# Thread-Spawning Schemes for Speculative Multithreading

Pedro Marcuello and Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain
{pmarcue,antonio}@ac.upc.es

**Abstract**

*Speculative multithreading has been recently proposed to boost performance by means of exploiting thread-level parallelism in applications difficult to parallelize. The performance of these processors heavily depends on the partitioning policy used to split the program into threads. Previous work uses heuristics to spawn speculative threads based on easily-detectable program constructs such as loops or subroutines. In this work we propose a profile-based mechanism to divide programs into threads by searching for those parts of the code that have certain features that could benefit from potential thread-level parallelism.*

*Our profile-based spawning scheme is evaluated on a Clustered Speculative Multithreaded Processor and results show large performance benefits. When the proposed spawning scheme is compared with traditional heuristics, we outperform them by almost 20%. When a realistic value predictor and a 8-cycle thread initialization penalty is considered, the performance difference between them is maintained. The speed-up over a single thread execution is higher than 5x for a 16-thread-unit processor and close to 2x for a 4-thread-unit processor.*

## 1. Introduction

Speculation is a well-known technique used to improve processor performance. These mechanisms have been widely used in order to reduce the penalties of both control and data dependences. Also, diminishing returns of instruction-level parallelism are boosting the use of alternative techniques to increase performance. Combining thread-level parallelism and instruction-level parallelism is an approach that has been considered by several processor vendors. These types of processors are usually referred to as multithreaded processors. The task of dividing programs into threads that will be executed in parallel is rather straightforward for regular or numeric applications, and the current compiler technology can perform it efficiently. However, this task becomes hard for irregular and non-numerical programs; compilers usually fail to discover the potential thread-level parallelism that could be effectively exploited in this class of applications.

Speculative multithreading is a promising approach to solving this problem. In these systems, threads that may be control and data dependent on previous threads are speculatively spawned and executed. Relaxing the constraints to spawn a thread results in a significant increase of opportunities to exploit thread-level parallelism [2][15][16], even though, obviously, roll-back mechanisms are needed in case of misspeculations.

The performance of speculative multithreaded architectures is very sensitive to the policies that determine which parts of the code are executed by speculative threads and when they start execution. We refer to this criteria as the *thread-spawning policy*. In several architectures such as Multiscalar[5][21], the SPSM architecture[4] and the Superthreaded[24], the compiler is responsible for dividing the program into speculative threads. Alternatively, the Dynamic Multithreaded Processor[1] and the Clustered Speculative Multithreaded Processor[12][13] rely only on hardware techniques; programs are partitioned at run-time. The thread-spawning policies proposed so far for speculative multithreaded architectures are very simple. They are based on assigning speculative threads to common program constructs such as loop iterations, loop continuations and subroutine continuations.

A thread-spawning operation is identified by two instructions: 1) the spawning instruction that creates a new thread when it is reached, and 2) the spawned instruction where the speculative thread starts its execution. These instructions are referred to the spawning point and the control quasi-independent point, respectively.

In this paper we propose a general framework for identifying effective spawning and control quasi-independent points in any sequential program. Thus, a profile-based analysis is done in order to find the best sections of the code to create speculative threads depending on several requirements that they should match. This approach tries to identify pairs of spawning and control quasi-independent points in such a way that the execution of

Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)
1503-0897/02 $17.00 © 2002 IEEE

the control quasi-independent point is very likely once the spawning point is reached. Most of the instructions below the control quasi-independent point should be independent of the instructions between the spawning point and the control quasi-independent point. We show that this thread-spawning policy is more effective than previous schemes, which they are based on just exploiting common program constructs.

The rest of the paper is organized as follows. Section 2 reviews the related work. The profile analysis is described in Section 3. Section 4 shows the performance potential of the profile-based spawning scheme and evaluates it under realistic assumptions, and finally Section 5 summarizes the main conclusions of the work.

## 2. Related Work

Several microarchitectural proposals for exploiting speculative thread-level parallelism have been recently proposed. The Expandable Split Window Paradigm [5] and its follow-up work, the Multiscalar Processor [21] were the pioneer works on this topic. In that architecture, the compiler is responsible for dividing the code into tasks. The policy used by the compiler is based on heuristics that try to minimize the data dependences among active threads or maximize the workload balance, among other compiler criteria [25].

In some other proposals such as the SPSM [4] and the Superthreaded [24] architectures, the compiler is also responsible for splitting the program into threads. But in both cases, threads are assumed to be loop iterations instead of the more complex analysis performed by 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 assistance. The Speculative Multithreaded Processor [12] and the Clustered Speculative Multithreaded Processor [13] identify loops at run-time and simultaneously execute iterations in different thread units.

In the same way, the Dynamic Multithreaded Processor [1] relies only on hardware mechanisms to divide a sequential program into threads. In this case, it speculates on loop and subroutine continuations instead of loop iterations. Moreover, the architectural design of the processor allows for out-of-order thread creation which requires communication among all hardware contexts.

