Reducing Misspeculation Penalty in Trace-Level Speculative Multithreaded Architectures

Carlos Molina[Ψ], Jordi Tubella[ϕ] and Antonio González[ϕλ]

[Ψ] Dept. Eng. Informàtica i Matemàtiques, Universitat Rovira i Virgili, Tarragona - SPAIN
[ϕ] Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona - SPAIN
[λ]Intel Barcelona Research Center, Intel Labs-UPC, Barcelona - SPAIN

E-mail:cmolina@etse.urv.es,antonio@ac.upc.es,jordit@ac.upc.es

Abstract. Trace-Level Speculative Multithreaded Processors exploit trace-level speculation by means of two threads working cooperatively. One thread, called the speculative thread, executes instructions ahead of the other by speculating on the result of several traces. The other thread executes speculated traces and verifies the speculation made by the first thread. Speculated traces are validated by verifying their live-output values. Every time a trace misspeculation is detected, a thread synchronization is fired. This recovery action involves flushing the pipeline and reverting to a safe point in a program, which results in some performance penalties. This paper proposes a new thread synchronization scheme based on the observation that a significant number of instructions whose control and data are independent of the mispredicted instruction. This scheme significantly increases the performance potential of the architecture at less cost. Our experimental results show that the mechanism cuts the number of executed instructions by 8% and achieves on average speed-up of almost 9% for a collection of SPEC2000 benchmarks.

1. Introduction

Data dependences are one of the most important hurdles that limit the performance of current microprocessors. Two techniques have so far been proposed to avoid the serialization caused by data dependences: data value speculation [12] and data value reuse [23]. Both techniques exploit the high percentage of repetition in the computations of conventional programs. Speculation predicts a given value as a function of past history. Value reuse is possible when a given computation has already been made exactly. Both techniques can be considered at two levels: the instruction level and the trace level. The difference is the unit of speculation or reuse: an instruction or a dynamic sequence of instructions.

Reusing instructions at trace level means that the execution of a large number of instructions can be skipped in a row. More importantly, as these instructions do not need to be fetched, they do not consume fetch bandwidth. Unfortunately, trace reuse introduces a live-input test that it is not easy to handle. Especially complex is the validation of memory values. Speculation may overcome this limitation but it introduces a new problem: penalties due to a misspeculation. Trace-level speculation avoids the execution of a dynamic sequence of instructions by predicting the set of live-output values based, for instance, on recent history. There are two important issues with regard to trace-level speculation. The first of these involves the microarchitecture support for trace speculation and how the microarchitecture manages trace speculation. The second involves trace selection and data value speculation techniques.

Recently, several thread-level speculation techniques [3], [7], [16], [20] have been explored to exploit parallelism in general-purpose programs. We lay on the same trend and focus on Trace-Level Speculative Multithreaded Architecture (TSMA) [14], which is tolerant to misspeculations in the sense that it does not introduce significant trace misprediction penalties and does not impose any constraint on the approach to building or predicting traces. This paper extends the previous TSMA microarchitecture with a novel verification engine that significantly improves performance. This new engine reduces the number of thread synchronizations and the penalties due to misspeculations. The main idea is that it does not throw away execution results of instructions that are independent of the mispredicted speculation, which reduces the number of instructions fetched and executed again.

The rest of this paper is organized as follows. Section 2 describes in detail the microarchitecture assumed to exploit trace-level speculation. Section 3 analyses the percentage of useful computation that is lost in recovery actions. Section 4 presents the novel verification engine. Section 5 analyses the performance of the processor with the proposed engine. Section 6 reviews related work and Section 7 summarizes our main conclusions and outlines future work.

2. Trace-Level Speculative Multithreaded Architecture (TSMA)

2.1. Trace-Level Speculation with Live-Output Test

Trace-level speculation is a dynamic technique (although the compiler may help) that requires a live-input or live-output test. The microarchitecture presented in this section focuses on the following approach: trace-level speculation with live-output test. This approach was introduced by Rotenberg *et al* [16], [17], [18] as the underlying concept behind *Slipstream Processors*. This approach is supported by means of a couple of threads (a speculative thread and a non-speculative one) working cooperatively to execute a sequential code.

