Compiler Analysis for Trace-Level Speculative Multithreaded Architectures

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

[ψ] Dept. Eng. Infomàtica 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-Univ. Politècnica de Catalunya
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. In this paper, we propose a static program analysis for identifying candidate traces to be speculated. This approach identifies large regions of code whose live-output values may be successfully predicted. We present several heuristics to determine the best opportunities for dynamic speculation, based on compiler analysis and program profiling information. Simulation results show that the proposed trace recognition techniques achieve on average a speed-up close to 38% for a collection of SPEC2000 benchmarks.

1. Introduction

Trace-level speculation avoids having to execute 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 regarding trace-level speculation. The first of these involves the microarchitecture support for trace speculation and concerns how the microarchitecture manages trace speculation. The second involves trace selection and data value speculation techniques.

Traces are identified by an initial and a final point in the dynamic instruction stream, and data speculation refers to the prediction of a trace´s live-output values. Traces can be built according to various heuristics such as basic blocks and, loop bodies, etc [7], [11], [12]. Once a trace is built, live-output values can be predicted in several ways, including using conventional value predictors such as last value, stride, context-based and hybrid schemes [15], [24].

In this paper we assume a microarchitecture called Trace-Level Speculative Multithreaded Architecture (TSMA) [16],[17]. This microarchitecture is tolerant to misspeculations in the sense that it does not introduce significant trace missprediction penalties and does not impose any constraint on the approach to building or predicting traces. This work focuses on the microarchitecture support for trace speculation and therefore assumes a simple mechanism for building traces and determining live outputs. We extend this initial work by proposing a trace selection method based on a static analysis that uses profiling data. In this way, we focus on developing effective trace selection schemes for TSMA processors.

The rest of this paper is organized as follows. The trace selection approach is presented in Section 2. An overview of the TSMA microarchitecture is presented in Section 3. The performance of the processor with the proposed trace selection scheme is analyzed in Section 4. Related work is analysed in Section 5. The main conclusions of this paper and outlines for future work are discussed in Section 6.

2. Trace Selection

Program profiling analysis is an effective technique for determining code regions whose live-output values may be reused at run-time [7],[13]. In this paper we propose a profile-guided analysis for selecting the traces to be speculated by a TSMA processor. This analysis is detailed in the following subsections.

2.1. Graph Construction

Trace selection is performed using an abstract data structure that is built from information obtained from the control flow graph, the data dependence graph and the predictability of values. The abstract data structure is a graph in which each node provides useful information for an static instruction. This information is obtained by running the test input set of the analyzed benchmarks. The

Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)
1550-6207/05 $ 20.00 **IEEE**

information maintained in each node or static instruction is:

- The type of instruction and number of dynamic executions.
- The pointers to succeeding instructions in the dynamic execution stream with their corresponding frequencies (a single pointer in the case of arithmetic or memory instructions and multiple pointers in the case of conditional instructions and indirect jumps).
- The pointers to instructions that produce values that are consumed by the current instruction, pointers to instructions that consume values that are produced by the current instruction and their corresponding frequencies.
- The predictability of live-output values for different value predictors (stride and context based predictors are considered).
- The percentage of times that the value produced by the current instruction is never used. Even with aggressive compiler optimizations, there are opportunities for removing code [5] that may be only dead on a specific path.

2.2. Graph Analysis

Once the graph is built, several heuristics are applied to identify large regions of code that are suitable for traces. Several issues may be considered in the process of trace selection. These are related to the method for selecting the initial point of a trace, the final point and the predictability of live-output values.

A trace is considered a good candidate for speculation if the predictability of the live-output values achieves a certain threshold. Once live-output values are identified, their predictability has to be checked. Two types of statistics are analyzed: prediction accuracy and utilization degree, which refers to the percentage of times that the value produced by an instruction is not consumed by any other instruction. If a live-output value does not achieve a certain threshold in terms of value predictability but is not frequently consumed, it is considered predictable.

