5

OTHER CRITICAL ISSUES ON SPECULATIVE MULTITHREADED PROCESSORS

In this Chapter, the impact on the performance of speculative multithreaded processors of several microarchitectural parts such as branch prediction, value prediction and initialization overhead are analyzed.

5.1. INTRODUCTION

In previous chapters, interthread data dependences and the spawning scheme have been presented as the main factors that could affect to the performance of speculative multithreaded processors. However, there are some other limiting factors that have not been studied yet, such as branch prediction and the initialization overhead.

In this Chapter, the impact of implementable branch predictors and a realistic initialization overhead is analyzed. Also, the impact of value prediction on the performance of the different spawning schemes is also investigated.

This Chapter is organized as follows: in section 5.2 the experimental framework is described. Branch prediction is studied in section 5.3. In section 5.4, different value predictors are investigated. The impact of initialization overhead is studied in section 5.5 and in Section 5.6 the performance of a 4-thread unit clustered processor is analyzed. Finally, the main conclusions of the Chapter are summarized in section 5.7.

5.2. EXPERIMENTAL FRAMEWORK

For the experiments in this Chapter, we will consider a fully-interconnected Clustered Speculative Multithreaded Processor as it was presented in Chapter 2. This microarchitecture is made up of several thread units, each one being similar to a superscalar out-of-order processor core. Each thread unit has its own physical register file, register map table, instruction queue, functional units, local memory and reorder buffer in order to execute multiple instructions out-of-order.

The baseline speculative multithreaded processor has 16 thread units 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), 2 memory address computation (1), 1 integer multiplication (4), 2 simple FP (4), 1 FP multiplication (6), and 1 FP division (17).
- Reorder buffer: 128 entries.
- Local branch predictor: 14-bit gshare[40].
- 32 KB non-blocking, 2-way set-associative 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. Memory depen-

dence violations are detected by means of a cache coherence protocol based on the Speculative Versioning Cache.

Performance is by default reported as the speed-up over a single-threaded execution. Any thread is allowed to spawn any speculative thread, that is, the spawning model assumed is the unrestricted spawning model.

Statistics in this Chapter are reported for the two families of spawning schemes: the combination of heuristics which spawns threads at loop iterations, loop continuations and subroutine continuations and the profile-based spawning scheme. The configuration of this profile-based policy is a minimum thread size spawning scheme combined with the cancellation scheme that removes spawning pairs after being executed 50 cycles alone (except for `compress` that is 200 cycles) and no reassign scheme.

5.3. BRANCH PREDICTION

Branches in the speculative multithreaded execution model are fetched out of the program order. Therefore, conventional branch predictors that make their predictions depending on the previous outcomes of the same and adjacent branches may be affected ([40][67] among others). In Chapter 2, two implementations of branch predictors were presented. Both consist of a local branch predictor for each thread unit and they work completely independent of the other predictors once a thread is started. The difference between them lays on the way the branch prediction tables are initialized. In the first one, that we refer to as *copying mechanism*, when a thread is initialized, its branch prediction table is copied from the table of the parent thread. Alternatively, a speculative thread may simply inherit the prediction table as it was left by the previous thread executed in that unit. We refer to this latter mechanism as *non-copying mechanism*.

Obviously, the initialization cost of both mechanisms is significantly different. Whereas for the copying mechanism, all the branch prediction has to be copied from the parent thread to the spawned thread, for the non-copying mechanism, no initialization is required. Moreover, the cost of copying the branch prediction table may be important on clustered architectures since the contents have to travel through the interconnection network.

Figure 5.1 compares the branch prediction accuracy of branch predictors initialized from the parent at thread creation and that of a non-initialization policy, considering the fully-interconnected Clustered Speculative Multithreaded Processor with 16 thread units and for the perfect register prediction configuration. In addition, it also shows the prediction accuracy of a centralized predictor that processes all branches in