Let us consider the program of Figure 1.a that is composed by three pieces of sequential code or traces. Figure 1.b shows the execution of the program from the point of view of code and Figure 1.c shows the execution of the program from the point of view of time. The speculative thread executes instructions and speculates on the result of whole traces. The non-speculative thread verifies instructions that are executed by the speculative thread and executes speculated traces. Each thread maintains its own state but only the state of the non-speculative thread is guaranteed to be correct. Communication between threads is done by means of a buffer that contains the executed instructions by the speculative thread. Once the non-speculative thread executes the speculated trace, instruction validation begins. This is done by verifying that source operands match the non-speculative state and updating the state with the new result. If validation does not succeed, recovery actions are required.

Note that speculated traces are validated by verifying their live-output values. Live-output values are those that are produced and not overwritten within the trace. The advantage with this approach is that only live-output values that are used are verified. Moreover, verification is fast because instructions consumed from the buffer have their operands ready (see trace3 execution and validation in Figure 1.c). Finally, speed-up is obtained when both threads execute instructions at the same time and validation does not

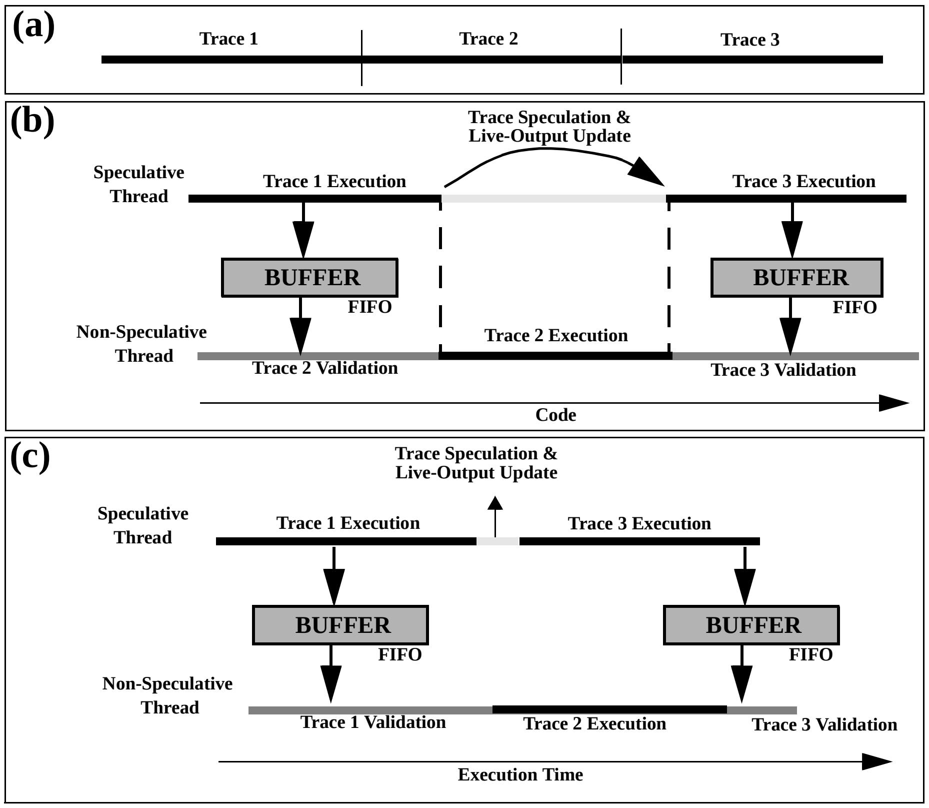

Figure 1. Trace-level speculation with live-output test
(a) program (b) point of view of code (c) point of view of time

produce a misspeculation, which implies to set some recovery actions (see trace2 and trace3 execution in Figure 1.c).

2.2. Microarchitecture

A TSMA processor can simultaneously execute a couple of threads (a speculative one and a non-speculative one) that cooperate to execute a sequential code. The speculative thread is in charge of trace speculation. The non-speculative thread is in charge of validating the speculation. This validation is performed in two stages: (1) executing the speculated trace and (2) validating instructions executed by the speculative thread. Speculated traces are validated by verifying their live-output values. Live-output values are those that are produced and not overwritten within the trace. In the rest of the paper we will use the terms ST and NST to refer to the speculative thread and the non-speculative thread, respectively. Note that ST runs ahead of NST.