Trace Processors [17] also exploit certain kinds of speculative thread-level parallelism. The mechanism to partition a sequential program into almost fixed-length traces is specially suited to maximize the workload balance among the different thread units with the help of the trace cache [18].

A different proposal to divide the program into threads was done by Codrescu et al. The MEM-slicing [2] scheme is also based on profile analysis, but the spawning algorithm starts new threads at memory instructions.

Several other recent techniques are also based on identifying dynamic sequences of instructions that could potentially have a high impact on performance and assigns them to a speculative thread [3][11][26]. However, this thread identification technique relies on simple heuristics (e.g.; mispredicted branches and load misses are the most critical instructions).

In addition, several works on speculative thread-level parallelism on multiprocessor platforms have been performed [8][9][10][23]. In all cases, programs are split by the compiler using heuristics based on loop iterations or subroutine continuations.

Some works comparing different spawning policies have been performed for both an on-chip multiprocessor [2][16] and a Clustered Speculative Multithreaded Processor [15]. The spawning policies considered were based on assigning speculative threads to loop iterations, loop continuations and subroutine continuations. The reported results of these two works cannot be compared since the baseline architectures were totally different and in the case of an on-chip multiprocessor, the present interactions between fine and coarse-grain parallelism were not considered. Subroutine continuation shows the best thread spawning potential for an on-chip single-issue in-order multiprocessor, whereas for the Speculative Multithreaded Processor spawning at loop iterations is the most effective scheme.

The importance of value prediction in such architectures has been pointed out elsewhere [14][15]. Predicting thread input values allows the processor to execute speculative threads as if they were independent. In some previous proposals (e.g., the SPSM [4] or the Superthreaded [24] architecture) no mechanisms for value prediction was considered. This implies that for each inter-thread dependent pairs of instructions the consumer must wait until the producer has been executed. On the other hand, the Dynamic Multithreaded [1] processor uses a very simple value prediction scheme; the register values of the spawned thread are predicted to be the same as those of the spawning thread at spawn time. More effective prediction schemes are proposed in the Trace Processors [17] and the Clustered Speculative Multithreaded architecture [13].

## 3. Speculative Thread-Level Parallelism

A thread spawning operation is identified by two instructions in the dynamic instruction stream that we refer to as the spawning and the control quasi-independent points. Each pair of points is referred to as a *spawning pair*. The spawning point is the instruction that, when reached by

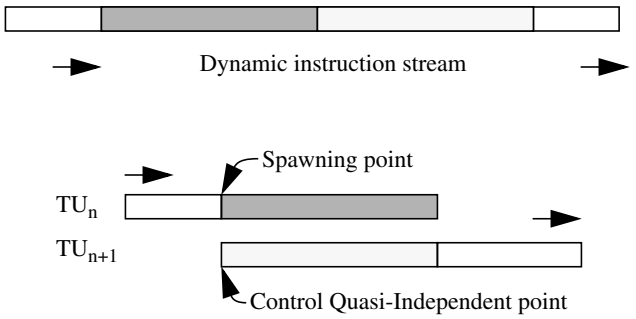


**Figure 1:** Spawning and Control Quasi-Independent Points.

the processor, it fires the creation of a new thread. The control quasi-independent point is where the new spawned thread starts. The spawning and the control quasi-independent points can be conventional instructions in the instruction set, denoted with special marks or special instructions such as fork and spawn. Figure 1 shows how speculative multithreaded processors work. Thread Unit n executes the instruction stream in the same way as a conventional superscalar processor until it reaches a spawning point. At this point, the processor identifies a future instruction (the control quasi-independent point) which will very likely be executed in the near future. Then, Thread Unit n+1 spawns a new thread speculatively starting at the control quasi-independent point while Thread Unit n continues executing instructions up to the control quasi-independent point which also becomes the join point between threads. That is, the join point of a thread is a control quasi-independent point of an on-going speculative thread.

It is obvious that the best instructions to be considered as spawning and control quasi-independent points are those where the speculative and the non-speculative thread were control and data independent, in such a way that the processor would be able to execute them concurrently. Unfortunately, these kind of threads are not common in some programs, especially in non-numerical codes, and thus, their potential thread-level parallelism may be rather low.

Effective spawning pairs should satisfy some requirements. First, the probability of reaching the control quasi-independent point after visiting the spawning point should be very high in order to conserve resources (executing instructions that will never be reached). Second, the distance[1] between the spawning point and the control quasi-independent point should not be too small or too large to keep the thread size within a certain limit. Small threads result in too much overhead and large threads may result in work imbalance. Third, instructions after the control quasi-independent point should have few dependences with instructions above it or alternatively, the values that flow through such dependences should be predictable.

Previous thread-spawning policies basically focused on the first criterion:

- Loop iterations: considers the first instruction in static order of a loop (the target of a backward branch) as both the spawning and the control quasi-independent point. Note that once an iteration is started, a further iteration is very likely regardless of the outcome of the branches inside the loop body.
- Loop continuation: considers the first instruction in static order of a loop as the spawning point and the instruction following the backward branch in static order that closes the loop as the control quasi-independent point. Note that after starting a loop, the instruction at the control quasi-independent point is very likely to be executed regardless of the control-flow inside the loop.
- Subroutine continuation: considers a subroutine call as the spawning point and the instruction following the subroutine call in static order (e.g., the point where the subroutine will return) as the control quasi-independent point. Note again that after the call, this latter instruction is very likely to be executed regardless of the path followed inside the call.