The initial and final points of a trace are the other important issues to be determined. Note that misspeculations occur when live-output values are misspredicted or the actual control flow does not reach the trace termination point. The trace termination point selected must try to maximize the trace length and minimize control flow misspeculations. Below we describe three basic heuristics for building traces: *procedure trace*, *loop trace* and *instruction chaining trac*e.

2.2.1. Procedure Trace Heuristic

Procedures are potential sources for trace speculation. They are relatively frequent in a program execution and the computations that follow a subroutine return are fairly independent of the subroutine, except for return values and some memory locations. This means that just a few values should be predicted. Also, the control return point is normally reached despite the complexity of the control flow inside the procedure, which means that it is quite easy to predict the end of the trace.

This heuristic tries to identify some procedures as traces. In this way, a call instruction is marked as the initial point of a trace, and the return address is set as its final point. Figure 1.a shows an example of procedure trace detection. Note that the whole subroutine is considered as a single trace regardless of the control flow followed at each invocation.

To determine the predictability of live-output values, a given number of instructions belonging to all significant paths after the execution of the procedure are checked. A

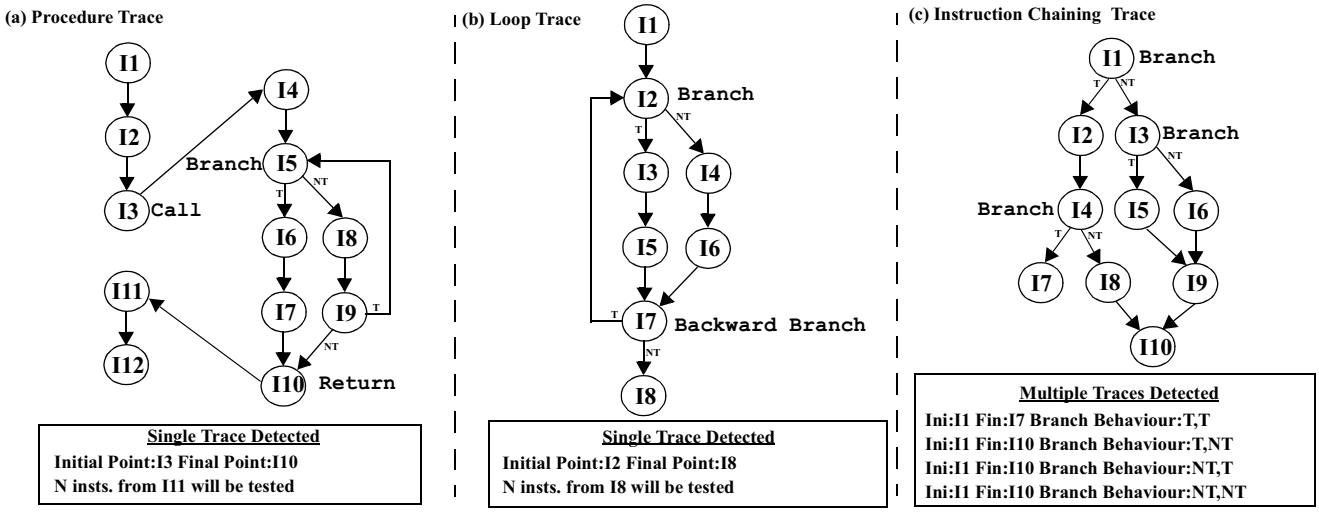

Figure 1. Trace Recognition Heuristics Examples

path is considered to be significant if its frequency of execution is above a certain threshold. For each instruction in a significant path it is checked whether any of its operands are produced by any instruction of the procedure. If this is the case, the predictability of the producer instructions are checked (through profiling) and if a certain threshold is not achieved, the trace is discarded. It has been empirically observed that there is no need to check too many instructions after the trace to identify good procedure traces. Moreover, binaries assumed in this paper (Alpha under Unix) help this validation because only a couple of registers are used to return values other than memory locations.