Both threads maintain their own architectural state by means of their associated architectural register file and a memory hierarchy with some special features. NST provides the correct and non-speculative architectural state, while ST works on a speculative architectural state. Note that each thread maintains its own state but that only the state of NST is guaranteed to be correct.

Additional hardware is required for each thread. ST stores its committed instructions to a special FIFO queue called *Look-Ahead Buffer*. NST executes the skipped instructions and verifies instructions in the look-ahead buffer executed by ST. Note that verifying instructions is faster than executing them because instructions always have their operands ready. In this way, NST catches ST up quickly.

ST speculates on traces with the support of a *Trace Speculation Engine* (TSE). This engine is responsible for building traces and predicting their live-output values. NST, on the other hand, uses special hardware called a *Verification Engine*. The NST executes the skipped instructions and verifies instructions in the look-ahead buffer executed by ST. This is done by verifying that source operands match the non-speculative state and by updating the state with the new result in case they match. If there is a mismatch between the speculative source operands and the non-speculative ones, a trace misspeculation is detected and a thread synchronization is fired. Basically, this recovery action involves flushing the ST pipeline and reverting to a safe point in the program. An advantage with this approach is that any live-output values used are the only ones that are verified. Note also that the verification of instructions is faster than their execution because instructions always have their operands ready. A critical feature of this microarchitecture is that this recovery is implemented with minor performance penalties.

2.3. Verification Engine

The verification engine (VE) is responsible for validating speculated instructions and, together with NST, maintains the speculative architectural state. Instructions to be validated are stored in the look-ahead buffer. Verification involves testing source values of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be updated in the non-speculative architectural state (register file or memory).

Memory operations require special considerations. First, the effective address is verified and, after this validation, store instructions update memory with the destination value. On the other hand, loads check whether the value of the destination register matches the non-speculative memory state. If it does, the destination value is committed to the register file.

This engine is independent of both threads but works cooperatively with NST to maintain the correct architectural state.

3. Thread Synchronization Analysis

Traces are identified by an initial and a final point in the dynamic instruction stream. They can be built according to different heuristics: basic blocks, loop bodies, etc [8], [9], [11]. Live-output values of a trace can be predicted in several ways, including with conventional value predictors such as last value, stride, context-based and hybrid schemes [13], [22].

Unfortunately, speculation accuracy decreases when the traces are large because they have a huge number of live-output values that have to be predicted. As outlined above, if source values of the instructions in the look ahead buffer do not match the non-speculative architectural state, a thread synchronization is required in the original TSMA architecture. This involves emptying all the ST structures and reverting to a safe point in

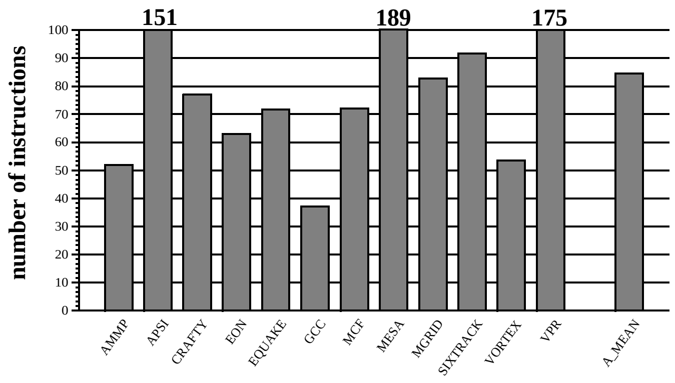

Figure 2. Number of squashed instructions in each thread synchronization.

the program. Note that, a misspeculation in one instruction causes younger instructions to be discarded from the look ahead buffer, though some may be correctly executed. Consider, for instance, a speculative trace in which just a single live-output value of the whole set is incorrectly predicted. Only the instructions dependent on the mispredicted one will be incorrectly executed by ST. In this section we analyse the number of correctly executed instructions that are squashed when a thread synchronization is fired. See Section 5.1 for details of the experimental framework.