### 3.1. Profile-Based Thread-Spawning Scheme

We are interested in identifying speculative threads that meet the three criteria discussed above. Threads are not necessarily associated with a particular program construct (e.g. loop iteration) and any instruction can be a spawning point or a control quasi-independent point.

The technique proposed here to identify spawning pairs is based on a profile-based analysis of the properties of any potential section of code. For this purpose, a dynamic control flow graph of the program is built. Each node of the graph represents a basic block and edges represent possible control flows among blocks. Edges are weighted with the frequency that the corresponding control flow has been followed during the profiling. Besides, to reduce the size of the graph, we prune the least frequently executed basic blocks. Thus, basic blocks are ordered by execution count and they are chosen from highest to lowest count until 90% of the total executed instructions are covered. However, in order not to lose information about possible control flows, whenever a node is pruned, any edge from a predecessor to it is transformed to a series of edges from that predecessor to their successors, and any edge from it to a successor is transformed to a series of edges from every predecessor to

[1] We refer to distance as the average number of instructions executed between the spawning point and the control quasi-independent point.


1503-0897/02 $17.00 © 2002 **IEEE**

that successor. During this transformation, if an edge is transformed into multiple edges, its original weight is proportionally split across the new edges.

Once the reduced control flow graph is generated, the probability to reach any basic block after executing any other else is computed. We will refer to these probabilities as *reaching probabilities*. These probabilities are stored in a two-dimensional square matrix that has as many rows and columns as nodes in the control flow graph. Each element of the matrix represents the probability to execute the basic block represented by the column after executing the basic block represented by the row. This probability is computed as the sum of the different frequencies for all the different sequences of basic blocks that exist from the source node to the destination node. In order to simplify the computation, the only constraint taken into account for these sequences is that the source and the destination nodes can only appear once in the sequence of nodes as the first and the last nodes respectively (obviously, when the reaching probability of execute a basic block after executing itself is computed, that basic block appears twice, the former as source node and the later as destination node). Anyway, this constraint does not restrict any other basic block from appearing more than once in the sequence.

Once all these probabilities are computed, pairs of nodes are evaluated to become spawning and control quasi-independent points (in fact, the spawning and the control quasi-independent point will be the first instructions of the basic blocks selected). Thus, the previous constraint, in addition to simplifying the computation of the reaching probability matrix, it reduces the control logic of the processor since otherwise, the identification of the starting and ending points of each thread would become quite cumbersome.

Then, a prune of those pairs of nodes that do not accomplish the minimum requirements to be considered as good candidates to spawning pairs must be done. Those requirements were mentioned in the previous subsection. The first property that must be satisfied by each of these pairs is that their associated reaching probability should be very high, i.e. higher than a given threshold; that is, the probability to reach the control quasi-independent point after the spawning point is higher than a given threshold.

The second requirement that the spawning pairs must satisfy is that a minimum average number of instructions between the spawning point and the control quasi-independent point should exist in order to reduce the relative overhead of thread creation. Consequently, while the reaching probability is being computed, additional calculation regarding the average number of instructions between the source node and the destination node is performed. The average is calculated as the sum of the number of instructions executed by each sequence of basic blocks multiplied by their frequency. Thus, good candidates

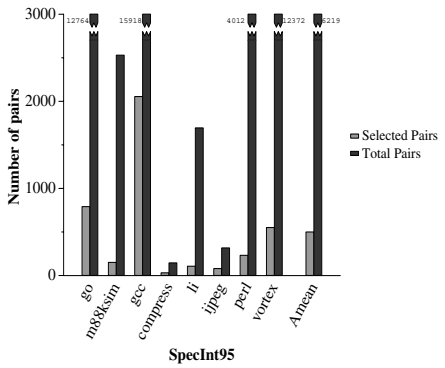


**Figure 2:** Number of pairs of basic blocks selected and number of selected pairs that have different spawning points.

to be considered as spawning pairs are those pair of basic blocks whose reaching probability is higher than a threshold and the average number of instructions between them is larger than a minimum size.

It is possible that for a given spawning point, there are several good candidates for its associated control quasi-independent point (i.e. for a given row of the probability matrix, there are more than one element that exceeds the minimum probability and the minimum size). Figure 2 shows the total number of pairs of basic blocks obtained for the SpecInt95 benchmarks, which is on average 6218, but only 499 have different spawning points. The minimum distance between the spawning point and the control quasi-independent point considered is 32 instructions and the minimum probability is 0.95.

