Speculative Multithreaded Processors

Nicolae-Andrei Vasile

Faculty of Computer Science and Automatic Control

POLITEHNICA University of Bucharest

Bucharest, Romania

Abstract – Speculative multithreading has been recently proposed to boost performance by means of exploiting thread-level parallelism in applications difficult to parallelize. The performance of these processors heavily depends on the partitioning policy used to split the program into threads. Previous work uses heuristics to spawn speculative threads based on easily-detectable program constructs such as loops or subroutines.

I. INTRODUCTION

Speculation is a well-known technique used to improve processor performance. These mechanisms have been widely used in order to reduce the penalties of both control and data dependences. Also, diminishing returns of instruction-level parallelism are boosting the use of alternative techniques to increase performance. Combining thread-level parallelism and instruction-level parallelism is an approach that has been considered by several processor vendors. These types of processors are usually referred to as multithreaded processors. The task of dividing programs into threads that will be executed in parallel is rather straightforward for regular or numeric applications, and the current compiler technology can perform it efficiently. However, this task becomes hard for irregular and non-numerical programs; compilers usually fail to discover the potential thread-level parallelism that could be effectively exploited in this class of applications.

Speculative multithreading is a promising approach to solving this problem. In these systems, threads that may be control and data dependent on previous threads are speculatively spawned and executed. Relaxing the constraints to spawn a thread results in a significant increase of opportunities to exploit thread-level parallelism, even though, obviously, roll-back mechanisms are needed in case of misspeculations.

The performance of speculative multithreaded architectures is very sensitive to the policies that determine which parts of the code are executed by speculative threads and when they start execution. We refer to this criteria as the thread-spawning policy. In several architectures such as Multiscalar, the SPSM architecture and the Superthreaded, the compiler is responsible for dividing the program into speculative threads. Alternatively, the Dynamic Multithreaded Processor [1] and the Clustered Speculative Multithreaded Processor [2], [3] rely only on hardware techniques; programs are partitioned at run-time. The thread-spawning policies proposed so far for speculative multithreaded architectures are very simple. They are based on assigning speculative threads to common program constructs such as loop iterations, loop continuations and subroutine continuations.

A thread-spawning operation is identified by two instructions: 1) the spawning instruction that creates a new thread when it is reached, and 2) the spawned instruction where the speculative thread starts its execution. These instructions are referred to the spawning point and the control quasi-independent point, respectively.

II. SPECULATIVE THREAD-LEVEL PARALLELISM

A thread spawning operation is identified by two instructions in the dynamic instruction stream that we refer to as the spawning and the control quasi-independent points [4]. Each pair of points is referred to as a spawning pair. The spawning point is the instruction that, when reached by the processor, it fires the creation of a new thread. The control quasi-independent point is where the new spawned thread starts. The spawning and the control quasi-independent points can be conventional instructions in the instruction set, denoted with special marks or special instructions such as fork and spawn.

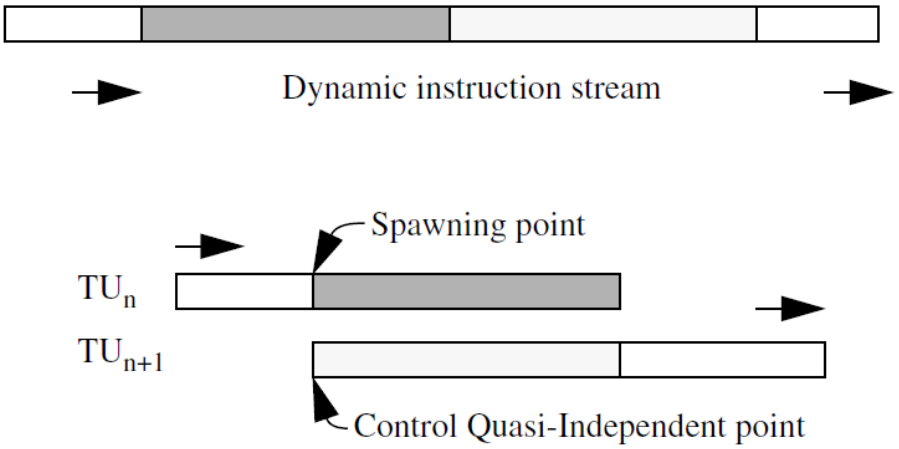

Figure 1. Spawning and Control Quasi-Independent Points

Figure 1 shows how speculative multithreaded processors work. Thread Unit *n* executes the instruction stream in the same way as a conventional superscalar processor until it reaches a spawning point. At this point, the processor identifies a future instruction (the control quasi-independent point) which will very likely be executed in the near future. Then, Thread Unit *n+1* spawns a new thread speculatively starting at the control quasi-independent point while Thread Unit *n* continues executing instructions up to the control quasi-independent point which also becomes the join point between threads. That is, the join point of a thread is a control quasi-independent point of an on-going speculative thread.