Figure 2 shows the number of instructions that are squashed from the look ahead buffer every time a thread synchronization was fired. Note that the number of discarded instructions is significant for all benchmarks. On average, up to 80 instructions were squashed from the look ahead buffer in each thread synchronization irrespective of weather they were correctly or incorrectly executed.

Figure 3 shows on average the percentage of squashed instructions from the look ahead buffer that were correctly executed by ST was over 20%. Combined with the previous results, this means that on average 16 instructions that were correctly executed were discarded every time a thread synchronization was performed. This led us to reconsider thread synchronizations in order to try to avoid this waste of activity and reduce the number of fetched and executed instructions.

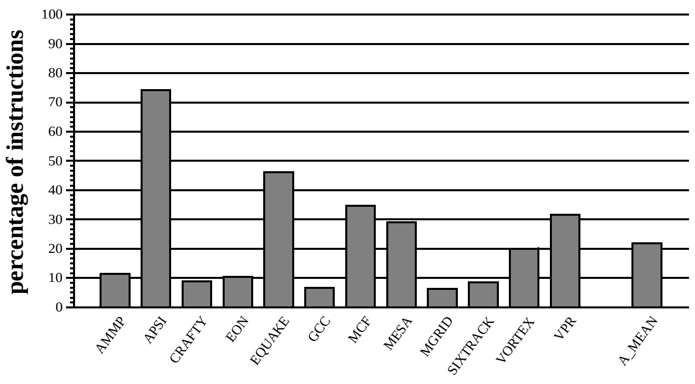

Figure 3. Percentage of the squashed instructions that were correctly executed.

4. Novel Verification Engine

The conventional verification engine is in charge of validating speculated instructions. Together with NST, it maintains the speculative architectural state. Instructions to be validated are stored in the look ahead buffer by ST. The verification consists of comparing source values of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be updated in the non-speculative architectural state (register file or memory). If they do not match, a thread synchronization is performed. Memory operations require special considerations. First, the effective address is verified. Then, store instructions update memory with the destination value. On the other hand, load instructions check whether the value of the destination register matches the non-speculative memory state. If it does, the destination value is committed to the register file. Note that this validation is fast and simple. Memory instructions stall verification if there is a data cache miss.

In this paper, we propose a novel verification engine that can significantly improve the performance potential of the architecture. The underlying concept is based on the idea that a misspeculation in one instruction does not necessarily causes valid work from sequential younger computations to be aborted. Thread synchronization can therefore be delayed or even avoided. Below we describe how this new verification engine behaves depending on the type of instruction that is validated:

- **Branch instructions:** These operations do not have an explicit destination value. Implicitly, they modify the program counter according to the branch direction that is taken or not taken. The idea is to validate the branch target instead of the source values. So, if source values are incorrectly predicted but the direction of the branch is correct, a thread synchronization is not fired.
- **Load instructions:** First, the effective address is verified. If validation fails, the correct effective address is computed. Therefore, load instructions do not check whether the value of the destination register matches the non-speculative memory state. Simply, the destination value obtained from memory is committed to the register file. Note that an additional functional unit is required in order to compute the effective address.
- **Store instructions:** As with load instructions, the effective address is first verified. If validation fails, store instructions update memory with the destination value obtained from the non-speculative architectural state, instead of the value obtained from the instruction. Note that only one functional unit is required to compute the effective address.
- **Arithmetic instructions:** As with the conventional engine, the verification of arithmetic operations involves comparing the source operands of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be committed to the register file. If they do not, the verification engine re-executes the instruction with values from the non-speculative state. In this case, verification is stalled and instructions after the re-executed one cannot be validated until the next cycle. Moreover, to maintain a high validation rate, this re-execution is only considered for single-cycle latency instructions. An additional functional unit is required in order to re-execute the instruction.