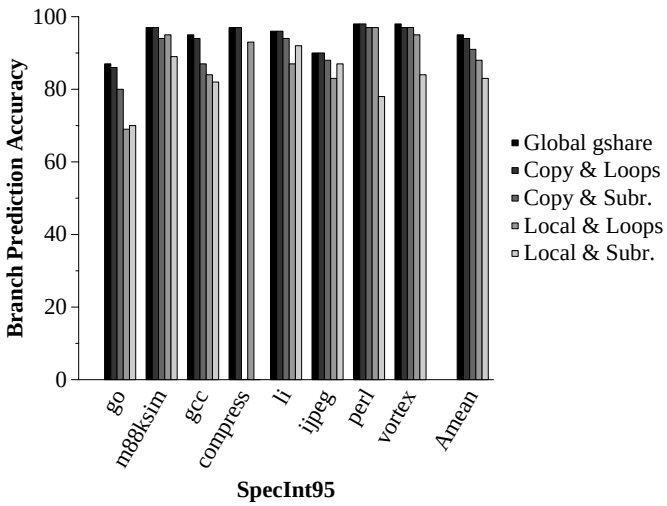

Figure 5.1. Branch prediction accuracy.

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).

On the other hand, Figure 5.2 shows the impact of 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.

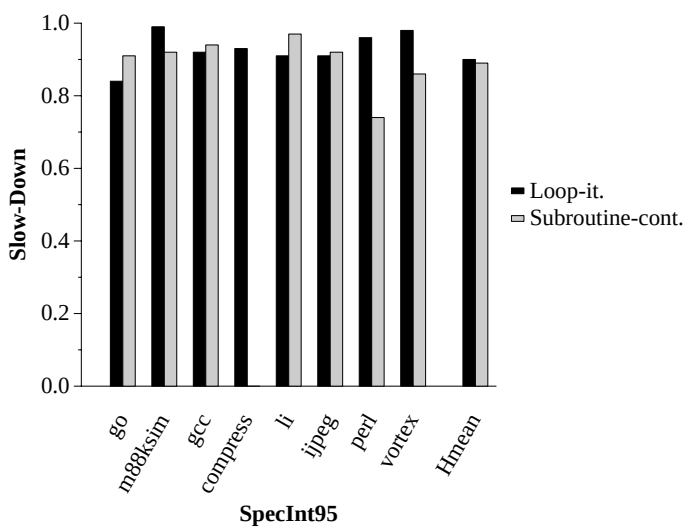

Figure 5.2. Slow-down when independent local branch predictors are used.

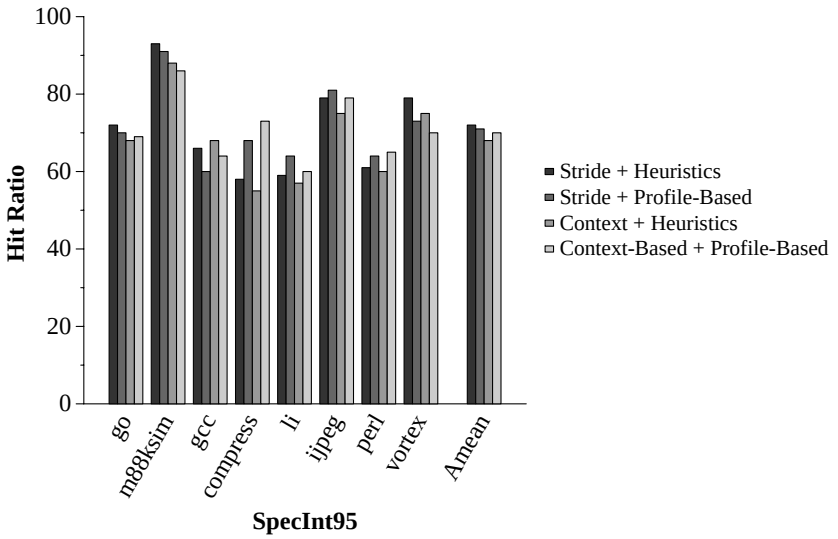

Figure 5.3. Value prediction accuracy.

For the profile-based spawning scheme, the degradation suffered is quite similar to the one obtained by the loop-iteration scheme with the same configuration (7% loss in prediction accuracy). This degradation results in less than 10% slow-down with respect the mechanism that copies the branch prediction tables.

For the next experiments, a non-copying mechanism is considered for the branch predictor.

5.4. VALUE PREDICTION

The importance of value prediction for speculative multithreaded processors has been previously shown in Chapter 3. In this Chapter, the relationship between the performance of the value prediction with the efficiency of the thread spawning scheme is demonstrated.

For the next experiments, the size of the value predictor has been fixed to 16KB for the two value predictors analyzed: the stride[18][64] and the context-based (FCM) value predictors[65]. Prediction tables are indexed by hashing 3 values, the program counter of both the spawning point and the control quasi-independent point and the identifier of the register being predicted.

