A Quantitative Assessment of Thread-Level Speculation Techniques

Pedro Marcuello and Antonio González

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain

Abstract

Speculative thread-level parallelism has been recently proposed as an alternative source of parallelism that can boost the performance for applications where independent threads are hard to find. Several schemes to exploit thread level parallelism have been proposed and significant performance gains have been reported. However, the sources of the performance gains are poorly understood as well as the impact of some design choices. In this work, the advantages of different thread speculation techniques are analyzed as are the impact of some critical issues including the value predictor, the branch predictor, the thread initialization overhead and the connectivity among thread units.

1. Introduction

Dynamically-scheduled superscalar processors have become the most popular processor microarchitecture in the recent years and many studies have been devoted to improve their performance. This processor microarchitecture is characterized by its ability to exploit the instruction-level parallelism (ILP) that inherently exists in programs at run-time.

The exploitation of ILP in superscalar processors is strongly constrained by several significant hurdles, such as the instruction window size [22] and the data dependences (RAW dependences) [22].

To exploit large amounts of ILP, dynamically-scheduled superscalar processors require a large instruction window filled with useful instructions so that their selection logic can search for independent instructions in a large number of them. However, the effective size of the instruction window is limited by the branch prediction accuracy since the amount of correctly speculated control-flow instructions depends on the number of consecutive branches that have been correctly predicted. This is due to the sequential nature of the fetching mechanism of superscalar processors since a single mispredicted branch prevents the instruction window from growing beyond the branch. For non-numeric programs, which have many difficult-to-predict branches, way be just by itself a very important limitation.

Data dependences are another important limit to ILP since instructions can only be issued for execution when all their operands have been produced. The execution ordering imposed by data dependences strongly limits the performance of superscalar processors. Experiments with ideal machines with infinite resources report that only serializing data dependent instructions produce average performance of just several tens of instructions per cycle (IPC) for some integer programs [22].

Data value speculation has recently been proposed to relieve the penalties due to data dependences and minimize their impact on the performance of the processor by means of predicting the input/output operands of instructions ([9] among others). However, recent studies show that the performance impact of data value speculation for superscalar processors is moderate, and its potential improvement approaches a linear function of the prediction accuracy [6]. On the other hand, the potential improvement is much higher for speculative multithreaded architectures.

Some microarchitectures have recently been proposed to boost performance by exploiting coarse-grain parallelism in addition to the instruction-level (fine-grain) parallelism. These microarchitectures split programs into specultive threads and then, they execute them concurrently. This kind of parallelism is referred to *Speculative Thread-Level Parallelism*. Threads are speculatively executed because they are in general both control and data dependent on previous threads, since independent threads are hard to find in many non-numeric applications such as the SpecInt95. These microarchitectures support execution roll-back in case of either a control or a data dependence misspeculation.

These microarchitectures provide support for multiple contexts and appropriate mechanisms to forward or to predict values produced by one thread and consumed by another. They differ in how programs are split into threads. In several proposals such as the Multiscalar [3][17], the SPSM architecture [2] and the Superthreaded architecture [19], the compiler splits the program into threads whereas others rely only on hardware techniques. Examples of the latter group are the Dynamic Multithreaded Processor [1] and the Clustered Speculative Multithreaded Processor [10][11].

These works have shown that speculative thread-level parallelism has significant potential to boost performance. However, most of them use different heuristics to partition a sequential instruction stream into speculative threads. Little insight exists to explain the source of the advantages for each particular partitioning approach and its interaction with value prediction and other microarchitectural components. In all cases significant benefits have been reported but the absolute figures are not comparable due to very different architectural assumptions. Some work comparing different spawning policies has been done[14]. However, their base architecture was an on-chip multiprocessor where each processing unit was one-way and issued the instructions in program order. In such architecture, the effects of the ILP on these techniques are omitted.

Here, the performance benefits of different approaches of thread-level parallelism are analyzed. Various previously proposed speculative thread spawning policies -loop iterations, loop continuations, subroutine continuations- are analyzed. The role of value prediction to avoid serialization of inter-thread dependent instructions is studied. Finally, the implications of implementation constraints including the thread unit interconnection topology, the branch prediction scheme and the thread initialization overhead are analyzed.

Below, Section 2 reviews related work. Different speculative thread spawning policies are described in Section 3 and their performance potential is analyzed in Section 4. Section 5 describes the impact of value prediction on the performance. Section 6 reveals the impact of the branch prediction scheme and the thread initialization overhead. Conclusions are summarized in Section 7.

