Trace-Level Speculative Multithreaded Architecture

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

[Ψ] Dept. d'Enginyeria Informàtica i Matemàtiques
Universitat Rovira i Virgili
Paisos Catalans s/n, 43007 Tarragona, Spain
e-mail: cmolina@etse.urv.es

[ϕ] Department d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, 08034 Barcelona, Spain
e-mail: {antonio,jordit}@ac.upc.es

Abstract

This paper presents a novel microarchitecture to exploit trace-level speculation by means of two threads working cooperatively in a speculative and non-speculative way respectively. The architecture presents two main benefits: (a) no significant penalties are introduced in presence of a misspeculation and (b) any type of trace predictor can work together with this proposal. In this way, aggressive trace predictors can be incorporated since misspeculations do not introduce significant penalties.

We describe in detail TSMA (Trace-Level Speculative Multithreaded Architecture) and present some initial results to show the benefits of this proposal. We show how simple trace predictors achieve significant speed-up in the majority of the cases. Concretely, results of a simple trace speculation mechanism results in an average speed-up of 16%.

1. Introduction

Data dependences are one of the most important hurdles that limit the performance of current microprocessors. Two techniques have been proposed so far to avoid the serialization caused by data dependences: data value speculation and data value reuse. Speculation tries to predict a given value as a function of the past history. Value reuse is possible when a given computation has been done exactly in the past. Both techniques may be considered at two levels: instruction level and 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, these instructions do not need to be fetched, and thus, 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 introduces a new problem: penalties due to a misspeculation.

Two important issues have to be considered regarding trace level speculation: (1) control and data speculation techniques and (2) the microarchitecture support for trace speculation. These two issues are completely orthogonal. Trace predictors are in charge of control and data speculation. Control speculation is related to the prediction of the initial and final points of a trace. Data speculation is related to the prediction of live-output values of a trace. Traces can be built following different heuristics: basic blocks, loop bodies, etc. On the other hand, once a trace is built, the form that live outputs are predicted may vary as well. Conventional value predictors include: last value, stride, context-based, hybrid, etc. This work focuses on the second issue: the way microarchitecture manages trace speculation. We present a microarchitecture that is tolerant to misspeculations. This architecture does not introduce significant trace misspredictions penalties and does not impose any constraint on the approach to building or predicting traces.

Trace level speculation may be managed in different ways. Meanwhile trace level reuse may be static or dynamic and it is not speculative, trace level speculation is dynamic (although the compiler may help) and requires a live-input or live-output test. Trace level speculation with live-output test is supported by means of a multithreaded architecture. The underlying concept is to have a couple of threads working cooperatively: a speculative thread and a non-speculative one. In this case the correctness of speculated traces is assured by means of the verification of live-output values. 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. 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 verifying that source operands match the non-speculative state and updating the state with the new produced result. Note that each thread maintains its own state, but only the state of the non-speculative thread is guaranteed to be correct. The advantage in this approach is that only live outputs that are used are verified. Figure 1 depicts this approach.

The microarchitecture presented in this work is focused on the latter approach: trace level speculation with live-output test. The rest of this paper is organized as follows. Section 2 describes in detail the microarchitecture proposed to exploit trace level speculation. The performance potential and results of simulations are analyzed in Section 3. Section 4 reviews some related work. Finally, Section 5 summarizes the main conclusions of this paper and the future.

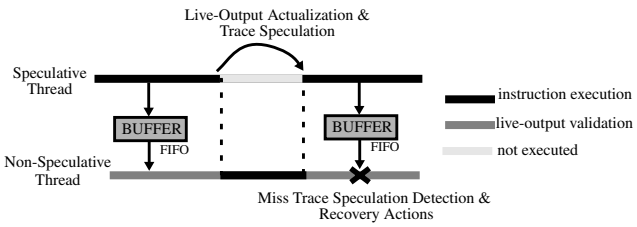

Figure 1. Trace level speculation with live-output test

Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)
1063-6404/02 $17.00 © 2002 **IEEE**

2. Trace-Level Speculative Multithreaded Microarchitecture

This section outlines the main characteristics of the Trace-Level Speculative Multithreaded Microarchitecture. First, an overview is presented. Following subsections describe the main components of the proposed microarchitecture in more detail. The last subsection presents a working example.