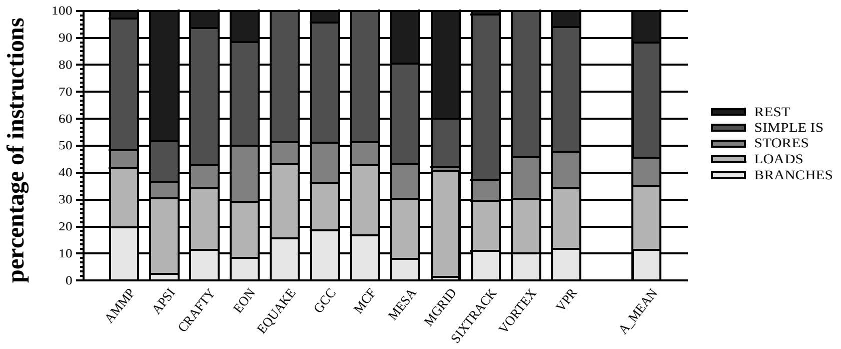

Figure 4. Type of incorrect speculated instructions

Note that only branch instructions with a wrong target and non-single-latency instructions with wrong source operands fire a synchronization.

Figure 4 shows the breakdown of instructions in the look-ahead buffer that fail validation for the original validation engine. From bottom to top the categories are: branch instructions, load instructions, store instructions, instructions with single-cycle execution latency and, finally, the rest of the instructions. Note that branches, memory operations and instructions with a single-cycle latency account for 90% of the total incorrectly executed instructions. This means that there is a huge potential benefit for the new verification engine. Also, simulation results show that on average just 1% of the instructions inserted in the look-ahead buffer are incorrectly predicted. This suggests that the new verification engine may not need to re-execute many instructions, so the validation rate will not be greatly affected. Therefore, for the novel TSMA we assume that the number of functional units is the same as for the conventional one. Finally, we assume that the maximum number of instructions validated per cycle is the same and that no more than one instruction is re-executed per cycle.

5. Performance Evaluation

In this section we describe the experimental framework assumed in this paper, analyse the performance of the novel engine and compare it with the conventional one.

5.1. Experimental Framework

The TSMA simulator is built on top of the Simplescalar Alpha toolkit [4]. Table 1 shows the parameters of the baseline conventional microarchitecture. The TSMA assumes the same resources as the baseline configuration except for the issue queue, reorder buffer and logical register mapping table, which are replicated for each thread. It also has some new structures, which are shown in Table 2.The following Spec2000 benchmarks were randomly chosen: *crafty, eon, gcc, mcf, vortex,* and *vpr* from the integer suite; and *ammp, apsi, equake, mesa, mgrid,* and *sixtrack* from the FP suite. The programs were compiled with the DEC C and F77 compilers with `-non_shared -O5` optimization flags (i.e. maximum optimization level). For the simulation, each program was run with the test input set and statistics were collected for 250 million instructions after skipping initializations.

TSMA assumes a trace selection method based on a static analysis that uses profiling data to determine traces to be speculated. These selected traces are communicated to the hardware at program loading time through a special hardware structure called trace table. Live-output values are predicted by means of a hybrid scheme comprising a stride value predictor and a context-based value predictor. See [15] for further details of the trace recognition approach based on profile guided heuristics.

Instruction fetch	4 instructions per cycle.
Branch predictor	2048-entry bimodal predictor
Instruction issue/ commit	Out-of-order issue, 4 instructions committed per cycle, 64-entry reorder buffer, loads execute only after all the preceding store addresses are known, store-load forwarding
Arch. Registers	32 integer and 32 FP
Functional units	4 integer ALUs, 4 load/store units, 4 FP adders, 2 integer mult/div, 2 FP mult/div
FU latency/repeat rate	int ALU 1/1, load/store 1/1, int mult 3/1, int div 20/19, FP adder 2/1, FP mult 4/1, FP div 12/12
Instruction cache	16 KB, direct-mapped, 32-byte block, 6-cycle miss latency
Data cache	16 KB, 2-way set-associative, 32-byte block, 6-cycle miss latency
Second Level Cache	Shared instruction & data cache, 256 KB, 4-way set-associative, 32-byte block, 100-cycle miss latency

Table 1. Parameters of the baseline microarchitecture.