2. Related Work

Several multithreaded architectures providing support for thread-level speculation have been proposed. Pioneer work on this topic was the Expandable Split Window Paradigm[3] and the follow-up work, the Multiscalar[17]. In this microarchitecture, the compiler is responsible for partitioning the program into threads based on several heuristics that try to minimize the data dependences among active threads or to have a better load balance, among other compiler criterias [21].

Other architectures such as the SPSM [2] and the Superthreaded architectures [19] also rely on the compiler to split the program into threads, but in these cases, threads are assumed to be loop iterations instead of the more complex analysis of the Multiscalar compiler.

On the other hand, some other architectures try to exploit thread-level parallelism speculating on threads dynamically created by the processor without any compiler intervention. The Speculative Multithreaded Processor [10] and its successor, the Clustered Speculative Multithreaded Processor [11] identifies loops at run time and simultaneously executes iterations in different thread units. Also, this architecture provides mechanisms for value prediction in order to execute the concurrent threads as if they were independent.

In the same way, the Dynamic Multithreaded Processor [1] relies only on hardware mechanisms to divide the sequential program into threads, but it speculates on loop and subroutine continuations. Moreover, the architectural design of the processor allows out-of-order thread creation which requires communication between any hardware context since it has a centralized implementation of hardware contexts in a similar way to the Simultaneous Multithreaded Architecture [20].

Trace Processors[15] also exploit certain kind of speculative thread-level parallelism. Its mechanism to split the sequential program into almost fixed-length traces is specially suited to maximize the load balance among the different thread units.

Finally, several works on speculative thread-level parallelism on multiprocessor platforms have appeared ([7][8][18] among others). In all cases, programs are split by the compiler and usually no mechanism for predicting dependent values among dependent threads is provided.

3. Dynamic Exploitation of Thread Level Parallelism

The particular approach to partitioning a sequential stream into threads, referred to as the thread spawning policy, is critical to performance. Desired properties for speculative threads are high predictability for control and data dependences. Regarding control dependences, the optimal choice would be that threads be control independent of previously executed (less speculative) threads or at least very likely to be executed.

Loop iterations are good candidates for speculative threads since once a loop iteration is started it is very likely that it is executed again. The scheme that spawns threads associated to loop iterations is called the *loop-iteration* spawning policy. Another highly predictable control-flow point is the continuation of a loop. Once a loop is started, the instruction that follows the backward branch that closes the loop in static order is very likely to be executed in the near future. We refer to the scheme that spawns a speculative thread starting at the instruction after each backward branch as *loop-continuation* spawning policy. Finally, another highly predictable control-flow point is the return address of a subroutine call. Every time that a subroutine call instruction is executed, the *subroutine-continuation* spawning policy spawns a speculative thread that starts at the instruction after the call in static order.

The threads that are running at any moment can be ordered according to their location in the sequential instruction stream. The first thread is not speculative whereas all the rest are speculative. We will say that thread A is less speculative than thread B if A precedes B in the sequential order.

4. Performance of Thread Level Speculation with Perfect Value Prediction

In this section, the performance of the different thread spawning schemes is investigated for two thread ordering schemes: restricted ordering (implemented on a point-to-point topology (ring)) and an unrestricted ordering implemented on a fully connected network (e.g. a crossbar). Either perfect value prediction or not prediction at all is assumed. Realistic value predictors are analyzed in the next section.

4.1. Experimental Framework

Consider a clustered microarchitecture made up of several thread units, each one being similar to a superscalar processor core. A clustered design was selected due to its scalability.

Performance statistics were obtained through trace-driven simulation of the whole SpecInt95 benchmark suite. The programs were compiled with the Compaq compiler for an AlphaStation 600 5/266 with full optimization (-O4) and instrumented by means of the Atom tool. For the statistics, we simulated 300 million of instructions after skipping initialization. The programs were executed with the ref input data since they reflect a more realistic workload, in particular for some parameters such as the number of loop iterations.

The baseline speculative multithreaded processor has a parameterized number of thread units (from 2 to 16) and each thread unit has the following features:

- Fetch: up to 4 instructions per cycle or up to the first taken branch, whichever is shorter.
- Issue bandwidth: 4 instructions per cycle

- Functional Units (latency in brackets): 2 simple integer (1), 1 integer multiplication (4), 2 simple FP (4), 1 FP multiplication (6), and 1 FP division (17).
- Reorder buffer: 64 entries.
- Local branch predictors: 14-bit gshare.
- 32 KB non-blocking, 2-way, local, L1 data cache with a 32-byte block size and up to 4 outstanding misses. The L1 latencies are 3 cycles for a hit and 8 cycles for a miss. Forwarding memory values from any thread unit to the consumer one has a cost of 3 cycles if perfect memory value prediction is not considered.