It is obvious that the best instructions to be considered as spawning and control quasi-independent points are those where the speculative and the non-speculative thread were control and data independent, in such a way that the processor would be able to execute them concurrently. Unfortunately, these kind of threads are not common in some programs, especially in non-numerical codes, and thus, their potential thread-level parallelism may be rather low.

Effective spawning pairs should satisfy some requirements. First, the probability of reaching the control quasi-independent point after visiting the spawning point should be very high in order to conserve resources (executing instructions that will never be reached). Second, the distance between the spawning point and the control quasi-independent point should not be too small or too large to keep the thread size within a certain limit. Small threads result in too much overhead and large threads may result in work imbalance. Third, instructions after the control quasi-independent point should have few dependences with instructions above it or alternatively, the values that flow through such dependences should be predictable.

Previous thread-spawning policies basically focused on the first criterion:

Loop iterations

Considers the first instruction in static order of a loop (the target of a backward branch) as both the spawning and the control quasi-independent point. Note that once an iteration is started, a further iteration is very likely regardless of the outcome of the branches inside the loop body.

Loop continuation

Considers the first instruction in static order of a loop as the spawning point and the instruction following the backward branch in static order that closes the loop as the control quasi-independent point. Note that after starting a loop, the instruction at the control quasi-independent point is very likely to be executed regardless of the control-flow inside the loop.

Subroutine continuation

Considers a subroutine call as the spawning point and the instruction following the subroutine call in static order (i.e., the point where the subroutine will return) as the control quasi-independent point. Note again that after the call, this latter instruction is very likely to be executed regardless of the path followed inside the call.

III. SPECULATIVE MULTITHREADED PROCESSOR MICROARCHITECTURE

The microarchitecture of a Speculative Multithreaded Processor (SM) is shown in Figure 1. It consists of several thread units (TU) that execute concurrently different threads of a sequential program. These threads are dynamically obtained by a control speculation mechanism based on identifying loops and executing speculatively different iterations of a loop (not necessarily an innermost loop) and they do not need to be independent.

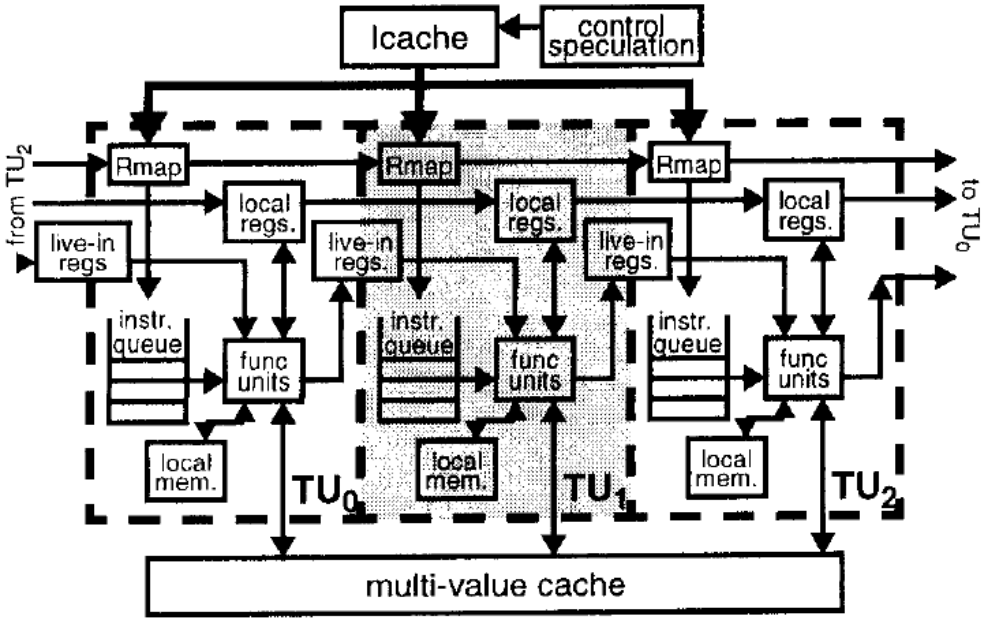

Figure 2. A speculative multithreaded processor with three thread units [3]

Thread units are interconnected through a ring topology and iterations are allocated to thread units following the execution order. Each thread unit has its own physical register file (local registers in Figure 2), register map table, instruction queue, functional units, local memory and reorder buffer in order to execute multiple instructions out-of-order. In this way, the issue bandwidth of a SM processor is scalable and the main bottlenecks observed in superscalar processors (wakeup, select and bypass logic) are avoided. High issue rates can be achieved by increasing the number of thread units, which has no impact on the cycle time.

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

Control speculation

On the one hand it is used to obtain threads from a sequential program. On the other hand, it is used to speculate on individual branches inside each thread like superscalar processors do.

Data dependence speculation

Dependences among different threads (inter-thread dependences) are predicted by means of address prediction. When a thread executes a memory instruction, it is speculatively disambiguated against other memory instructions in previous threads by using the predicted addresses of those instructions if they have not yet been executed. The memory dependence speculation scheme is implemented by means of the multi-value cache. Inter-thread register dependences are managed by identifying at run time which registers hold live values at the beginning of an iteration (live-in registers) and the number of writes that each iteration performs to them. Live-in registers whose value is not predictable are read from the live-in register file after being produced by the previous thread.