2.1. Overview

The underlying concept of our proposal is based on a couple of threads working cooperatively as shown in Figure 1. One thread, called speculative thread, executes instructions and speculates on the result of whole traces. The second thread executes speculated traces and verifies instructions that are executed by the speculative thread. This second thread is called non-speculative thread. In the rest of the paper we use the terms ST to refer to the speculative thread and NST to refer to the non-speculative one. Note that ST runs ahead of NST.

Each thread maintains its own architectural state by means of their associated architectural register file and a memory hierarchy with some special features (described below). NST provides the correct and non-speculative architectural state. Meanwhile, ST works on a speculative architectural state. Additionally, ST stores their committed instructions to a special first-input first-output 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 since instructions always have their operands ready. In this way, NST catches ST up quickly.

Additional hardware is required for each thread. ST speculates traces with the support of a *Trace Speculation Engine*. This engine is responsible for building traces and predicting their live-output values. The study of alternative designs of the speculation engine is beyond the scope of this paper. On the other hand, NST is supported by means of a special hardware called *Verification Engine.*

Instruction verification is straightforward. NST takes instructions from the look ahead buffer and validates values of source operands. If source operand values match the non-speculative architectural state values, it determines that the instruction was correctly executed by ST. In this way, the non-speculative architectural state is updated with the destination value. On the other hand, if there is not a match between the values, a recovery action is initiated. A critical part of the microarchitecture is to implement this recovery with minor performance penalties. This is one of the main foci of this paper.

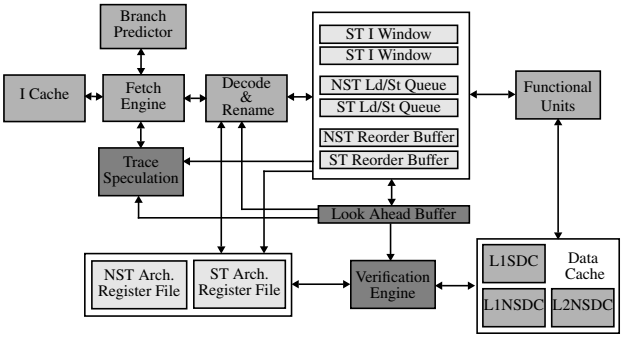

Figure 2. Trace level speculative multithreaded microarchitecture

Figure 2 shows the proposed microarchitecture with the additional hardware requirements highlighted over a baseline superscalar architecture. The hardware can be divided in three categories:

- *Local*: each thread maintains a logical register file, an instruction window, a load store queue and a reorder buffer. All this hardware is replicated for both threads. (light gray color in Figure 2)
- *Shared*: not replicated hardware shared by both threads. These resources are an instruction cache, a fetch engine, a branch predictor, a decode and rename logic, functional units, a modified data value cache and logical control. (medium gray color in Figure 2)
- *Additional*: hardware requirements to provide trace level speculation. This resources are the look ahead buffer, the verification engine and the trace speculation engine. (dark grey color in Figure 2).

The main parts of the *Trace Level Speculative Multithreaded Microarchitecture* are described below in more detail.

2.2. Trace Speculation Engine

The trace speculation engine (TSE) is in charge of two issues: (1) to implement a trace level predictor and (2) to communicate a trace speculation opportunity to the fetch engine. We assume in this work that the trace predictor maintains a simple PC-indexed table with N entries. Each entry contains live-output values and the final program counter of the trace. TSE receives from NST and the verification engine the information required to build traces and to determine live-output values. This information comes from correctly executed instructions at commit time.