Consider two different spawning ordering policies: a sequential thread ordering policy and an unrestricted ordering policy. For the former, threads are created in a sequential order in such a way that one thread cannot be created to be executed between two current threads. The latter policy allows to create new threads out-of-order. Here no considerations are given to the particular bandwidth provided by each possible topology, so the network has a fixed delay of 1 cycle.

Three different approaches were used to deal with data dependences among instructions in different threads (inter-thread dependences for short) are analyzed. In the first model, all values corresponding to inter-thread dependences through both registers and memory are correctly predicted (referred to as perfect register and memory prediction in Figures 1 and 3). In the second model, inter-thread dependent register values are always correctly predicted but inter-thread dependent memory values must be forwarded from the producer to the consumer (referred to as perfect register prediction in Figures 1 and 3). Finally, the last model considers that the inter-thread dependences cause a serialization between the producer and the consumer and is called synchronization model in these figures.

Performance is by default reported as the speed-up over a single-threaded execution. A varying number of thread units ranging from 2 to 16 has been considered.

4.2. Sequential thread ordering

The most straightforward way to implement a speculative multithreaded processor is by interconnecting the different thread units by means of an unidirectional ring topology in such a way that the communication among them is restricted to only forward values from one unit to the following one. The main advantage of a ring is its low hardware complexity.

In a ring, the sequential order among threads (i.e. from less to more speculative) always corresponds with the physical order that they occupy once they have been mapped to their corresponding thread units. Therefore, when a thread (which may be speculative) decides to spawn a new speculative thread and the next thread unit is already busy, then the thread running on that busy unit is squashed, which in turn causes the squashing of the following threads. The new thread, which is less speculative than all the squashed ones, is allocated to the first freed unit.

However, simplicity of a ring is also a major drawback because it strongly limits the amount of speculative thread-level parallelism that the processor can exploit as we can observe in figure 1. This figure shows the speed-up of each benchmark over a single-threaded execution, as well as the harmonic mean, for a 16 thread unit configuration. The three bars for each program correspond to the three different approaches to deal with inter-thread dependences (synchronization, perfect prediction of register values and perfect prediction of register and memory values). This performance limitation is especially remarkable when the microarchitecture speculates on subroutines, for which the average speed-ups are rather low (13% of speed-up). This is due to the limited communication capabilities of the ring topology which restricts the thread speculation mainly to leaf subroutines. To illustrate this problem let us assume a subroutine A, which is being executed in thread unit 1, that calls subroutine B which in turn calls subroutine C. When the call to subroutine B is found, a speculative thread that executes the continuation of B (rest of subroutine A) is spawned and allocated to thread unit 2, whereas thread unit 1 proceeds with the execution of subroutine B. If later on subroutine B calls subroutine C, thread unit 1 proceeds with subroutine C and the speculative thread corresponding to the continuation of C (rest of B) is allocated to thread unit 2, which causes the thread running the continuation of B to be squashed.

On the other hand, we can observe that for the other spawning policies, speculating on loop iterations and speculating on the continuation loops, somewhat higher speed-ups are achieved, especially for the `ijpeg`. The reason is that in these two models, there are more threads that are created in their sequential order than for the subroutine model and thus, they generate less number of squash operations.

Note that the subroutine-continuation scheme does not improve the single-threaded execution of `compress` since the evaluated part of the program does not contain any subroutine call that returns to the following instruction in static order.