2.2.2. Loop Trace Heuristic

Loops are a traditional source of parallelization and speculation. This heuristic considers the whole execution of a loop as a trace. The aim of this heuristic is to detect loops whose live-outputs after their whole execution are predictable (in fact, we are only concerned with outputs that are consumed relatively early).

This heuristic sets the initial point of a trace as the target of a backward branch and the final point of the trace is the fall-through instruction of the same backward branch. Figure 1.b shows an example of a loop trace. Note again that the whole loop is considered as a single trace regardless of the control flow followed at each invocation. As for subroutines, the predictability of the live-output values is checked by analyzing a given number of instructions belonging to the significant paths after the execution of the loop. The trace is selected only if the predictability of the producer instructions is above a certain threshold.

2.2.3. Instruction Chaining Heuristic

The aim of this heuristic is to identify large sequences of dynamic instructions besides procedures and loops (and not necessarily contiguous in the static binary), with potential for speculation.

First, the initial point of a trace is selected. The taken and non-taken targets of all conditional branches are considered as initial points of a trace The trace is then extended by adding successive instructions until a final point of the trace is reached. A trace reaches its final point when a new instruction already belongs to the same trace, the trace reaches a maximum size, or the new instruction is an indirect jump.

A trace in this case corresponds to a single control-flow path. Therefore, every time a conditional branch is found, a trace is split into two, one for each potential path. Figure 1.c shows an example of various traces with the same initial point. Each trace is identified by its initial point, its final point, and the behavior of the conditional branches within the trace. To limit the number of different traces with the same initial point, paths whose frequency of execution is below a given threshold are ignored.

Once a candidate trace has been identified, its live-output values are determined and its predictability is checked. For each live-output value, the highest value between its prediction accuracy and its utilization degree is chosen. The percentages of different live-outputs are then multiplied to estimate the probability that the trace is correctly speculated (a value is correctly speculated if it is correctly predicted or if it is not frequently used). If this probability is above a certain threshold, the trace is considered predictable and the process finishes. Otherwise, the final instruction of the trace is removed and the process starts again. This process is repeated until the trace achieves the defined threshold or the size of the trace reaches a minimum. Note that this process tries to select the longest predictable traces.

3. Trace-Level Speculative Multithreaded Architecture

Trace-level speculation can be implemented in various ways. It generally requires a live-input or live-output test to validate the speculation. 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. 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 first thread, called the 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 the non-speculative thread. 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*.

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. In this paper, we extend the TSE engine proposed in [16] to support a trace selection method based on compiler analysis. The above off-line profile-guided analysis determines trace candidates to be speculated. These selected traces are communicated to the hardware at program loading time by filling a special hardware structure called trace table. We assume in this paper a simple 4-way set associative PC-indexed table with 128 entries. We have empirically observed that this number of entries and this degree of associativity leads to a good distribution of traces along the structure, and minimizes aliasing. As shows Figure 2, each entry contains the following information:

- *PcIni*: the initial program counter of the trace.
- *PcFin*: the final program counter of the trace.
- *BranchHist*: some bits that encode the history of some preceding branches.
- *LOValues*: value prediction information of N live-output values.
- *FreqCount*: a counter that determines the number of times that the trace has been found.

Live-output values are predicted by means of a hybrid scheme comprising a stride predictor and a context-based predictor. Based on the data in the trace table, the trace speculation engine (TSE) responsible for detecting initial and final points of a trace, maintaining value prediction information to compute live-output values, updating branch history and incrementing frequency counter. When the frequency counter of a trace reaches the maximum value, all frequency counters of traces with that initial program counter are initialized to zero.