To determine trace speculation opportunities, TSE scans the current program counter of ST. This value is provided by the fetch engine. If TSE determines that the current PC is the beginning of a potentially predictable trace, it provides some trace information to the fetch engine. This information consists of a special INI_TRACE instruction and some MOV instructions. The INI_TRACE instruction contains the final program counter of the trace and the number of times that this PC is repeated inside the trace (this allows the TSE to construct traces that consists of multiple loop iterations). Additional MOV instructions inserted into the ST pipeline update live-output values of a trace. Afterwards, ST continues with normal instruction fetch from the final point of the speculated trace. Next cycle, NST wakes-up and begins to fetch and execute instructions of the speculated trace.

2.3. Look Ahead Buffer

The objective of this structure is to store instructions executed by ST. Later on, NST will validate them. The look ahead buffer is just a first-input first-output queue. Thus, a huge look ahead buffer can be managed easily. ST introduces instructions at commit time whereas the verification engine takes these instructions and test their correctness. The fields of each entry of the look ahead buffer are the following:

- Program counter
- Operation type: indicates memory operation
- Source register ID 1 & Source value 1
- Source register ID 2 & Source value 2
- Destination register ID & Destination value
- Memory address

2.4. Verification Engine

The verification engine (VE) is in charge of validating speculated instructions and, together with NST, it maintains the speculative architectural state. Instructions to be validated are stored in the look ahead buffer. The verification consists of testing source values of the instruction with the non-speculative architectural state. If they match, 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. 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 so, 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.The verification engine is replicated for each thread because any of them can take the role of speculative or non-speculative thread. Anyway, the hardware required to perform the verification is minimal.

2.5. Thread Synchronization

Thread synchronization is required when a trace misspeculation is detected by the verification engine. Basically, this implies to flush the ST pipeline and go back to a safe point of the program. The recovery actions involved by a synchronization are simple:

- Instruction execution is stopped.
- ST structures are emptied (instruction window, load store queue, reorder buffer and look ahead buffer).
- Speculative data cache and logical register file associated to ST are invalidated.

NST executes traces speculated by ST and the verification engine validates ST executed instructions once ST puts them in the look ahead buffer. NST keeps on execution beyond the final point of the speculated trace but commit of these instructions is disabled. This is done to significantly reduce the penalties caused by synchronization. In this way, two types of synchronizations may occur: total and partial.

Total synchronization occurs when a misspeculation is detected by the verification engine and NST it is not executing instructions after the end of the trace. This implies squashing ST and paying the penalty of starting to fill its pipeline from the point it detected the misspeculation. On the other hand, partial synchronization occurs when a misspeculation is detected and NST is already executing instructions. In this way, ST pipeline does not need to be refilled. NST takes the role of ST enabling the commit of the already executed instructions after the end of the speculated trace. Meanwhile recovery actions are taken to initialize ST with a correct architectural state at the failure point. After this synchronization, roles of threads are interchanged. This partial synchronization avoids the pipeline refill penalty at the expense of the constraint that whereas NST is executing instructions beyond the end of a speculated trace, ST cannot speculate on a new trace. So, this number of additional executed instructions should not be very large. On the other hand, it is important to minimize the number of total synchronizations without losing speculation opportunities. Empirically we have observed that trace misspeculations are detected relatively early. The processor dynamically determines the number of instructions to be executed after an speculated trace, based on the number of verified instructions before a misspeculation is detected.

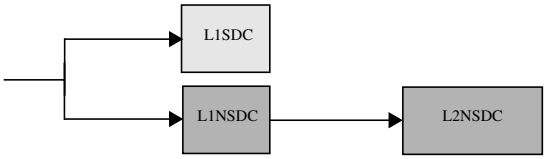

Figure 3. Memory subsystem

2.6. Memory Subsystem

A new first level data cache architecture is proposed (see Figure 3). This cache architecture is responsible for maintaining the speculative memory state of ST. The first level of the memory hierarchy is composed of two modules: the level 1 speculative data cache (L1SDC) and the level 1 non-speculative data cache (L1NSDC). The second level just contains non-speculative data and will be referred to as level 2 non-speculative data cache (L2NSDC).