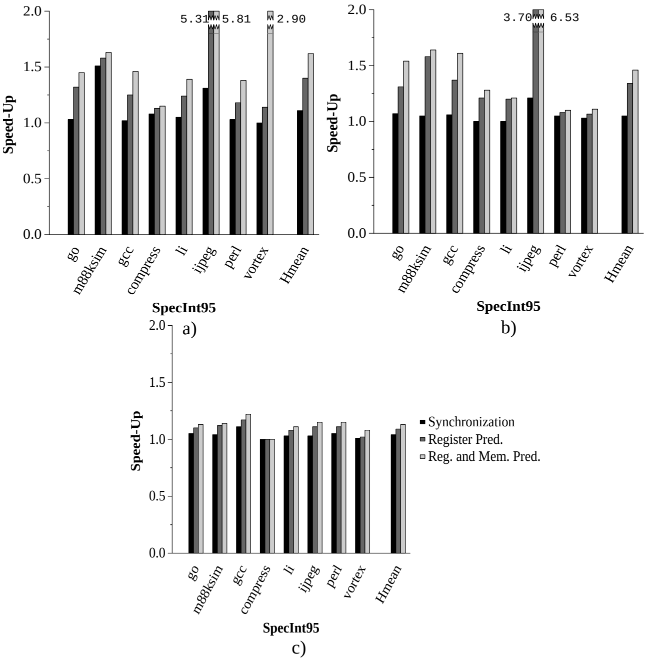


Figure 1: Speed-ups for the three different spawning policies a) loop-iteration, b) loop-continuation and c) subroutine-continuation for a ring topology.

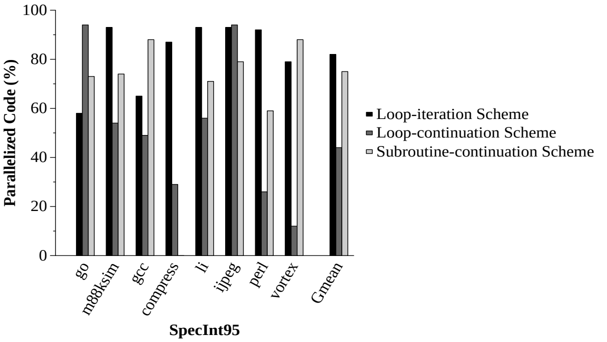


Figure 2: Percentage of parallelized code for each spawning policy

Therefore, for the average calculation of this scheme, `compress` is not considered.

4.3. Unrestricted thread ordering

Other thread unit interconnection topologies allow each thread unit to communicate with anyone else. In this case, when a new speculative thread is created, any thread unit can be allocated for its execution. An idle unit will be chosen if any; otherwise, the unit running the most speculative thread will be chosen and the thread currently running on it will be squashed. This thread management is more complex than that of the ring topology because the hardware must be aware of the logical (sequential) order of the speculative threads running in parallel, since now, the logical order does not correspond anymore to the physical order of thread units and the hardware must provide full connectivity among thread units. Relaxing this constraint allows the processor to speculate at different loop nesting levels, that is, the processor can simultaneously execute different iterations of different nested loops or different nested subroutines instead of the single level of speculation that permits the ring topology.

Allowing the processor to speculate at different levels highly increases the percentage of code which can be parallelized as it is shown in Figure 2. Observe that for codes such as `m88ksim`, `li`, `perl` and `ijpeg` practically the whole code can be parallelized by speculating on loop iterations. On average, 83% of the code is parallelized for this scheme. The percentage of parallelized code for the subroutine spawning policy is also quite high (76% on average) and this percentage suffers a significant drop for the loop-continuation scheme, which explains its much lower performance.

This efficiency of speculatively parallelizing code is translated in high speed-ups as shown in Figure 3, especially for the models that speculates on loop iterations and on subroutines. For these two scenarios, the processor can achieve an average speed-up of 5.66 and 3.08 respectively with 16 thread units. Nonetheless, observe that the speed-up achieved by the model that speculates on loop continuations are lower than those of the two other policies (only 2.07 on average). This is due to the fact that this policy is very similar to speculating on loop iterations for any non-innermost loop because in a loop nest, the continuation of a given loop corresponds to an iteration of the next outer loop, in such a way that the granularity of the speculated threads will be larger than for speculating on loop iterations. The size of the speculated threads depends on the number of iterations of the innermost loop and small differences of them cause load unbalance and this significant loss in performance. Besides, impact of value prediction is more significant for small threads than for larger ones.