Speculative data cache	1 KB, direct-mapped, 8-byte block
Verification engine	Up to 8 instructions verified per cycle. Memory instructions stall verification for L1 misses. Only one instruction may be re-executed per cycle.
Trace speculation engine	128 history table, 4-way set-associative,.
Look ahead buffer	512 entries

Table 2. Parameters of TSMA additional structures

5.2. Analysis of Results

The main objective of this section is to show that the number of thread synchronizations is lower when the new verification engine is used.

Figure 5 plots the percentage of thread synchronizations against the number of trace speculations. Figure 6 shows the speed-up of TSMA over the baseline architecture. The first bar in each figure represents TSMA with the conventional engine and the second bar represents TSMA with the new verification engine. Our results show that the average speed-up for TSMA was 27 with the conventional verification engine. As

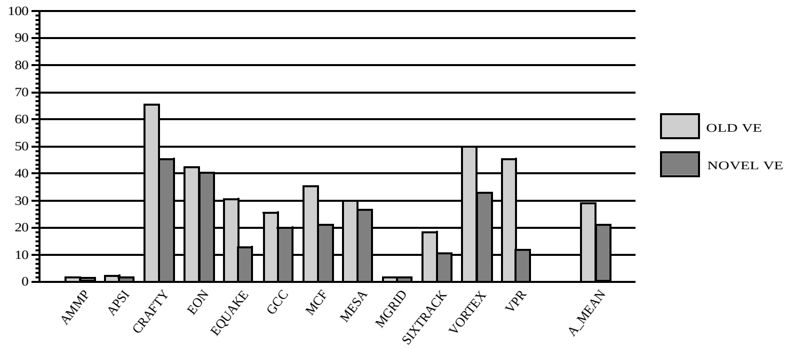

Figure 5. Percentage of thread synchronization

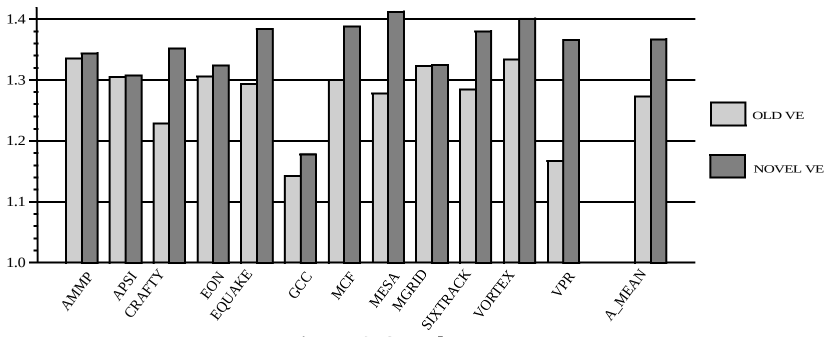

Figure 6. Speed-up

expected, these speed-ups were significant for all the benchmarks despite a thread synchronization rate close to 30%.

On the other hand, the number of thread synchronizations was about 10% lower (from 30% to 20%) with the new verification engine than with the conventional scheme. Note that this engine did not always fires a thread synchronization to handle a miss trace speculation. It also provided a higher speed up (close to 38%), which implies that the average performance improvement was 9%. Note that the performance of most benchmarks improved significantly. Only benchmarks such as *ammp*, *apsi* or *mgrid*, whose misspeculation with the traditional verification engine was negligible, hardly improved since thread synchronizations were already low with the original verification engine.

These results demonstrates the tolerance to misspeculations of the proposed microarchitecture and encourage further work to develop more aggressive trace prediction mechanisms. Note that the novel verification engine opens up a new area of investigation i.e. aggressive trace predictor mechanisms that do not need to accurately predict all live output values.