This new organization is guided by the following rules:

1. ST store instructions update values in L1SDC only.
2. ST load instructions get values from L1SDC. If a value is not in L1SDC, it is obtained from L1NSDC or L2NSDC in a traditional way. The accesses to L1SDC and L1NSDC are done in parallel. No line from L1SDC is copied back to L2NSDC.
3. NST store instructions (executed by NST or verified by the verification engine) update values and allocate space just in the non-speculative caches.
4. NST load instructions (executed by NST or verified by the verification engine) get values and allocate space just in the non-speculative caches.
5. A line replaced in L1NSDC is copied back to L2NSDC.

Note that rules 3 to 5 correspond to the normal management of traditional caches, while rules 1 and 2 describe the behavior of the new speculative cache. Simulations show that a very small L1SDC may suffice to provide good performance.

The following figure presents different scenarios corresponding to different orderings of the actions involved in the speculation, execution and verification of a trace. Note that this example considers a correct trace speculation. Figure 4 shows ST performing a trace speculation which includes a store instruction (1). After speculation, ST executes a load which refers to the memory location of the speculated store (2). On the other hand NST executes the speculated store (3) and later on, the load instruction is verified (4). A complete set of working examples of the memory system under different scenarios (incorrect and correct speculated traces) is presented in [12].

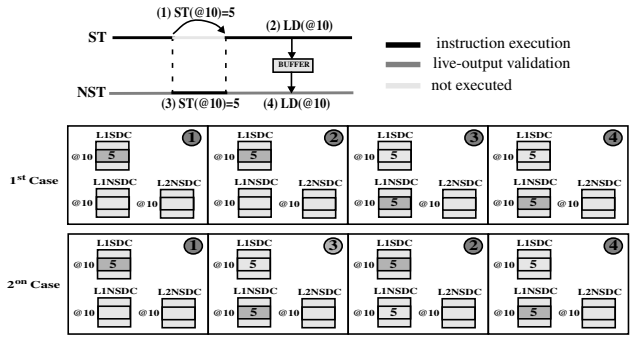

Figure 4. Example: correct trace speculation

2.7. Register File

The proposed microarchitecture assumes a register renaming mechanism in which speculative register values are kept in the reorder buffer. The register map table contains for each entry the following fields:

- Committed Value: it contains the last committed value of the register.
- ROB Tag: it points to the ROB entry that has (or will have) the latest value of the register.
- Counter: it determines the number of instructions in ST that are using this register after a trace speculation.

Note that the difference with respect to traditional structures is the new counter field. This field is used to provide NST the ability of beginning the execution of speculated traces as soon as possible. In particular, once ST speculates a trace, NST begins the execution of that trace immediately. Note that there may be instructions from ST before the speculation point that still have not been executed. It is possible that these instructions produce values to be consumed by NST instructions. In this way, these dependent NST instructions have to wait for ST completion. For this reason, NST needs to know whether its instructions have source operands that are still not ready because ST has not finished their execution.

The counter field is maintained as follows:

1. When a ST instruction enters the instruction window, the counter associated to its destination register is increased.
2. When an instruction is committed to the look ahead buffer by ST, the destination register counter is decreased.
3. After a trace speculation, the counter is no longer increased. It is just decreased until it reaches the value zero. If an instruction decoded by the NST encounters a source operand with a counter field equal to zero, this indicates that the instruction is younger than the speculation point.

ST passes a copy of the counters to NST when the special INI_TRACE instruction that determines trace speculation is renamed. The other fields of the register map table, i.e., the committed values and ROB tags do not need to be communicated. Then, the verification engine decreases the counters as it validates instructions. This is done until a special mark in the look ahead buffer that determines the start of a trace speculation is reached. In this way, a counter greater than zero indicates that the register value is not ready since it has not been verified by the verification engine. This does not prevent NST from executing instructions that do not depend on this value. On the other hand, to guarantee correctness of memory state, memory instructions are stalled until the verification engine reaches the starting point of the trace speculation.