Data value speculation

Inter-thread data dependences, either through registers or memory, do not cause a serialization between the producer and the consumer in SM processors, provided that the values that flow through such dependences are predictable. Previous works have shown that many of such values are predictable, including the results of arithmetic instructions, the values read from memory and the register values used to compute the effective address of memory instructions.

Another important feature of SM processors is that they may exploit a large amount of instruction level parallelism with a simple instruction fetching mechanism. This feature is based on the observation that the majority of iterations of the same loop follow the same path. In particular, we have evaluated that the most frequent path of each loop represents about 85% of the total number of iterations for the Spec95 [3]. ,This suggests that if thread units are devoted to execute iterations of the same loop with the same control flow, the instruction fetching mechanism can be shared by all the threads. A single fetch engine fetches a single instruction stream from the instruction cache using a conventional branch predictor. The instructions fetched in each cycle are broadcast to all the thread units where their registers are renamed using a different register map table and then, they are dispatched to their corresponding instruction queue. In this way, the processor peak performance can be equal to the actual fetch bandwidth multiplied by the number of thread units. This organization overcomes one of the most important hurdles of multithreaded architectures. In those machines, the processor is required to fetch from different program counters, simultaneously or alternatively, which makes the fetch engine to be one of the critical parts of such architectures. In SM processors, instructions are always fetched from a single program counter. Each thread validates the predicted intra-thread control flow by executing branch instructions and comparing the result with the prediction. In case of misprediction, this thread and the following ones are squashed.

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

IV. RELATED WORK

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

Control speculation has been extensively researched. Proposed schemes for superscalar processors are based on predicting branches in the sequential order of the program. This means that a single mispredicted branch will cause the squash of every instruction fetched after it. In these schemes, branches that are difficult to predict may prevent the processor from speculating beyond them, even if successive branches are highly predictable. To obtain multiple threads of control, SM processors speculate only on highly predictable branches, and therefore they have more potential to build a large instruction window. On the other hand, speculating on non-contiguous branches results in a non-contiguous instruction window that is more complex to manage [3].

Data dependence speculation is used by some processors in the memory disambiguation stage. However, these techniques assume that any load is independent of all the previous stores whose addresses are unknown. More sophisticated approaches have recently been proposed in order to predict more accurately the existence of a data dependence through memory and avoid some of the expensive recovery actions that are required by misspeculations. Data dependence speculation has also been proposed to improve the parallelization of code for single-chip multiprocessors [2].

There are few proposals in the literature dealing with the dynamic management of a large window that consist of several threads of control not necessarily independent among them obtained from a sequential program. Pioneer work on this area was the Expandable Split Window paradigm and the follow-up work on Multiscalar processors. Other proposals are the SPSM architecture, the Superthreaded architecture the Multithreaded Decoupled architecture, the Dependence Speculative Multithreaded Architecture (DeSM), and Trace processors [2], [3]. There are important differences between the SM microarchitecture and those previous proposals [3]:

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

- Data value speculation has been used in previous proposals, mainly in the context of a superscalar processor with a single thread of control. Data speculation is also used by Trace processors. However, there are significant differences between Trace processors and SM processors. First, SM processors speculate on inter-thread data dependences by predicting all memory references at the beginning of a thread, whereas Trace processors execute speculatively memory instructions based on the predicted value of source operands but these instructions compete with the other instructions for securing issue slots. Thus, when a

SM processor disambiguates a load instruction all the addresses of previous stores have been predicted (assuming that they are predictable) whereas this is not the case of Trace processors. In consequence, Trace processors will experience a much higher number of memory dependence misspeculations, that is, the data dependence speculation mechanism of SM processors is more accurate than that of Trace processors. The second important difference is that Trace processors require a global register file that may become a bottleneck for the scalability of the system, whereas SM processors have all the register files completely distributed. Third, unlike SM processors, the multiple instructions windows simultaneously managed by Trace processors are adjacent. Finally, the approach to build the instruction window is different: whereas in Trace processors it is based on a trace cache, in SM processors it is based on a loop prediction technique.

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

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

V. CONCLUSIONS

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

VI. REFERENCES

[1] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proceedings of 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Dallas, TX, USA, 1998.

[2] P. Marcuello and A. Gonzalez, "Clustered speculative multithreaded processors," in *Proceedings of the 13th international conference on Supercomputing*, Rhodes, Greece, 1999.

[3] P. Marcuello, A. Gonzalez and J. Tubella, "Speculative Multithreaded Processors," in *Proceedings of the 12th International Conference on Supercomputing*, Melbourne, Australia, 1998.

[4] P. Marcuello and A. Gonzalez, "Thread-Spawning Schemes for Speculative Multithreading," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, Cambridge, MA, USA, 2002.