The TSE also has to determine trace speculation opportunities by scanning the current program counter of the speculative thread and checking it against the trace table. In this way, if the current PC is the beginning of a potentially predictable trace, the trace is speculated since the architecture is very tolerant to trace misspredictions. As we discussed in Section 2.2.3, multiple traces with the same initial program counter may be stored in the trace table. In this case, the trace predictor selects a trace from those with the same initial point based on the history of the preceding branches. If the current branch history matches that of a stored trace, this trace is selected for speculation. If no branch history matches the current one, the most frequent trace is selected among all with the same initial program counter by checking frequency counters.

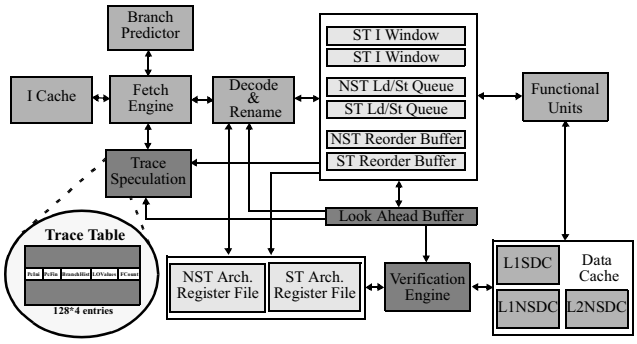

Figure 2. Trace level speculative multithreaded microarchitecture

Once the TSE determines that the current PC is the beginning of a potentially predictable trace, it provides the final program counter for the fetch engine. Also, some MOV instructions are generated in order to initialize the live-outputs with the predicted values. Further details may be found in [16].

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 implies flushing the ST pipeline and reverting to a safe point in the program. An advantage of 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. In this way, the NST quickly catches up to the ST.

A critical feature of this microarchitecture is that this recovery is implemented with minor performance penalties. Also, this paper extends the previous TSMA microarchitecture with a novel verification engine that can significantly improve performance. This novel verification engine does not always produce a thread synchronization in the presence of a trace misspeculation. Therefore, the number of recoveries may be decreased without increasing the complexity. Further details of this architectural enhancement may be found in [17].

Figure 2 shows the proposed microarchitecture with the additional hardware requirements highlighted. The hardware can be divided into three categories:

1. *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 grey in Figure 2)
2. *Shared*: non-replicated hardware is shared by both threads. These resources are the instruction cache, the fetch engine, the branch predictor, the decode and rename logic, functional units, a modified data value cache and logical control. (grey in Figure 2)
3. *Additional*: hardware requirements to support trace-level speculation. These resources are the look ahead

buffer, the verification engine and the trace speculation engine. (dark grey in Figure 2).

4. Performance Evaluation

This section discusses the experimental framework and analyzes the performance of the proposed scheme.

4.1. Experimental Framework

The TSMA simulator is built on top of the Simplescalar Alpha toolkit [4]. The following Spec2000 benchmarks have been considered: *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 have been compiled with the DEC C and F77 compilers with -non_shared -O5 optimization flags (i.e. maximum optimization level)

Table 1 shows the main parameters used in the program analysis phases. These values have been empirically checked to represent a good design point. First, it is important to minimize the number of misspeculations without losing speculation opportunities. In this way, the percentage of speculated traces is higher when the trace recognition heuristics are less conservative, but this also increases the percentage of misspeculation. However, the percentage of speculated traces and therefore the opportunities for speculation, decreases when the trace recognition heuristics are more conservative. Second, it is. important to maximize the number of speculated instructions and minimize the number of trace speculations. This means speculating traces as long as possible since every speculation introduces a minor penalty. Unfortunately, speculation accuracy decreases when the traces are larger because a huge number of live-output values have to be predicted. For the profiling data, each program was run with the test input set and statistics were collected for 250 million instructions after skipping initializations.