Note that there may be speculated traces in a path that is incorrectly speculated by a branch. In this case, NST begins execution but when ST determines an incorrect path and recovers, it stops and empties its thread private structures

2.8. Working Example

In order to understand the behavior of the microarchitecture, a detailed working example is presented. Figure 5 shows the key steps of a trace speculation. Below, a detailed explanation of each step is presented.

1. ST begins the execution of the program and commits the instructions to the look ahead buffer.
2. The trace speculation engine identifies a trace speculation opportunity, notifies the fetch engine, and provides ST with the information required through a special INI_TRACE instruction. At this point the program counter is modified and additional instructions to update live-output values are provided. When the INI_TRACE instruction is renamed, NST receives a copy of the ST mapping table. Now ST maintains a speculative architectural state using its mapping table and the memory hierarchy in a speculative way.
3. NST begins to execute the ST skipped instructions immediately. This prompt execution is done through the support of the special mapping table as described above.
4. VE consumes instructions from the look ahead buffer and updates the architectural register file shared with NST. It decreases the register map counters, and stores the committed value in the mapping table. This is done until an INI_TRACE instruction is reached, and it guarantees that NST does not execute instructions with values that are still not produced by ST.
5. NST executes instructions normally. It commits instructions and maintains the correct state.
6. NST detects the final point of the speculated trace. The verification engine begins to validate instructions and to update the architectural state. NST continues executing instructions but commit is disabled. Memory instructions are stalled.
7. The verification engine validates ST executed instructions after the trace speculation. The verification engine guarantees the correct state and verifies N instructions from the look ahead buffer. This number is set dynamically to be a bit larger than the average number of verified instructions that precede a misspeculation detection. If verification fails, recovery actions are taken. If NST is still executing instructions, it takes the role of ST. This is known as partial synchronization. At this point the state is safe so the verification engine becomes idle.
8. If there is no a misspeculation among the N verified instructions the NST structures are flushed and it becomes idle. The verification engine continues verifying instructions and maintaining the correct architectural state. If the verification engine finds a misspeculation when NST is idle, a total synchronization occurs. ST is squashed and refilled starting from the incorrect instruction.
9. ST may speculate on a new trace when the look ahead buffer is empty. This ensures the correctness of the architectural state. In this way, it is guaranteed that NST receives a correct copy of the mapping table.
10. NST executes the trace as described in point 2.

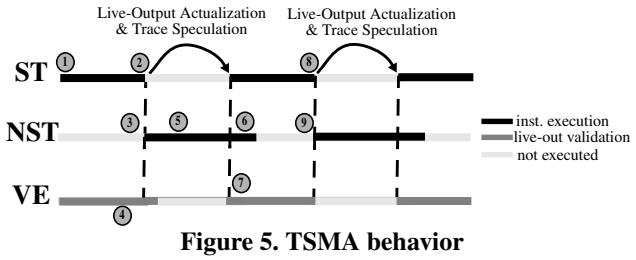

Figure 5. TSMA behavior

3. Performance Evaluation

3.1. Experimental Framework

The simulation environment is built on top of the Simplescalar [4] Alpha toolkit. Simplescalar models an out of order machine and it has been modified to support trace level speculative multithreading. Table 1 shows the parameters of the baseline microarchitecture. The *Trace Level Speculative Multithreaded Architecture* does not modify sizes of baseline structures. It just replicates for each thread unit the instruction window, reorder buffer and logical register mapping table. It also adds some new structures as shown in Table 2.

Instruction fetch	4 instructions per cycle.
Branch predictor	2048-entry bimodal predictor
Instruction issue/ commit	ooo issue, 4 insts committed/cycle, 64-entry reorder buffer, loads execute after all preceding stores are known, st-ld forwarding
Arch. registers	32 integer and 32 FP
Functional units	4 int ALUs, 4 ld/st units, 4 FP ALUs, 2 int 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 insts verified per cycle. Memory instructions block verification if fail in L1.Number of additional insts verified after average number to find an error is 8
Trace speculation engine	History Table: 64 entries, 2-way set, 9 instances/entry
Look ahead buffer	128 entries

Table 2. Parameters of TSMA additional structures

The following Spec95 benchmarks have been considered: *compress, gcc, go, li, ijpeg, m88ksim, perl and vortex* from the integer suite; and *applu, mgrid and turb3d* from the FP suite. The programs have been compiled with the DEC C and F77 compilers with `-non_shared -O5` optimization flags (i.e, maximum optimization level). Each program was run with the test input set and statistics were collected for 125 million of instructions after skipping an initial part of 250 million of instructions.