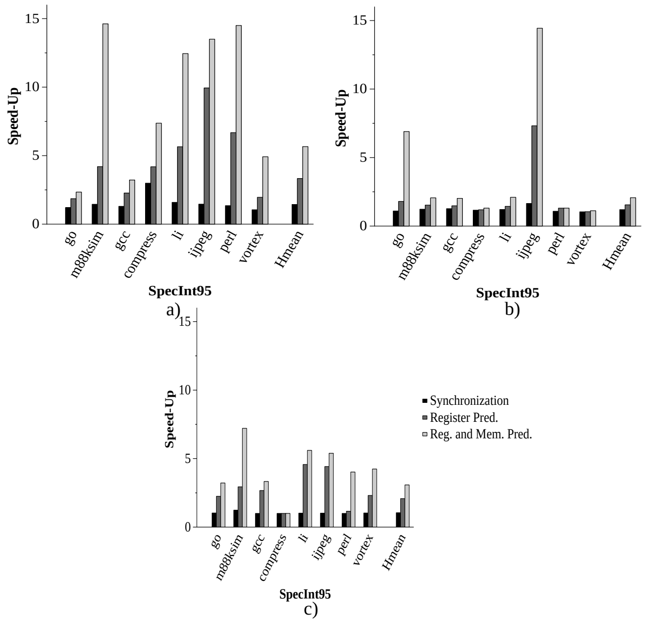


Figure 3: Speed-ups for the three different spawning polices a) loop-iteration, b) loop-continuation and c) subroutine-continuation with unlimited interconnectivity

Figure 4 shows the average speed-up for a number of thread units ranging from 2 to 16, for the loop-iteration and subroutine-continuation spawning schemes. For each scheme, the figure shows the speed-up achieved predicting register values and both register and memory values. Observe that performance scales quite well for all cases, specially when all data values are predicted. Moreover, the performance for the loop-iteration scheme is always higher than for the subroutine-continuation approach.

To summarize, ring topologies have a very limited performance potential. Even for the best performing thread speculation model (loop-iterations) with perfect value prediction for both inter-thread register and memory dependences, the speed-up achieved by 16 thread units is lower than 1.7. The unlimited connectivity architecture has a much higher performance potential. For perfect value prediction the

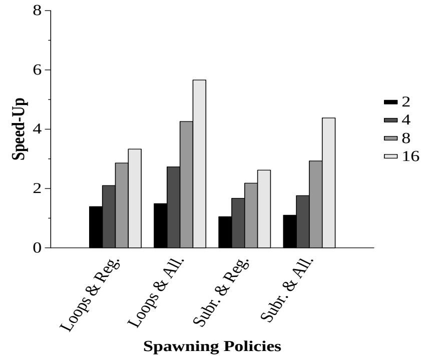


Figure 4: Speed-up for different number of thread-units.

best performing thread speculation model is again the loop-iteration and the average speed-up is close to 6 for 16 thread units. Finally, in addition to the higher hardware complexity of the unrestricted ordering policy, it degrades the branch prediction accuracy as it will be shown in Section 6.

5. The Role of Value Prediction

In Figures 1 and 3, value prediction is shown to play a crucial role in speculative multithreaded processors. For all speculation models and the two ordering policies, the gap between no value prediction and perfect prediction is wide. Even the best performing model without value prediction (loop-iterations and unrestricted ordering) would just perform slightly better than a single threaded execution (1.43 average speed-up), which would not justify the added complexity and cost of supporting multiple speculative threads.

Moreover, the quite different behavior of value prediction for the two interconnection topologies is revealed. For a ring topology, predicting register values has much more relevance than predicting memory values. For instance, for the loop-iteration model, adding memory value prediction on top of register value prediction only increases the speed-up by 10% when `vortex` is not considered and from 22% if it is. On the other hand, for an unlimited connectivity scenario, predicting memory values becomes an important source of speed-up since the effective instruction window size becomes huge and in general, the average distance between memory dependent instructions is much longer than that of register dependent ones.

Nevertheless, even though memory value prediction plays an important role in these architectures and it is worth investigating, this paper focuses on register value predictors, since they are much simpler due to the limited number of register input values that each thread has to predict and they have just by themselves a significant performance potential.

In this section, we evaluate realistic value predictors and how they affect the performance of speculative multithreaded architectures with unlimited connectivity for loop-iteration and subroutine-continuation spawning policies. These are the two schemes with most potential, as shown in the previous section.

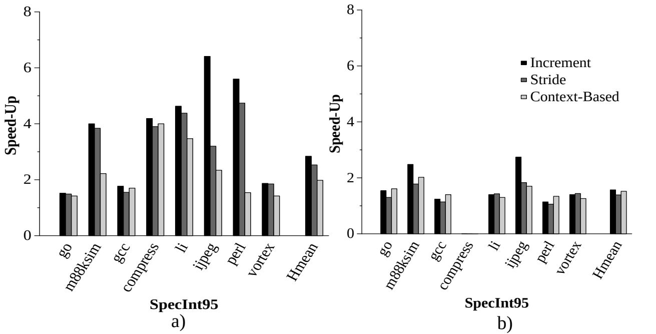
5.1. Value predictors

Value prediction is a technique which tries to improve performance by allowing instructions to be executed before their operands are computed. It is based on the observation that values usually repeat or follow a fixed pattern over the time.