When the processor reaches the spawning point it will start a speculative thread at only one control quasi-independent point. Thus, the alternative quasi control-independent points associated to each spawning point must be ordered according to the expected benefit. Three alternative criteria have been considered to produce such an ordering: a) maximizing the distance between the spawning and the control quasi-independent point - this is an estimation of the size of the corresponding speculative thread if we assume that the spawning thread and the spawned thread have the same instruction throughput -; b) the number of instructions of the spawned thread that are independent of previous instructions - again assuming a size of the thread equal to the distance between the spawning and the control quasi-independent point -; and c) maximizing the number of instructions of the spawned thread that are either independent of or dependent on a predictable value generated outside the thread. We initially consider the first criterion and we later present results for the other two.

Finally, all return point pairs (pairs of subroutine calls and the return point) are added to the list of spawning and control quasi-independent points if they satisfy the minimum size constraint, since some of them may not have

been selected by the previous algorithm. In particular, if a subroutine is called from multiple locations, it will have multiple predecessors and multiple successors in the control-flow graph. If all the calls are executed a similar number of times, the reaching probability of any pair call and return point will be low since the graph will have multiple paths with similar weights.

## 4. Performance Evaluation

In this section we evaluate the performance of the proposed thread-spawning policy and compare it with previous proposals. For this evaluation we assume a particular speculative multithreaded processor, namely the Clustered Speculative Multithreaded Processor [12][13], but most of the results can be extrapolated to other architectures. We just focus on irregular applications (SpecInt95) for which compilers typically fail to exploit thread-level parallelism.

### 4.1. Experimental Framework

A Clustered Speculative Multithreaded Processor is made up of several thread units, each one being similar to a superscalar processor core. Communications among clusters occur for both memory and register values and a fully-interconnected topology is considered. Further details of the processor can be obtained elsewhere [12][13].

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[22]. For the statistics, we have executed each to completion using training input data.

The baseline speculative multithreaded processor has a parameterized number of thread units (from 4 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
- Physical Registers: 64.
- Functional Units (latency in brackets): 2 simple integer (1), 2 load/store units (1 for address calculation plus the latency of accessing the cache), 1 integer multiplication (4), 2 simple FP (4), 1 FP multiplication (6), and 1 FP division (17).
- Reorder buffer: 64 entries.
- Local branch predictors: 10-bit gshare. Local predictors are not initialized when a new thread is spawned at a thread unit; instead, it will use the previous contents of such tables.

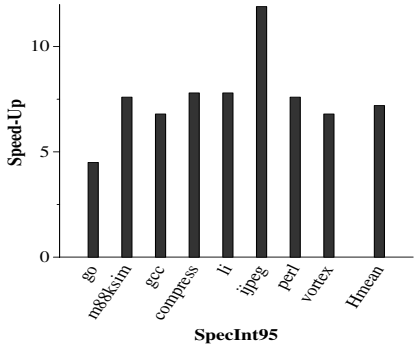


**Figure 3:** Speed-up over a single-threaded execution using the profile-based spawning scheme with 16 Thread Units and perfect value prediction.

- 32 KB non-blocking, 2-way set-associative, local, first-level data cache with an 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.

Initially, we assume a perfect register value prediction (i.e., input register values are available when the speculative thread is started) and no thread creation overhead. Later, the impact of a realistic register value predictor is analyzed as well the impact of considering a realistic thread creation penalty.

Dependent values through memory locations are not predicted. The cost for forwarding data values from the producer thread unit to the consumer has been set to be 3 cycles. Memory dependence violations are detected by means of a cache coherence protocol based on the Speculative Versioning Cache [7].

In figures 3 to 12, spawning pairs are obtained from the profile-based analysis discussed in the previous section being the minimum reaching probability 0.95 and the minimum distance 32 instructions.

### 4.2. Performance Figures

Figure 3 shows the speed-up obtained by a Clustered Speculative Multithreaded Processor with 16 thread units over a single-threaded execution. We are using our profile-based spawning policy and assume a perfect value predictor for inter-thread register dependences. The average speed-up is 7.2 (harmonic mean) and it is quite important for all benchmarks. This shows the effectiveness of the proposed scheme for exploiting thread-level parallelism in irregular applications. For some programs such as `ijpeg`, which is the most regular program in the set, the speed-up reaches 11.9.

Figure 4 shows the average number of active threads for each program. As it can be expected the average number of active threads is closely related to the speed-up. On average, the average number of active threads is 7.5 and for

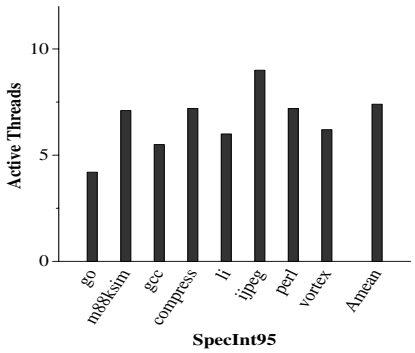


**Figure 4:** Average number of active threads

the program with the highest speed-up, ijpeg, it is 9.0. The additional speed-up achieved to that produced by thread-level parallelism is due to value prediction. Even though the effectiveness of our profile-based spawning policy is quite high, there are still about half of the processor resources that are wasted on average. This may be due to the application´s features but also to some limitations in the spawning policy. We show below how some of these limitations can be overridden.