3.2. Analysis of Results

A main objective of this section is to show that trace misspeculations cause minor penalties in the microarchitecture. We propose a simple mechanism for building traces and determining live outputs. Traces are built following a simple rule: a trace starts at a backward branch and terminates at the next backward branch. Traces are also terminated at calls and returns, and have a minimum and maximum size (8 and 64 instructions respectively). On the other hand, live-output values are predicted by means of a hybrid scheme composed of a stride predictor and a context-based predictor. This mechanism maintains in each entry of the history table and in an ordered way, the last 9 dynamic instances of a trace. At prediction time, if the last instance of the trace appears among the previous 8 instances, next trace is predicted. In other case, stride prediction is performed. Figure 6 presents the percentage of misspeculations of the mechanism described above. As shown in Table 2 capacity of prediction tables is relatively small. Note that this mechanism produces a huge percentage of misspeculations, which is close to 70% on average.

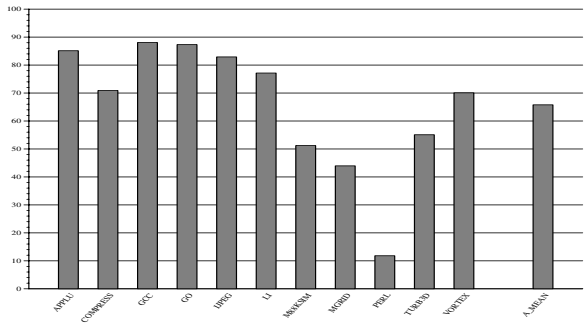

Figure 6. Percentage of misspeculations

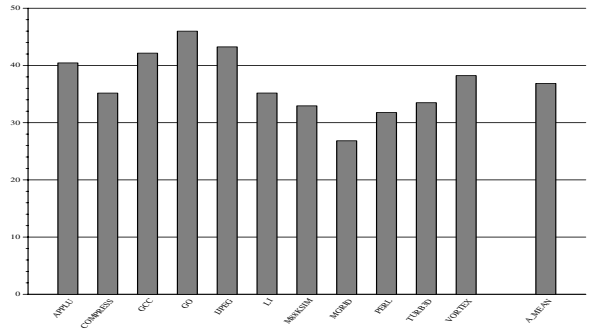

Figure 7. Percentage of predicted instructions

Figure 7 shows the percentage of speculated instructions. It represents on average close to 40%, so speculation is relatively frequent. Note that the ideal scenario is when the percentage of speculated instructions is around 50% since the microarchitecture has two threads.

Figure 8 presents the speed-up obtained over the baseline model. Notice that no slow-down is presented in any of the analyzed benchmarks although is huge the percentage of misspeculations. In fact, significant speed-ups are obtained for most of them. Results show that an average speed-up of 16% is obtained in spite of speculating a small percentage of instructions correctly and, misspeculating on average close to 70% of the traces. These results demonstrate the tolerance of the proposed microarchitecture to misspeculations. Furthermore, it encourage further work on developing more accurate trace prediction mechanisms. Some previous works [8], [9], [10], [14] have shown that there is a significant potential for trace repeatability/predictability, which suggests that there may be effective schemes to significantly increase the accuracy of trace predictors. Additional results and further details of the microarchitecture are provided in [12].

