similarly, lots of works have appeared in the literature regarding the relaxation of memory disambiguation constraints to obtain more thread-level parallelism. Steffan and Mowry presented the Thread-Level Data Speculation scheme[71][72]. In this technique, threads are also created by the compiler and misspeculations are detected by means an extended mechanism of the cache coherence protocol. Torrellas *et al*. also relax the parallelization constraints to be able to create more parallel threads, but memory misspeculations are detected by means of the MDT (Memory Disambiguation Table), which keeps information of the memory locations accessed by the speculative threads[34][6]. Another approach considered for exploiting speculative thread level parallelism in on-chip multiprocessors was proposed by Rajwar and Goodman with the Speculative Lock Elision[57]. Multithreaded applications use locks to protect memory accesses that can cause conflicts. These locks are usually included conservatively by the compiler and lots of them are not necessary. In that work, they propose to eliminate some of these locks speculatively which allow more threads to run concurrently.

Other similar works for dynamic parallelization in on-chip multiprocessors are [31][32].

1.3. THESIS OVERVIEW

This thesis is focused on how exploiting speculative multithreading as a main source of speculative thread level parallelism. As it was mentioned above, the execution model to effectively exploit this parallelism is based on identifying which parts of the program are suitable to provide thread level parallelism. To create a new thread two points of the program have to be identified, the *Spawning Point* which is the instruction that spawns the new thread and the *Control Quasi-Independent Point* which is the instruction where the new thread will start their execution[48]. The pairs of *Spawning* and *Control Quasi-Independent Points* are referred to as *spawning pairs*. Programs are executed sequentially until a *Spawning Point* is reached, then a new speculative thread starting at the *Control Quasi-Independent Point* is spawned and both proceed in parallel. When the non-speculative thread reaches the first *Control Quasi-Independent Point*, which is also the join point, it stops its execution and verifies if the speculative thread has been correctly created and if it has been executed with the correct input values. Some verification can also be performed while the speculative threads are being executed. If so, the non-speculative thread is committed, their context is freed and the speculative thread becomes the non-speculative thread. If not, a recovery mechanism is started.

In addition to the execution model, the two principal topics considered in this thesis are the hardware support needed to implement this execution model and the proposal and the study of schemes to partition the programs into speculative threads.

Regarding the hardware support, in this thesis different microarchitectural models are discussed. A key requirement to exploit thread level parallelism at processor level is providing support for managing several contexts simultaneously taking into account that some values used can be shared while others can be private. Note also that each thread may have a different value for the same variable. In other words, the system needs to be able to manage multiple versions of the same variable concurrently. Clustered microarchitectures are good for wire delays, power and complexity but communications among threads can suffer from delays[43][44]. On the other hand, centralized architectures are suitable to share resources and speed up communication among threads[40][42][45].

Related to the hardware support, this thesis pays special attention to the way inter-thread data dependences are managed in speculative multithreaded processors. Different proposals to deal with both register and memory data dependences among threads as well as techniques for thread initialization are considered. In both cases, synchronization (the consumer thread has to wait for the computation of the value by the producer thread) and value prediction mechanisms are analyzed [44][46][47].

Finally, regarding the spawning policies, different proposals based on heuristics that try to exploit control independence are analyzed[47]. The relationship between the spawning policy and the overall performance of the processor is studied. A new proposal that in addition to control independence also takes into account other important factors such as data independence and data value predictability is also proposed[48]. A systematic method to find the best spawning pairs for splitting the program into threads based on an off-line analysis that uses profiling data is proposed in this thesis.

1.3.1. Thesis Contributions

The main contributions of this thesis are:

1) On the execution model and their hardware support:

- An execution model based on creating threads at *spawning pairs* is proposed (pairs of spawning and control quasi-independent points).
- A family of microarchitectures to exploit speculative multithreading, either clustered or centralized.

2) On managing inter-thread data dependences:

- A new value predictor (the increment predictor) specially targeted for this execution model is presented.
- The MultiValue Cache, a special first level architecture of the memory hierarchy to detect memory misspeculations and fast forwarding of dependent memory values among dependent concurrent threads.

3) On the spawning policies:

- Formalize the criteria required by good spawning pairs.
- A systematic method to identify the parts of the code most suitable for speculative thread level parallelism. This scheme is based on an off-line analysis that uses profiling data.

1.4. Document Organization

The rest of this thesis is organized as follows:

- Chapter 2 identifies the hardware requirements of speculative multithreaded processors specially those referred to the creation and commit of the speculative threads. The execution model is also detailed. Different proposals to maintain the speculative state of the threads are analyzed.
- Chapter 3 studies the microarchitectural part related to inter-thread data dependences. An effective mechanism to manage them is necessary in order to achieve good performance and value prediction is shown to be an effective approach. Different conventional value predictors are analyzed and a new one, the increment predictor is proposed.
- Chapter 4 studies different spawning policies and their impact on performance. Partitioning techniques based on heuristics are initially considered, and then, policies based on studying the code of the programs and looking for those points suitable to provide best performance are developed.
- Chapter 5 evaluates the impact on the performance of several subsystems such as branch and value predictors.
- Chapter 6 outlines some future steps and the open research areas and finally summarizes the main conclusions of this thesis.