Since threads must commit in program order, thread units become available in the same order and thus, workload balancing may be a critical issue for performance. Threads that are being executed for long periods of time alone, or in parallel with very few other threads while the other thread units have finished the execution and are waiting for the completion of such threads to commit their respective threads, are undesirable. Thus, we extend the spawning scheme with a dynamic mechanism that monitors how much time a thread is executing alone. If it is above a certain threshold, the corresponding spawning pair is removed so that this thread is not created in the future[1]. This removal of spawning pairs can be done either the first time the above situation is observed or after the above situation has been repeated for a number of times. Figure 5a shows the performance when spawning pairs are never removed, when they are removed after executing 50 cycles alone, or when they are removed after executing 200 cycles alone. It can be observed that in general, the most aggressive spawning removal policy results in significant improvement, except for `compress`, whose performance dramatically drops when a small number of cycles is considered. This is due to the small number of selected spawning pairs in this program (only 30), when left to an aggressive removal mechanism leaves the program with too few spawning pairs. On average, the speed-up achieved for 200 cycles is higher than 8 over a single-threaded execution and represents a 10% improvement compared with the non spawning removal scheme.

[1] We have also evaluated a policy that considers again a removed thread after a certain period of time but we observed very small improvements.

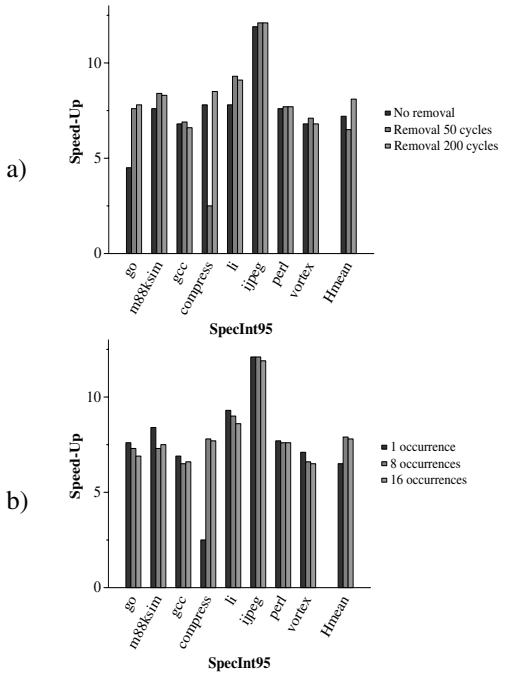


**Figure 5:** Speed-ups achieved by the different spawning pair removal scheme for a) different number of cycles executing alone and b) different number of occurrences before cancelling for the 50-cycle removal scheme.

An alternative way to temper the removal mechanism is to hold off cancelling a spawning pair until the speculative thread that is executing alone a minimum of occurrences. Figure 5b shows the performance for a cancelling policy with 50 cycle alone scheme when the number of occurrences is 8 and 16. On average, delaying the removal decision results in an improvement, but it is basically due to the huge improvement achieved by `compress`. In fact, the rest of the programs suffer a small performance loss. Although not shown in the graphs, we have also evaluated the delayed removal policy for the 200 cycle alone scheme and we have observed a small performance drop for all programs. We have also evaluated a policy that removes a spawning pair whenever the corresponding thread is executing with just a few threads instead of just one. This resulted in a small improvement on average, although most of the benefit came from three programs (`compress`, `m88ksim` and `gcc`).

Figure 2 shows that the number of candidate spawning pairs is much higher than the final number of selected pairs. Remember that only one spawning pair for a given spawning point is considered according to the criteria introduced in section 3.1. Also, whenever a thread reaches a spawning point and finds another more speculative thread already started in that control quasi-independent point, it does not spawn a new thread. An alternative policy may be considered. That says whenever a spawning point is

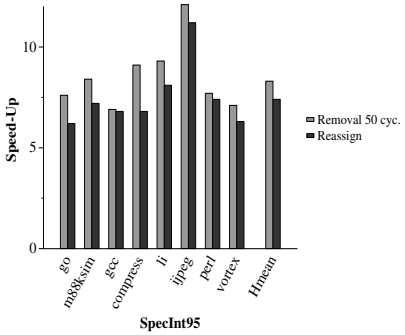


**Figure 6:** Speed-up of the reassign spawning policy compared with the 50-cycle removal policy (for compress, 200 cycles).

reached, if a thread cannot be spawned at the most convenient control quasi-independent point, the next control quasi-independent point is tried according the previously mentioned criteria. Likewise, whenever a spawning pair is removed we may consider the next most convenient pair with the same spawning point. We refer to this policy as the *reassign* spawning policy since it re-assigns a spawning point to a different control quasi-independent point. The result of these modifications are shown in Figure 6, together with the previous policy that just considers a single spawning pair per spawning point. It can be observed that the results are a bit worse for the reassign policy. One reason for this performance degradation is the fact that whenever a control quasi-independent point cannot be chosen, the next control quasi-independent point is usually too close and this results in generating very small threads as well introducing more spawning pairs, and does not necessarily imply better performance.