Figure 5.3 shows the prediction accuracy for a 16KB value predictor (stride and context-based FCM value predictors) for the combined spawning scheme and the profile based spawning scheme. It can be observed that there are no significant differences between the spawning scheme and the accuracy of the predictor. Thus, the prediction accuracy for the stride value predictor is about 72% for the combination of heuristics and slightly lower for the profile-based scheme whereas the context-based value predictor performs the best for the profile-based spawning scheme. In any case, the prediction accuracy for both value

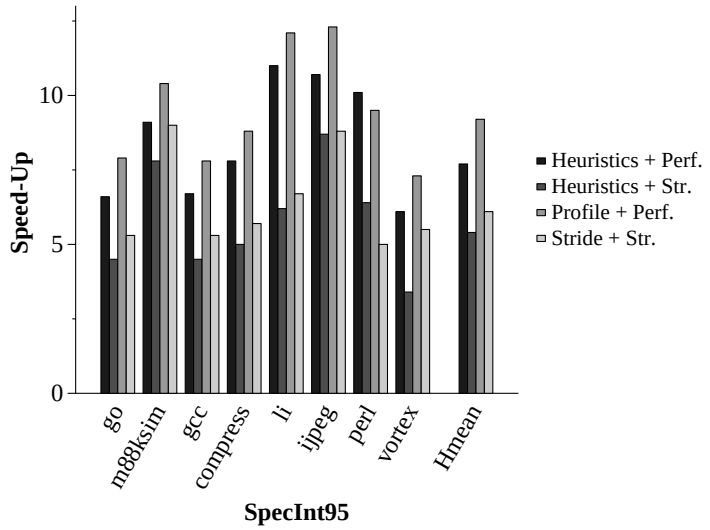

Figure 5.4. Speed-up with a perfect and a realistic value predictor.

predictors is very similar. It is also remarkable that `m88ksim` achieves a prediction accuracy close to 90% with the stride predictor.

Figure 5.4 shows the speed-up over a single threaded execution for the combined and the profile-based spawning schemes with a 16-KB stride value predictor. For comparison, the speed-ups achieved by both spawning schemes with perfect value prediction are also depicted in the figure. It can be observed there a significant slow-down compared with the speed-up achieved with a perfect register value predictor, about 28% for the combined scheme and 35% for the profile-based scheme. Nevertheless, the performance presented by this architecture is still quite impressive obtaining an average speed- up close to 5 for the combined scheme and close to 6 for the profile-based and the difference among both spawning policies is maintained close to 20%.

5.4.1. Data-Dependence Aware Profile-Based Spawning Schemes

In order to avoid this important difference between the performance obtained for a perfect and a realistic value predictor alternative criteria to choose among the different control quasi-independent points for a given spawning point may be considered. Instead of choosing the control quasi-independent point that results in the largest sized thread, we have evaluated a scheme that selects the control quasi-independent point that tries to maximize the number of independent instructions between the spawner and the spawned thread. More concretely, the scheme chooses the one that maximizes the number of independent instructions beyond the control quasi-independent point relative to the number of instructions executed between

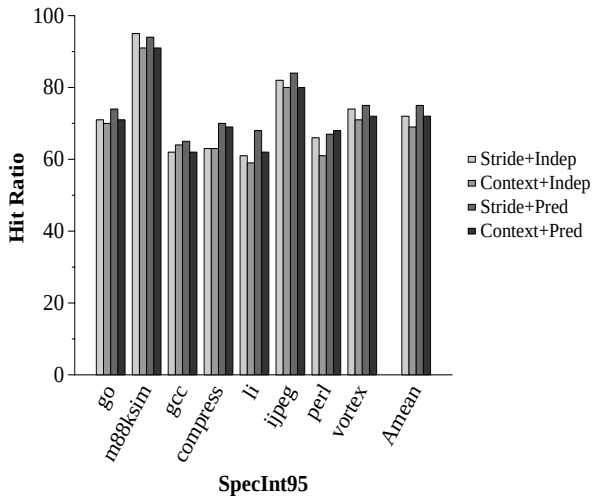

Figure 5.5. Value prediction accuracy for the independent and the predictable profile-based spawning policies.

the spawning and the control quasi independent point. We have referred to these spawning policy the *independent profile-based spawning scheme*.