Value prediction will only be applied to those registers that are live at the beginning of a thread (those registers that will be read before being written during the execution of that thread) and that are written by any previous (in the logical order) concurrent thread.

Three value predictors are considered. Two of them, the stride [4][5] and the FCM context-based value predictors[16] are well-known and have been thoroughly studied for superscalar and VLIW architectures. We also evaluate the increment value predictor [12].

The increment value predictor is an output-value trace-oriented predictor that predicts each output value of a thread to be the value of that register at the beginning of the thread plus an increment. This increment is computed as the value of this register at the end of the thread minus the value at the beginning of it for a previous execution of the same thread with the same control-flow as that predicted for the current one.



a) b)

Figure 5: Speed-ups for the different value predictors and for the a) loop-iteration and b) subroutine-continuation spawning schemes.

5.2. Performance Evaluation and Value Prediction Accuracy

Centralized value predictors with relatively small size (limited to 16 KB) are considered. In particular, each table of the FCM context-based value predictor has 1024 entries, while the tables for the increment and the stride predictor have 2048 entries. The predictors are only requested to predict those values that are live at the beginning of the thread. When a value that is live at the beginning of a thread is mispredicted, in addition to wait for the computation of the value, a 1-cycle penalty is assumed.

Figure 5 shows the speed-up achieved by the different predictors when the processor speculates on loop-iterations and subroutine-continuations. Observe that the behavior of the predictors is different depending on the spawning policy. For loop-iteration, the increment predictor outperforms the others.

Moreover, observe in figure 5 and figures 1 and 3 that the gap in performance between perfect prediction and realistic prediction is quite wide and very different depending on the particular spawning policy. In general, speculating on subroutines is more degraded by realistic value predictors than speculating on loop iterations. On average, in comparison with perfect register value prediction, the losses due to a realistic value predictor are 16% on the loop-iteration scheme and 50% on the subroutine-continuation. This is explained by the lower value prediction accuracy for the former scheme (between 80% and 69% on average) as shown in figure 6 and its higher misspeculation penalty. In fact, 10% more values are mispredicted for the subroutine-continuation scheme than for the loop-iteration scheme. This penalty is related to the average

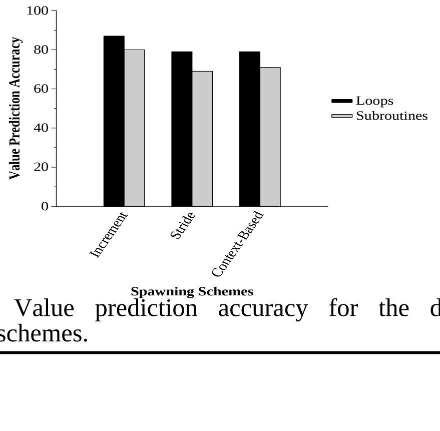


Figure 6: Value prediction accuracy for the different spawning schemes.

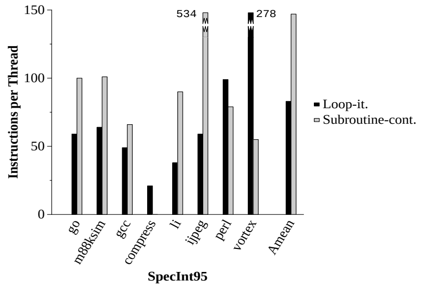


Figure 7: Average thread size.

size of threads, since misspeculated values have to wait until they are produced by previous threads. As figure 7 shows, the average number of instructions per thread is much higher (close to the double) for the subroutine spawning policy, so when the processor has to wait for the computation of any value, on average, it must wait longer than for the loop-iteration scheme.

Overall, the benefits of speculative thread-level parallelism are still quite high. The loop-iteration model achieves an average speed-up of 2.84.

6. Other Implementation Issues

In addition to the value prediction accuracy, there are other important factors for the performance of a speculative multithreaded processor. This section studies the impact of the branch predictor and the thread initialitation overhead.

6.1. Distributed Branch Predictor with Local Updates

A centralized branch predictor would not be adequate for speculative multithreaded processors since it would require a large number of ports. In addition, branches are not processed in sequential order, and thus the history information, especially global history registers, would be significantly degraded by this. In the previous experiments a distributed branch prediction scheme was assumed that consists of a local branch predictor for each thread unit that work completely independent of the other predictors once a thread is started. Here, when a thread is initialized, its branch predictor table is copied from the table of the parent thread. This may imply a too high initialization overhead. Alternatively, branch prediction tables could be not initialized at thread creation. Instead, when a new thread is started in a thread unit, it simply inherits the prediction table as it was left by the previous thread executed in that unit.