Figure 7a shows the average thread size performing spawning pair removal and with no reassign.We refer to *thread size* as the number of instructions executed in a thread unit starting when a speculative thread is assigned to this thread unit until it reaches a control quasi-independent point of an on-going thread, that is, a join point. It can be observed that the thread size for most of the benchmarks is smaller than 32, which was the minimum size we considered when selecting a spawning pair. This is due to the overlapped execution of speculative threads. Figure 7b shows the performance when a minimum size for the threads is enforced, in such a way that the spawning pairs whose associated threads are smaller than a minimum size are removed. It can be observed that the speed-up achieved is 10% over the conventional removal policy (a 50-cycle threshold is considered for all the benchmarks, except for `compress`, which is set to 200 cycles).

### 4.2.1. Comparison with traditional heuristics

In a previous study [15], a comparison among basic thread spawning heuristics for a Clustered Speculative

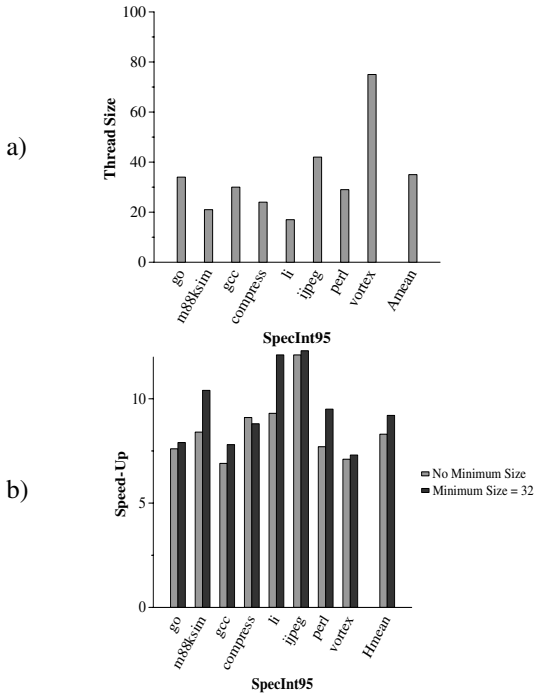


**Figure 7:** a) Average thread size b) Speed-up achieved when a minimum thread size is considered to spawn new speculative threads.

Multithreaded Processor was done. Although the best individual results were reported for the loop-iteration spawning scheme, it was pointed out that the best spawning policy may be a combination of all of them.

In Figure 8, the spawning policy proposed here and a combination of loop-iteration, subroutine-continuation and loop-continuation spawning schemes are compared. Results are reported as speed-ups achieved by the profile-based spawning scheme over the traditional heuristics. It can be observed that on average the improvement is close to 20%, being quite high for `vortex` and more than 10% for the rest of the benchmarks (except for `perl`, which suffers a slight slow-down (8%)). This fact is due to the work imbalance present in this benchmark based on our profile-based spawning policy.