Value predictors considered	stride &context
Minimum size of trace	16
Maximum size of trace	1024
Maximum number of live-output values	32
Minimum combined percentage to consider a set of live-output values predictable	25%
Minimum frequency to consider a path as significative	10%
Minimum accumulative frequency to consider multiple paths	1%

Table 1. Profiling Analysis Parameters

Table 2 shows the parameters of the baseline microarchitecture. The TSMA assumes the same sizes as the baseline configuration and for each thread unit replicates the instruction window, reorder buffer and logical register mapping table. It also adds some new structures (see Table 3).

For the simulation, each program was run with the ref input set and statistics were collected for 250 million instructions after skipping initializations.

4.2. Analysis of Results

Figure 3 shows the type of speculated instructions corresponding to instruction chaining traces, call traces and

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
Architectural 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 2. 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 block verification if fail in L1. Number of additional instructions verified after average number to find an error is 16
Trace speculation engine	128 history table, 4-way set associative. Hybrid predictor (stride + context)
Look ahead buffer	128 entries

Table 3. Parameters of TSMA additional structures

loop traces. Note that almost 45% of the speculated instructions are due to speculation of instruction chaining traces, 40% are due to speculation of call traces and the remaining 15% correspond to speculation of loop traces. Although the numbers of speculated call and loop traces are relatively small, they are significantly larger than instruction chaining traces. Table 4 shows that loop traces have an average trace size of 215.8 instructions, while instruction chaining traces have an average size of 36.4 instructions. Other statistics, such as the average number of live-output values and average numbers of branches within a trace, are also shown in Table 4.

Average size of speculated traces per type (Instruction Chaining, Calls and Loops)	36.4, 97.3, 215.8
Average size of speculated traces	65.7
Average number of live-output values	16.4
Average number of branches within a trace (Instruction Chaining Heuristic)	5.3
Average number of traces with the same initial point (Instruction Chaining Heuristic)	1.57

Table 4. Additional simulation results

Note that the number of skipped instructions is larger when the traces are larger. However, this also implies a larger number of live-output values and therefore increases the probability of a live-output misspredict ion. The best performance depends on finding the best trade-off the between size of the traces and the predictability of their live-output values.

A trace misspeculation can be produced by incorrectly predicting a live-output value or incorrectly predicting the final point of a trace. Also, the final point of a trace may be correctly predicted but paths between the initial and the final point of a trace may be incorrectly predicted. Note that this does not necessarily produce a misspeculation. For example, if-then-else structures that do not generate different live-output values may produce different traces

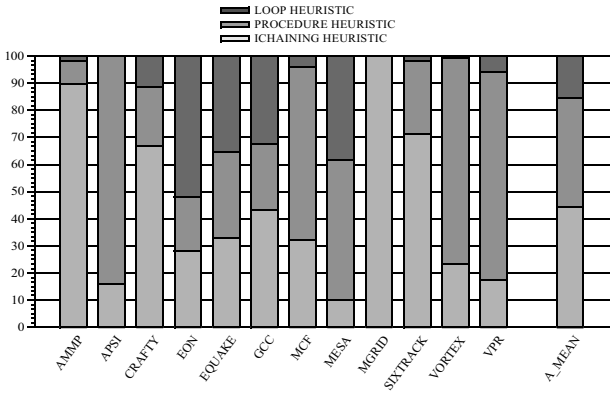

Figure 3. Type of speculated instructions

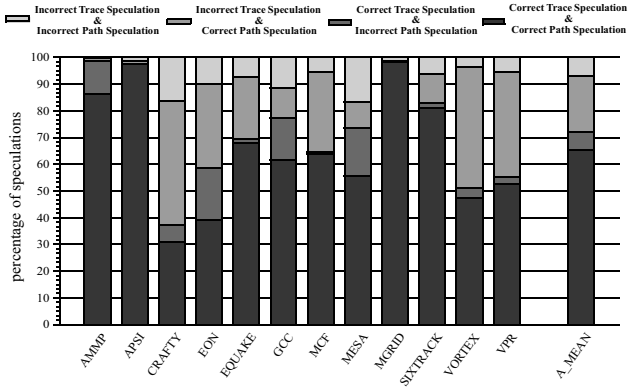

Figure 4. Type of speculations

with the same initial and final points.