Predicting the outcome of branches only based on the history of branches executed in the same thread unit may cause negative effects in the accuracy of the predictor and therefore, in the overall performance of the processor. In this section these performance implications are evaluated.

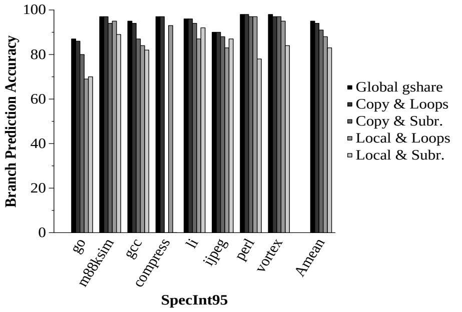


Figure 8: Branch prediction accuracy

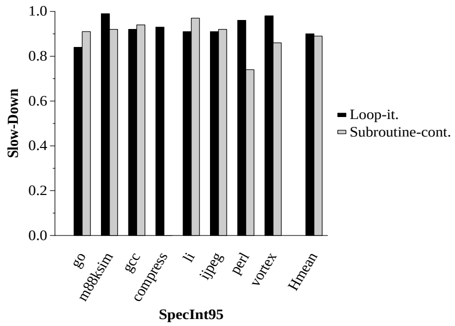


Figure 9: Slow-down when independent local branch predictors are used.

Figure 8 compares the branch prediction accuracy of a speculative multithreaded processor with branch predictors initialized from the parent at thread creation and that of a non-initialization policy. In addition, it also shows the prediction accuracy of a centralized predictor that processes all branches in sequential order as a superscalar microprocessor does, as a baseline for comparison. Observe that the degradation suffered when the copy mechanism is implemented is very low (only 1% for a loop-iteration spawning policy and 4% for the subroutine-continuation one), but it is significant when predictors are independently managed (higher than 10% on average).

Figure 9 shows the impact of this loss in branch prediction accuracy on the overall performance of the speculative multithreaded processor. This figure depicts the slow-down caused by not initializing the local predictors. On average, the slow-down is close to 10% and significant for some programs such as `perl` for the subroutine continuation spawning policy.

6.2. Initialization Overhead Penalty

Starting a new thread in a thread unit requires several operations that may take some non-negligible time. In particular, registers that are live at the beginning of a thread must be initialized with their predicted values and the remaining registers that are not written by this new thread and may be read by any subsequent thread must be initialized with the same value as the parent thread, either at thread creation or when the parent produces this value. The penalty associated to all these operations is referred to initialization overhead. Note that several registers per cycle can be read/written in a multi-

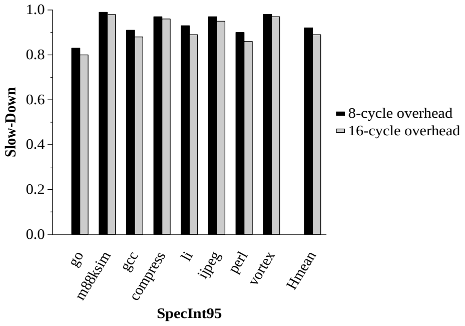


Figure 10: Slow-down when an overhead penalty is considered.

port register file, and several values can be forwarded in parallel depending on the bandwidth of the interconnection network.

In this section, the impact of the initialization overhead for a penalty of either 8 or 16 cycles is evaluated, which may be reasonable for 32 integer and 32 FP registers. Figure 10 shows that on average, the performance loss is about 8% for the loop-iteration spawning scheme with increment register value prediction, which is the one with the highest performance, with an 8-cycle overhead penalty, and 10% when a 16-cycle overhead penalty is considered.

7. Conclusions

There is a significant performance impact of the main design parameters of processors that exploit speculative thread-level parallelism. Shown here, a restricted-ordering policy limits the performance since it can only speculate successfully when speculative threads are created in their sequential order. On the other hand, an unrestricted-ordering spawning policy provides very high performance benefits. The loop-iteration speculation scheme would be the one with the highest performance if all values corresponding to inter-thread dependences could be correctly predicted. When a realistic predictor is considered, the loop-iteration spawning scheme is the best performing one, since values are more predictable due to the smaller average thread size. For this spawning scheme, among the different value predictors analyzed, the increment predictor is the most accurate one. Thread initialization overhead can have a low impact on performance provided that branch prediction tables do not need to be initialized at thread creation. The price to pay for that is a small reduction in branch prediction accuracy which translates in a small performance degradation.