Besides, we have also considered a third scheme that selects the control quasi-independent point that maximizes the number of instructions either predictable or independent between the spawner and the spawn-new thread. We refer to this scheme as the *predictable profile-based spawning scheme*. For the study of the predictability, we have considered the stride predictor since it provides the best value prediction accuracy for the assumed predictor size[47].

Figure 5.5 shows the prediction accuracy achieved by the value predictors when these new policies are applied. The first two bars correspond to the independent policy and the last two bars to the predictable profile-based spawning policy. As expected, the policy oriented to predict values achieves a better value prediction hit ratio. It correctly predicts 75% of the live-in register values with a stride value predictor and 73% of the those values with a context-based. The independent profile-based spawning policy achieves also better results than the previous spawning scheme obtaining a 73% for the stride value predictor and 71% for the context-based.

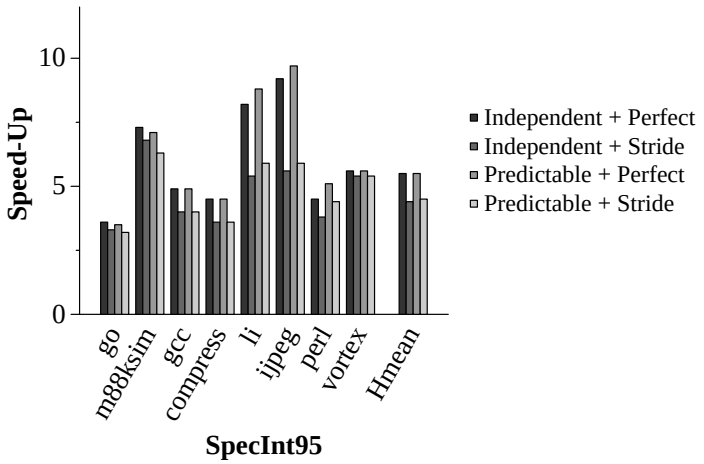

Figure 5.6. Speed-up obtained by the independent and the predictable profile-based spawning scheme with perfect and a realistic value predictor.

Nevertheless, better value predictor accuracy does not imply better overall performance. In figure 5.6 it can be observed that for a stride predictor, the speed-ups achieved by these two new spawning policies is about 4.4 for the independent and 4.5 for the predictable, which is approximately 35% lower than the one obtained by the scheme that maximizes the distance between the spawning and the control quasi-independent point. However, the loss in performance for these profile-based spawning policies with perfect value prediction is about 22% for the independent spawning policy and 18% for the predictable. Also, it can be observed that some benchmarks like `gcc`, `compress` and `vortex` present the same results for both spawning schemes. This is due to the fact that the predictable and the independent profile-based spawning schemes produces very similar set of spawning pairs.

The worse results produced by both spawning policies are due to the more restrictive approach used for selecting spawning pairs that results in a much lower coverage than the one that maximizes the distance between the spawning and the control quasi-independent point.

The best spawning policy would possibly be one that looks for a trade-off among the thread size, the independence of the instructions executed on the speculative threads and the predictability of the dependent values that flow from one thread to the others instead of considering each of these elements independently.

5.5. Initialization Overhead

Starting a new speculative thread in a thread unit requires several operations that may take some non-negligible time as it was detailed in chapter 2. In addition, registers that are live at the beginning of a thread

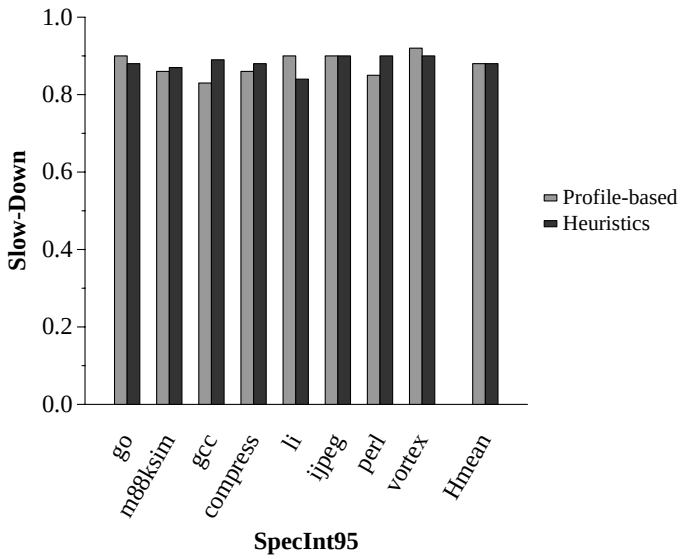

Figure 5.7. Slow-down for an 8-cycle initialization overhead.

must be initialized with their predicted values. On the other hand, 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. Note that several registers per cycle can be read/written in a multi-port register file, and several values can be forwarded in parallel depending on the bandwidth of the interconnection network.