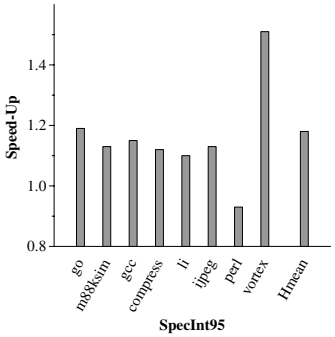


**Figure 8:** Speed-up of the profile-based spawning policy over the traditional heuristics.

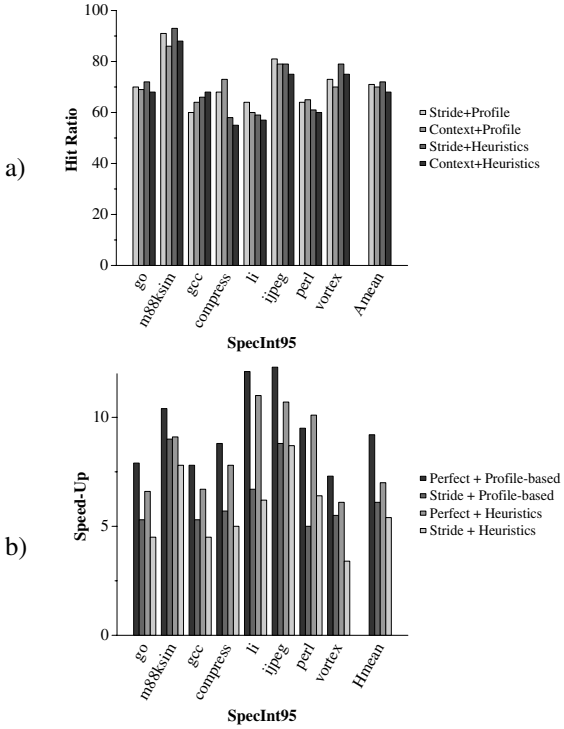


**Figure 9:** a) Value prediction accuracy and b) Speed-ups for the stride value predictor.

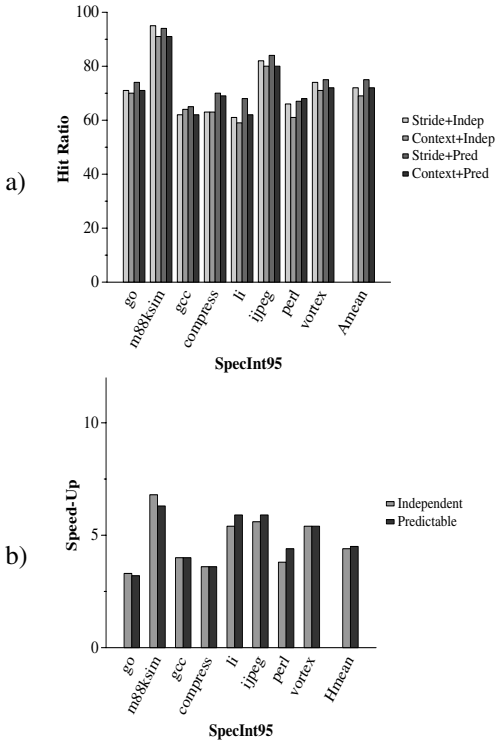


**Figure 10:** a) Value prediction accuracy and b) Speed-up achieved by the independent and predictable spawning policies.

## 4.3. Critical Issues on the Clustered Speculative Multithreaded Processors

Next, the impact of a realistic value predictor, the thread creation overhead and a the number of thread units is studied.

### 4.3.1. Value Prediction

The importance of value prediction for speculative multithreaded architectures has been previously shown [15]. In this subsubsection, we present performance figures that show the impact of different value predictors on the performance of a Clustered Speculative Multithreaded processor with the propose profile-based spawning policy. A study regarding how value predictors work in speculative multithreaded architectures has been presented in [14]. The size of the value predictor has been fixed to 16KB for the two value predictors analyzed: the stride [6][19] and the context-based (FCM) [20] value predictors.

Figure 9a shows the prediction accuracy of the different value predictors for both spawning policies, the profile-based and the heuristic-based schemes. It can be observed that there are no significant differences in prediction accuracy for the different spawning policies and value predictors. On average, the hit ratio is around 70% (note that only thread input values are predicted). 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 9b shows the speed-ups achieved by both spawning policies compared with the single-threaded execution, when a stride predictor is considered. Results for the FCM value predictor are very similar and are not present in the figure. It can be observed that the speed-ups reported are still quite high: the traditional heuristics obtain a speed-up close to 5.5, and the profile-based higher than 6, even though the gap between them has been reduced to only 13%. Moreover, note that the loss in performance when a realistic value predictor is considered is in both cases higher than 25% (30% for the traditional heuristics and 34% of slow-down for our proposed profile-based scheme).

For 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 point that results in the largest sized thread, we have evaluated a scheme that selects the point that maximizes the number of independent instructions between the spawning and the spawned thread. We have also considered a third scheme that selects the control quasi-independent point that maximizes the number of instructions either predictable or independent. We refer to these two new policies as *independent* and *predictable* spawning policies. For this study we have considered the

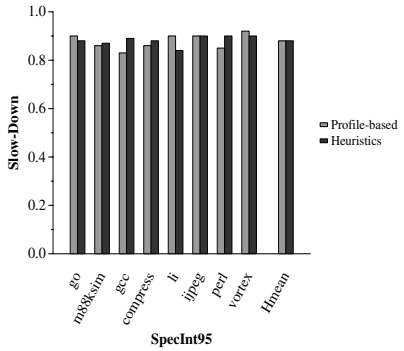


**Figure 11:** Slow-down suffered when an overhead penalty is considered

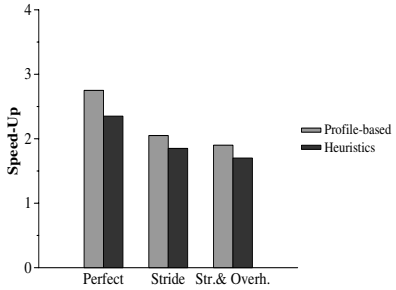


**Figure 12:** Average speed-ups for the spawning policies for 4 Thread Units.

stride predictor since it provides the best value prediction accuracy.

Figure 10a 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 policy. As expected, the policy oriented to predict values achieves the best value prediction hit ratio. It correctly predicts 75% of the values.

Nevertheless, a better value predictor accuracy does not imply a better overall performance. In Figure 10b it can be observed that for a stride predictor, the speed-ups achieved by these two new spawning policies are 35% lower than the obtained by the original one, which maximizes the distance between the spawning and the control quasi-independent point. This is due to the smaller sized threads created by these two spawning schemes. For perfect value prediction (not shown in the figures), the slow-down of the new spawning policies is somewhat lower (21% on average).