Overall, the loop-iteration spawning policy with an increment predictor and an unlimited connectivity architecture is an effective design to exploit speculative thread-level parallelism. On average, it provides a speed-up of 2.70 for 16 thread units. Finally, several spawning policies may be implemented in the same microarchitecture, together with some heuristics that identify the most effective one for each particular section of code.

8. Acknowledgments

This work has been supported by grants CICYT TIC 511/98, ESPRIT 24942 and AP96-52274600 and CEPBA. The authors would like to thank Bodo Parady (Sun Microsystems) his insightful comments on this paper.

9. References

[1] H. Akkary and M.A. Driscoll, “A Dynamic Multithreading Processor”, *Proc. 31st. Int. Symp. on Microarchitecture*, 1998.

[2] P.K. Dubey, K. O’Brien, K.M. O’Brien and C. Barton, “Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading”, in *Proc. of the Int. Conf on Parallel Architectures and Compilation Techniques*, pp. 109-121, 1995.

[3] M. Franklin and G. Sohi, “The Expandable Split Window Paradigm for Exploiting Fine Grain parallelism”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 58-67, 1992.

[4] F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction”, *Technical Report #1080, Technion*, 1998.

[5] J. González and A. González, “Memory Address Prediciton for Data Speculation”, *Technical Report UPC-DAC-1996-50.*

[6] J. González and A. González, “Data Value Speculation in Superscalar Processors”, in *Microprocessors and Microsystems*, 22(6), pp. 293-302 November 1998

[7] L. Hammond, M. Willey and K. Olukotun, “Data Speculation Support for a Chip Multiprocessor”, *Proc. of Int. Conf. on Architectural Support for Prog. Lang. and Op. Systems*, 1998.

[8] V. Krishnan and J. Torrellas, “Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor”, in *Proc. of the 12th. Int. Conf. on Supercomputing*, pp. 138-147, 1998.

[9] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, “Value Locality and Load Value Prediction”, in *Proc. of the 7th. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, Oct. 1996.

[10] P. Marcuello, A. González and J. Tubella, “Speculative Multithreaded Processors”, in *Proc. of the 12th Int. Conf. on Supercomputing*, pp. 77-84, 1998.

[11] P. Marcuello and A. González, “Clustered Speculative Multithreaded Processors”, in *Proc. of the 13th Int. Conf. on Supercomputing*, pp. 365-372 1999.

[12] P. Marcuello, J. Tubella and A. González, “Value Prediction on Speculative Multithreaded Architectures”, in Proc. of the 32nd. Int. Symp. on Microarchitecture, pp. 230-236, 1999.

[13] D. Matzke,”Will Physical Scalability Sabotage Performance Gains”, *IEEE Computer* Vol. 30, num 9, pp. 37-39, Sept. 1997.

[14] J. Oplinger, D. Heine and M. Lam, “In Search of Speculative Thread-Level Parallelism”, *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 303-313, 1999.

[15] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, “Trace Processors”, in *Proc. of the 30th. Int. Symp. on Microarchitecture*, pp. 138-148, 1997.

[16] Y. Sazeides and J.E. Smith, “Implementations of Context-Based Value Predictors”, Technical Report #ECE-TR-97-8, University of Wisconsin-Madison, 1997.

[17] G. Sohi, S.E. Breach and T.N. Vijaykumar, “Multiscalar Processors”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 414-425,1995.

[18] J. Steffan and T. Mowry, “The Potential of Using Thread-Level Data Speculation to Facilitate Automatic Parallelization”, in *Proc. 4th Int. Symp. on High-Performance Computer Architecture*, pp. 2-13, 1998

[19] J.Y. Tsai and P-C. Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation”, in *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35-46, 1996.

[20] D.M. Tullsen, S.J. Eggers and H.M. Levy. “Simultaneous Multithreading: Maximizing On-chip Parallelism”, in *Proc of the 22nd Int. Symp. on Computer Architecture*, 1995.

[21] T.N. Vijaykumar, “Compiling for the Multiscalar Architecture”, Ph. D. Thesis, University of Wisconsin-Madison, 1998.

[22] D.W. Wall, “Limits of Instruction-Level Parallelism”, *Tech. Report WRL 93/6*, Digital Western Research Laboratory, 1993.