Figure 7 shows the reduction in executed instructions with the new verification engine. On average, this reduction is almost 8%. Note that this also reduces memory pressure since these instructions do not need to be fetched all together. Again,

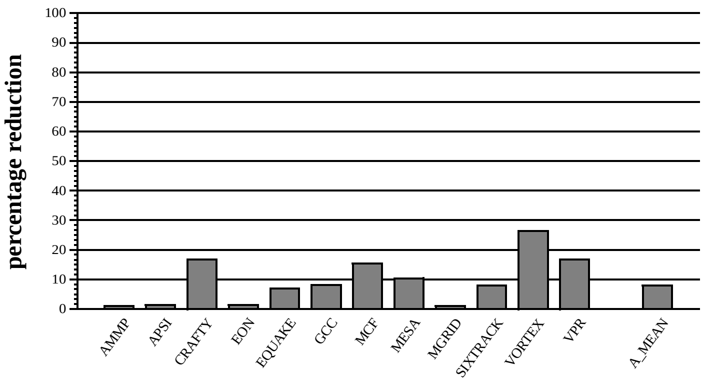

Figure 7. Reduction in executed instructions

benchmarks whose percentage of synchronization was negligible experienced a very small reduction in executed instructions. For the other benchmarks, on the other hand, the number of executed instructions and the number of thread synchronizations decreased, which led to significant speed-ups.

6. Related Work

Several techniques for reducing recovery penalties caused by speculative execution have been proposed. Instruction reissue, or selective squashing, was first proposed in [12]. The idea is to retain instructions dependent on a predicted instruction in the issue queue until the prediction is validated. If the prediction is wrong, all dependent instructions are issued again. This technique trades the cost of squashing and re-fetching instructions for the cost of keeping instructions longer in the issue queue. A similar scheme that focused on load instructions was presented in [10]. The performance of the instruction reissue was investigated thoroughly in [25]. A practical and simple implementation of instruction reissue based on a very slight modification of the register update unit was proposed in [21].

Squash reuse has also been proposed as a way to reduce branch miss-speculation penalty. This concept was first introduced in [23]. These authors proposed a table-based technique for avoiding the execution of an instruction that has previously been executed with the same inputs. As well as squash reuse, they also cover general reuse. A different implementation based on a centralized window environment was proposed in [6]. These authors also introduced the idea of dynamic control independence and showed how it can be detected and used in an out-of-order superscalar processor to reduce the branch misprediction penalty. Finally, register integration [19] has also been proposed as a simple and efficient implementation of squash reuse. This mechanism allows speculative results to remain in the physical register file after the producer instruction is squashed. They can later be reused through a modified renaming scheme.

The concept of dynamic verification was introduced in [17]. The proposed AR_SMT processor employs a time redundant technique that allows some transient errors to be tolerated. Slipstream processors [16] dynamically avoid the execution of a program’s non essential computations. These authors suggested creating a shorter version of the original program by removing ineffectual computation. Using dynamic verification to reduce the burden of verification in complex microprocessor designs is covered in [5].

Several thread-level speculation techniques have been examined to exploit parallelism in general-purpose programs [1][2][13][24]. Other recent studies have also focused on speculative threads. The pre-execution of critical instructions by means of speculative threads is proposed in several studies [3][7][20]. Critical instructions, such as mispredicted branches or loads that miss in cache are used to construct traces called slices that contain the subset of the program that relates to that instruction.

7. Conclusions and Future Work

In this paper, we have proposed a novel hardware technique to enhance the *Trace Level Speculative Multithreaded Architecture* (*TSMA*). This hardware improvement focuses on the verification engine of the TSMA. The idea is to avoid the re-execution of instructions even when source values are incorrectly predicted. Instead of firing a thread synchronization that wastes useful computations, the correct value is re-computed and used to update the architectural state. The new engine reduces the number of thread synchronizations and the penalty due to misspeculations. This avoids discarding instructions that are independent of a mispredicted one, thus reducing the number of fetched and executed instructions and cutting energy consumption and contention for execution resources.

Simulation results of the novel verification engine of the *Trace-Level Speculative Multithreaded Architecture* show that it can significantly improve performance without increasing complexity. These results encourage further work to develop more aggressive speculation schemes based on the idea that not all live-output values need to be highly predictable. This is also motivated by the relatively low penalty of misspeculations achieved by the *Trace-Level Speculative Multithreaded Architecture*.