#### 4.3.2. Overhead Considerations

Starting a new thread requires several operations that may take some non-negligible time. These operations include the prediction of the *live-in values*[1] for a thread. We refer to the penalty associated with all these operations as initialization overhead. In this section, we evaluate the impact of the initialization overhead for a penalty of 8 cycles, since it is known that the number of live-in values is relatively small. Penalty overhead is only suffered by the new spawned thread. Figure 11 shows the slow-down due to this overhead when a stride value predictor is considered. The slow-down is 12% on average for both cases and it ranges from 16% to 8% for all the benchmarks.

#### 4.3.3. 4 Thread Units

Finally, in order to evaluate the scalability of the architecture with our spawning policy, the performance of the profile-based spawning policy is evaluated for a 4-thread-unit configuration. Figure 12 shows the average speed-ups achieved by both spawning policies for a perfect register value predictor, a stride predictor without overhead penalty and a stride predictor with an initialization overhead of 8 cycles. It can be observed that the speed-up obtained is quite high, 2.75 for perfect value prediction, slightly higher than 2 for a stride predictor without initialization overhead and about 1.9 for a stride predictor with a 8-cycle thread initialization overhead. Note that the degradation in performance between perfect and realistic value predictors is about the same for 4 and 16 thread units.

The bottomline of this study is that a speculative multithreaded processor with a relatively low number of thread units, a simple value predictor and reasonable thread initialization overhead can achieve a significant speed-up for irregular applications such as the SpecInt95. The performance of the scheme scales reasonable well for 16 thread units, where the average speed-up is higher than 5 for a stride predictor and a 8-cycle thread initialization overhead.

## 5. Conclusions

In this work a new approach to spawn speculative threads in a sequential program has been presented. This technique is based on a profile-based analysis to detect which are the best instructions to spawn new threads and where the spawned thread has to start.

We have shown that the potential benefits of this spawning policy are quite high, reporting speed-ups close to 7x. Avoiding the creation of threads that will be executed alone and enforcing a minimum size can increase these speed-ups up to 9.4. The performance achieved by the profile-based spawning policy outperforms the best combination of traditional heuristics such as loop-iteration, loop-continuation and the subroutine-continuation spawning schemes by almost 20%.

When realistic assumptions are considered, the performance obtained is diminished but the results are still quite promising. With a realistic 16-KB stride predictor and an 8-cycle thread creation penalty, the speed-up achieved

[1] Live-in values are those register values that will be read in a speculative thread before they were written and they are produced by a previous thread[14].

by the profile-based scheme is still higher than 5, which is almost 15% better than the obtained by the traditional heuristics.


## Acknowledgements

This work has been supported by grant CICYT TIC 511/98, The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA). The authors would also thank Professor David Kaeli his comments in the realization of the camera ready of this paper.


## 6. References


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

[2] L. Codrescu and D. Wills, "On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm", on Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 40-46, 1999

[3] J. Collins et al., "Speculative Precomputation: Long Range Prefetching of Delinquent Loads", in *Proc. of the 28th. Int. Symp. on Computer Architecture*, 2001.

[4] 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.

[5] 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.

[6] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction", *Technical Report #1080, Technion*, 1996.

[7] S. Gopal, T.N. Vijaykumar, J. E. Smith and G. S. Sohi, "Speculative Versioning Cache", in *Proc. 4th Int. Symp. on High-Performance Computer Architecture*, 1998.

[8] L. Hammond, M. Willey and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor", in *Proc. of Int. Conf. on Architectural Support for Prog. Lang. and Operating Systems,* 1998

[9] G. A. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing", in *Proc. of the Int. Conf. on Parallel Processing*, pp. 239-246, 1996.

[10] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor", in *Proc. of ACM Int. Conf. on Supercomputing,* pp. 85-92, 1998

[11] C. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", in *Proc. of the 28th. Int. Symp. on Computer Architecture*, pp. 40-51, 2001.

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

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

[14] P. Marcuello, J. Tubella and A. González, "Value Prediction for Speculative Multithreaded Architectures", in *Proc. of the 32th. Int. Conf. on Microarchitecture,* pp. 230-236, 1999.

[15] P. Marcuello and A. González, "A Quantitative Assessment of Thread-Level Speculation Techniques", in *Proc. of the 15th. Int. Parallel and Distributed Processing Symposium*, 2000.

[16] 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.

[17] 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.

[18] E. Rotenberg, S. Bennett and J.E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", in *Proc. of 29th Int. Symp. on Microarchitecture*, 1996.

[19] Y. Sazeides, S. Vassiliadis and J.E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing", in *Proc. of the 29th. Int. Symp on Microarchitecture*, Dec. 1996.

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

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

[22] A.Srivastava and A.Eustace,"ATOM: A system for building customized program analysis tools", in *Proc.of the Int. Conf. on Programming languages Design and Implementation*, 1994

[23] 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

[24] 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.

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

[26] C. Zilles and G. Sohi, "Execution-Based Prediction using Speculative Slices", in *Proc. of the 28th. Int. Symp. on Computer Architecture*, pp. 2-13, 2001.