Figure 4 shows the distribution of speculated traces divided into four categories: (1) correct trace speculation and correct path speculation, (2) correct trace speculation despite incorrect path speculation, (3) incorrect trace speculation but correct path speculation, and finally (4) incorrect trace speculation and incorrect path speculation. We observe a significant percentage of correctly speculated traces (almost 70%). Note that the contribution of traces that do not produce misspeculation, even though the paths between the initial and the final point of the trace were not correctly predicted, is around 7%. On the other hand, the percentage of misspeculations is close to 30% (21% for correctly predicted paths and 9% for misspredicted paths or misspredicton of the final point of a trace). These results confirm that the proposed mechanism for predicting paths and final points of traces provides a significant level of accuracy.

Figure 5 shows the speed-up obtained by the TSMA processor over the baseline superscalar configuration. Our results show that the average speed-up was almost 38% and very high speed-ups were achieved for all benchmarks. Note that significant speed-up was obtained despite misspeculating an average of 30% of the traces. These results also confirm that the proposed microarchitecture is

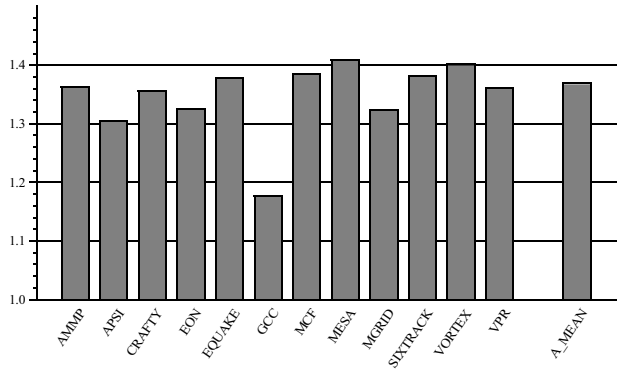

Figure 5. Speed-up

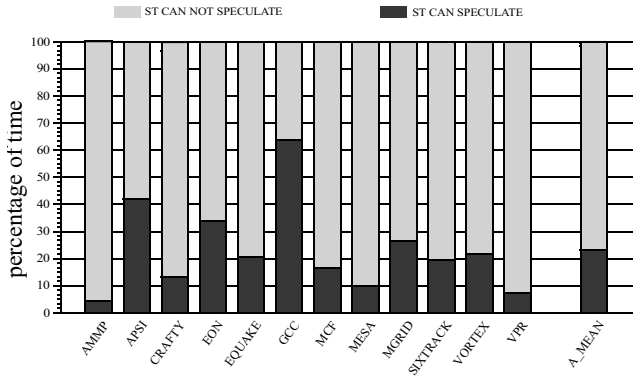

Figure 6. Type of cycles of the Speculative Thread

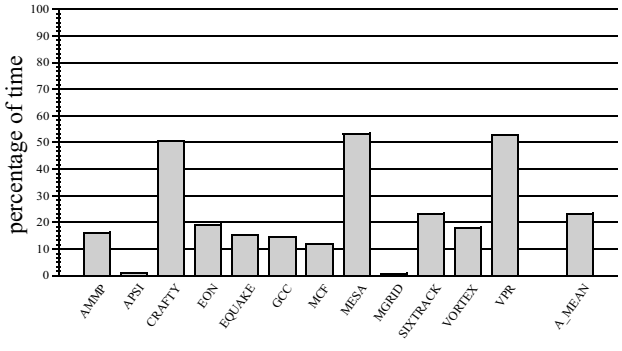

Figure 8. Useless cycles of the Speculative Thread

tolerant to misspeculations and encourage further work to develop more aggressive trace prediction mechanisms.