Finally, TSMA processor can simultaneously execute a couple of threads that cooperate to execute a sequential code. To ensure the correctness of the architectural state, ST may only speculate a new trace when the look-ahead buffer is empty. This means that TSMA has only a single unverified trace speculation at any given time. Future work includes to modify the architecture in order to allow multiple unverified traces while maintaining the relatively low penalty of misspeculations. Future areas for investigation also include generalising the architecture to multiple threads in order to perform sub-trace speculation during the validation of a trace that has been speculated.

8. Acknowledgments

This work has been partially supported by the Ministry of Education and Science under grants TIN2004-07739-C02-01 and TIN2004-03072, the CICYT project TIC2001-0995-C02-01, Feder funds, and Intel Corporation. The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona and the resources of the Robotics and Vision Group of Tarragona.

9. References

[1] P. S. Ahuja, K. Skadron, M. Martonosi and D. W. Clark. “Multipath Execution: Opportunities and Limits”. *In Proceedings of the International Symposium on Supercomputing*, 1998.

[2] H. Akkary and M. Driscoll. “A Dynamic Multithreaded Processor”. I*n Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[3] R. Balasubramonian, S. Dwarkadas and D. Albonesi. “Dynamically Allocating Processor Resources between Nearby and Distant ILP”. *In Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[4] D. Burger, T.M. Austin and S. Bennet, “Evaluating Future Microprocessors: The SimpleScalar Tool Set”. Technical Report CS-TR-96-1308. University of Wisconsin, 1996.

[5] S. Chaterjee, C. Weaver and T. Austin. “Efficient Checker Processor Design”. *In Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000.

[6] Y.Chou, J. Fung, J.Shen, “Reducing Branch Missprediction Penalties Via Dynamic Control Independence Detection”, *In Proceedings of International Conference on Supercomputing*, 1999.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery and J. Shen. “Speculative Precomputation: Long-range Prefetching of Delinquent Loads”. *In Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[8] D. A. Connors and W. W. Hwu. "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results". I*n Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999.

[9] A. González, J. Tubella and C. Molina, "Trace Level Reuse". *In Proceedings of the International Conference on Parallel Processing*, 1999.

[10] J. González and A. González, "Speculative Execution via Address Prediction and Data Prefetching", *In Proceedings of the 11th International Conference on Supercomputing, 1997.*

[11] J. Huang and D. Lilja. "Exploiting Basic Block Value Locality with Block Reuse". *In Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[12] M. H. Lipasti, "Value Locality and Speculative Execution", Ph.D. Dissertation, department of Electrical and Computer Engineering, Carnegie Mellon University, April 1997.

[13] P. Marcuello, J. Tubella and A. González. "Value Prediction for Speculative Multithreaded Architectures". *In Proceedings of the 32th Annual International Symposium on Microarchitecture*, 1999.

[14] C. Molina, J. Tubella and A. González. "Trace-Level Speculative Multithreaded Architecture". *In Proceedings of the International Conference on Computer Design*, 2002.

[15] C. Molina, J. Tubella and A. González. "Compiler Analysis to Support Trace-Level Speculative Multithreaded Architectures", *In Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, 2005.

[16] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. "A Study of Slipstream Processors". *In Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[17] E. Rotenberg. "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors". *In Proceedings of the 29th Fault-Tolerant Computing Symposium*, 1999.

[18] E. Rotenberg. "Exploiting Large Ineffectual Instruction Sequences". Technical Report, North Carolina State University, November 1999.

[19] A. Roth and G. S. Sohi, "Register Integration: A Simple and Efficient Implementation of Squash Reuse", *In Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[20] A. Roth and G. Sohi. "Speculative Data-Driven Multithreading". *In Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[21] T. Sato and I. Arita, "Comprehensive Evaluation of an Instruction Reissue Mechanism", *In Proceedings of the 5th International Symposium on Parallel Architectures, Algorithms and Networks*, 2000.

[22] Y. Sazeides and J. E. Smith, "The Predictability of Data Values". *In Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[23] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", *In Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[24] D. M. Tullsen, S. J. Eggers and H. M. Levy. "Simultaneous Multithreading: Maximizing on-chip Parallelism". *In Proceedings of the 22th Annual International Symposium on Computer Architecture,* 1995.

[25] G. S. Tyson and T. M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming". *In Proceedings of the 30th Annual Symposium on Microarchitecture,* 1997.