Figure 5.7 shows the impact of an 8-cycle initialization overhead for both spawning policies. 8 cycle may be a reasonable penalty for copying 32 integer and 32 FP registers from the parent to the spawned thread. The baseline configuration for this figure includes the 16-KB stride register value predictor. It can be observed that the slow-down due to this overhead is 12% on average for both spawning schemes and it ranges from 16% to 8% for all the benchmarks.

5.6. 4-THREAD UNIT CONFIGURATION

Finally, the performance of a clustered speculative multithreaded processor with only 4 thread units has been evaluated. The main advantage of reducing the number of thread units is the reduction of the communication latency. For instance, for fully interconnected clustered processors, the design of a crossbar is simpler and the latency for traverse it is lower than for 16 thread units. Besides, it also reduces some requirements: i.e., the number of registers needed for supporting 4 contexts in a centralized processor is much lower than for 16 and the access time to the register file will be also decreased.

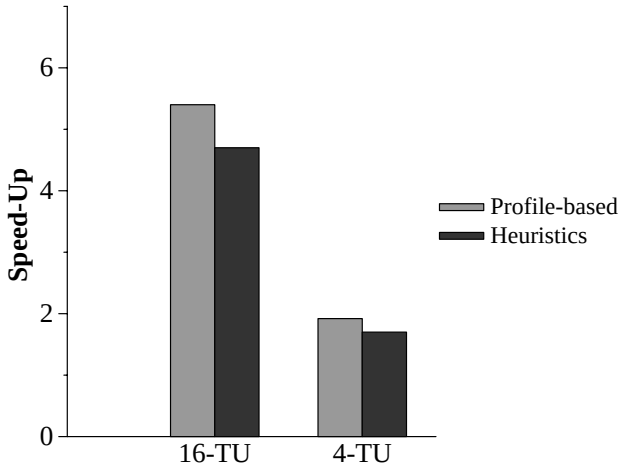

Figure 5.8. Average speed-ups for a 4-Thread Unit clustered processor.

Figure 5.8 shows the average speed-up over a single-threaded execution of a 4-thread unit clustered processor. Each thread unit has the same characteristics than for the 16-thread unit configuration. The baseline configuration also includes a 16-KB stride register value predictor and an 8-cycle initialization overhead is considered. For comparison, the speed-up achieved by the 16-thread unit clustered processor is also depicted in this figure. It can be observed than the performance for such configuration is still quite high. Thus, the profile-based spawning scheme almost obtains an speed-up of 2 and the combined one a 70% improvement.

5.7. CONCLUSIONS

The main conclusion of this Chapter is that the speculative multithreaded execution model still reports impressive speed-ups even when it is evaluated under realistic conditions. Thus, a clustered speculative multithreaded processor with 16 thread units, a 16-KB stride value predictor, local branch predictors that are not initialized at thread creation and an 8-cycle initialization overhead still results in an average speed-up of 5.4 for the profile-based spawning scheme and 4.7 for the combination of the loop-iteration, loop-continuation and the subroutine-continuation spawning schemes for the SpecInt95 benchmark set. For a 4-thread unit configuration, the speed-up is almost 2 for the profile-based scheme and 1.7 for the heuristic-based one.

The most important degradation of the performance compared with the ideal scenario comes from value prediction. The slow-down of a 16-KB stride value predictor with respect to perfect prediction is 35% for the profile-based scheme and almost 30% for the combination of heuristics. In this Chapter, two new profile-based spawning schemes have been also presented. These spawning schemes are focused on

reducing the impact of realistic value prediction on the performance of the processor. To do that, the selection of the spawning pairs is aware of the independence and the predictability of the spawned threads. The degradation due to realistic prediction by such spawning schemes is much lower than the reported for the spawning policy that maximizes the distance between the spawning and the control quasi-independent point. However, the performance of these alternative spawning schemes is also smaller than the one that maximizes the distance due to the low coverage of both spawning schemes.