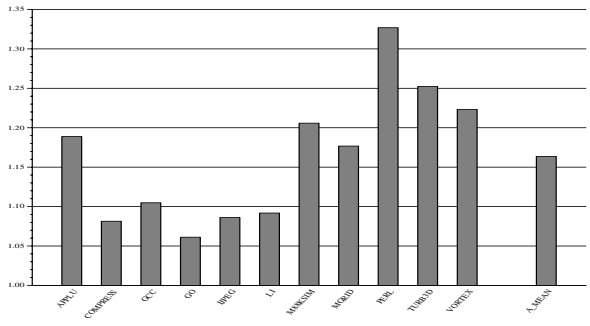

Figure 8. Speedup

4. Related Work

The performance potential of data value prediction and data value reuse, has been deeply investigated [16]. Trace level reuse [8], [9], [10] has been shown to be able to boost performance in superscalar processors. Unfortunately, checking the correctness of a trace reuse introduces a significant complexity that limits performance and decreases reuse opportunities.

The idea of dynamic verification was introduced in [14]. The AR_SMT processor proposed employs a time redundant technique that it permits to tolerate some transient errors. Slipstream processors [13] dynamically avoid the execution of non essential computations of a program. They propose to create a shorter version of the original program removing ineffectual computation. The use of dynamic verification to reduce the burden of verification in complex microprocessor designs is covered in [6].

Speculative multithreading [2], [11] is a well known technique based on the concurrent execution of speculative threads. Several works have studied the impact of different value predictors to alleviate dependence constraints and enable look ahead execution of speculative threads. Simultaneous Multithreading [17] allows independent threads to issue instructions to multiple functional units in a single cycle. Multiple Path Execution [1], [18] permits the speculative execution of multiples paths in parallel. Simultaneous Subordinate Microthreading [5] was proposed to execute subordinate threads that perform optimizations on the single primary thread.

Recent studies have focused on speculative threads. Pre-execution of critical instructions as standalone speculative threads is proposed in [7], [15], [19]. Critical instructions, such as misspredicted 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. A novel microarchitecture that dynamically allocates processor resources between a primary and a future thread was proposed in [3]. The future thread executes instructions when the primary thread is limited by resource availability, warming up certain microarchitectural structures.

5. Conclusions and Future Work

This paper has presented TSMA (*Trace-Level Speculative Multithreaded Architecture*). This novel microarchitecture is designed to exploit trace-level speculation with special emphasis on minimizing misspeculation penalties. Simulations presented in this paper based on a simple mechanism to build traces and to predict its live outputs show that the microarchitecture is very tolerant to trace misspeculations. In fact, significant speed-up is presented in the majority of the analyzed benchmarks in spite of the relatively poor accuracy of the assumed trace predictor. On average, a speed-up of 16% is achieved with a trace predictor that misses in 70% of the cases.

Due to the misspeculation tolerance aggressive trace level predictors can be incorporated to the processor. This opens an interesting area of research on the design of better trace level predictors. Another area for future investigation is the generalization of the architecture to multiple threads. With multiple threads, the execution of a speculated trace may be done in a cascade way. The idea is to perform sub-trace speculation inside a given trace.

6. Acknowledgments

This work has been supported by projects CYCIT 511/98 and ESPRIT 24942. The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

7. 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,* July 1996

[5] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. "Simultaneous Subordinate Microthreading (SSMT)". *In Proceedings of the 26th International Symposium on Computer Architecture,* 1999.

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

[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. M. Hwu. “Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results”. *In Proceedings of the 32th 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. 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.

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

[12] C.Molina, A. González and J. Tubella. “Trace-Level Speculative Multithreaded Architecture”. *Technical Report UPC-DAC-2001-35*, 2001.

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

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

[15] A. Roth and G. Sohi. “Speculative Data-Driven Multithreading”. I*n Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[16] Y. Sazeides and J. Smith. “The Predictability of Data Values”. *In Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 1997.

[17] 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.

[18] S. Wallace, B. Calder and D. Tullsen. “Threaded Multiple Path Execution”. *In Proceedings of the 25th Annual International Symposium on Computer Architecture,* 1998.

[19] C. Zilles and G. Sohi. “Execution-based Prediction Using Speculative Slices”. *In Proceedings of the 28th International Symposium on Computer Architecture,* 2001.