Figure 6 and Figure 7 provide several statistics about the activity of the speculative thread and the non-speculative thread, respectively. The dark-grey bar in Figure 6 represents the percentage of time that ST can speculate but does not find a trace to be speculated, while the light-grey bar represents the percentage of time that ST cannot speculate traces because NST is executing and verifying a speculated trace. Note that speculation may be performed only when NST catches up to ST. On average, almost 25% of the time the trace speculation engine did not communicate a trace speculation opportunity to the fetch engine because of this reason, which again confirms that performance may be improved by further analysing of the impact of the trace size. Note that the ideal scenario is when the ST finds a point to speculate right after the NST has caught up to it.

The dark-grey bar in Figure 7 represents the percentage of time that the NST executes traces speculated by ST, while the light-grey bar represents the percentage of time that the NST verifies instructions from the look-ahead buffer. In general, more speculated instructions (see Figure 6) imply more time executing instructions for the NST and, since verifying instructions is faster than executing them, this follows a superlinear relation.

Figure 8 shows the percentage of time that ST executes instructions beyond a misspeculation point. On average ST wastes up to 20% of the time executing instructions that will be discarded. Note that the ideal scenario would be when this percentage is negligible, which also implies a minimal number of trace misspeculations.

Finally, we also observed that, despite the significant number of branches within the trace, the instruction chaining heuristic does not provide many traces with the same initial point (see Table 4). In this way, we studied branch behavior and concluded that the majority of branches almost always take the same direction. Figure 9 shows the accumulated distribution of the branch behaviour for all the benchmarks used in this paper. The X-axis represents the percentage of times that a branch takes the most common direction (50% means that the branch takes the taken and the not taken paths the same number of times and 100% means that the branch always takes the same path). The Y-axis represents the accumulated number of dynamic branches. Note that almost 80% of the branches take the same direction more than 90% of the times. This result, combined with the parameters used for the analysis phase (listed in Table 1), significantly limits the number of traces with the same initial point.

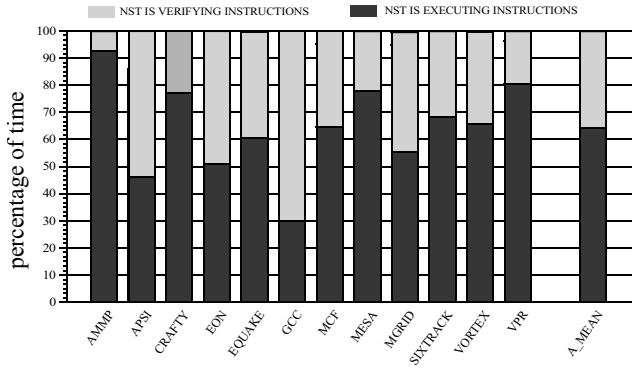

Figure 7. Type of cycles of the Non Speculative Thread

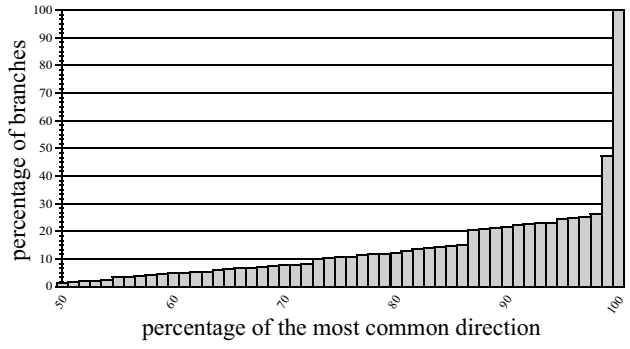

Figure 9. Branch Behaviour Distribution

5. Related Work

Several previous studies [7],[11],[12],[21] have shown that programs usually have a significant degree of repeatability/predictability, which suggests that there may be effective schemes to significantly increase the accuracy of trace predictors. [19], identified the potential sources of speculative parallelism in programs and concluded that a combination of loop and procedural speculation is a promising parallelization scheme for speculative thread-level parallel machines

Value profiling has also been studied as a mechanism to assist value prediction or value reuse schemes. A value prediction scheme guided by value profiling is presented in [10]. Compiler-directed approaches for identifying code regions whose computation can be reused during dynamic execution are proposed in [7] and [13]. A code specialization approach that uses value profiling is presented in [18]. A compiler framework that includes analysis for speculative optimizations has recently been proposed [14]. This uses profiling information and simple heuristics to supplement traditional non-speculative compile-time analysis.

The idea of dynamic verification was introduced in [22]. The proposed AR_SMT processor uses a time redundant technique that enables some transient errors to be tolerated. Slipstream processors [20] dynamically avoid the execution of non essential computations of a program. These authors suggest creating a shorter version of the original program by removing ineffectual computation. The use of dynamic verification to reduce the burden of verification in complex microprocessor designs is covered in [8].

Several thread-level speculation techniques have been explored to exploit parallelism in general-purpose programs. Speculative multithreading [2],[15] is a well-known technique based on the concurrent execution of speculative threads. Various authors have studied the impact of different value predictors to alleviate dependence constraints and enable look-ahead execution of speculative threads. Simultaneous Multithreading [25] allows independent threads to issue instructions to multiple functional units in a single cycle. Multiple Path Execution [1],[26] permits the speculative execution of multiple paths in parallel. Simultaneous Subordinate Microthreading [6] was proposed in order to execute subordinate threads that perform optimizations on a primary thread.

Other recent studies have also focused on speculative threads. The pre-execution of critical instructions by means of speculative threads has also been proposed [9],[23],[27]. 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 which therefore warms up certain microarchitectural structures.

6. Conclusions

In this paper we propose a profile-guided analysis for identifying highly predictable, large traces to be speculated by a *Trace-Level Speculative Multithreaded Architecture*. We propose three basic heuristics to determine opportunities for speculation. This analysis substitutes the dynamic process of detecting speculative traces and their corresponding live-output values, which considerably reduces hardware complexity. Our simulation results show that these techniques achieve an average speed-up of almost 38%.

Future areas for investigation include generalising the architecture to multiple threads in order to perform sub-trace speculation during the validation of a trace that has been speculated. The relatively low penalty of misspeculations means that another area for future work is to investigate more aggressive speculation schemes.

7. 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.

8. 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] J. A. Butts and G. Sohi, “Dynamic Dead-Instruction Detection and Elimination”, *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[6] 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.

[7] 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.

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

[9] 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.

[10] F. Gabbay and A. Mendelson, “Can Program Profiling Support Value Prediction?”, I*n Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[11] A. González, J. Tubella and C. Molina, “Trace level Reuse”, *In Proceedings of the International Conference on Parallel Processing*, 1999.

[12] 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.

[13] J. Huang and D. J. Lilja, “Extending Value Reuse to Basic Blocks with Compiler Support”, *IEEE Transactions on Computers*, 2000.

[14] J. Lin, T. Chen, W. C. Hsu, P. C. Yew, R. D. C. Ju, T. F. Ngai, S. Chan, “A Compiler Framework for Speculative Analysis and Optimizations”, *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

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

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

[17] C. Molina, A. González and J. Tubella, “Reconsidering Misspredictions in Trace-Level Speculative Multithreaded Architectures”, *Technical Report UPC-DAC-2004-20, Universitat Politècnica de Catalunya,* 2004.

[18] R. Muth, S. Watterson and S. Debray, “Code Specialization Based on Value Profiles”, *In Proceedings of the 7th. International Static Analysis Symposium*, 2000.

[19] J. Oplinger, D. Heine, M. S. Lam, “In Search of Speculative Thread-Level Parallelism”, *In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques,* 1999.

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

[21] E. Rotenberg, “Exploiting Large Ineffectual Instruction Sequences”, Technical Report, North Carolina State University, 1999.

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

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

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

[25] 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.

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

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