4

THREAD-SPAWNING SCHEMES

In this chapter, the importance of the partitioning mechanisms on the performance of speculative multi-threaded processors is analyzed. Different approaches for splitting programs into threads are studied. These mechanisms identify in which parts of the program speculative thread-level parallelism can be effectively exploited. The selection mechanisms for those speculative threads are referred to as thread-spawning policies. Two families of these policies are studied in this chapter; in one of them, speculative threads are selected based on some heuristics whereas in the other, speculative threads are chosen through a deeper analysis of the properties of the code.

4.1. INTRODUCTION

In Chapter 1, two main requirements to exploit speculative thread-level parallelism were pointed out: hardware support for keeping the speculative contexts and run them in parallel and a partitioning scheme to divide the program into speculative threads. In this Chapter, different partitioning mechanisms are investigated and their impact on the performance is analyzed.

The idea of partitioning programs into small pieces that run simultaneously to reduce the total execution time is not a novel idea; hundreds of studies have been done, especially for non-speculative threads. Some partitioning schemes rely on programmers to identify which parts of the program can be parallelized and introduce special instructions for spawn, release and cancel parallel threads. Usually, it is necessary to use specific programming languages or directives and it is required that programmers know exactly which parts of the code and under which conditions can be parallelized. Alternatively, automatic parallelization mechanisms can be implemented in the compiler. Current compiler techniques to parallelize applications focus on analyzing well-known structures of the code (for instance, loops) and study the probably data dependences that there may be among the parallel sections (e.g. iterations). Compiler techniques are usually conservative, specially for memory dependences (e.g. a dependence is assumed whenever independence cannot be proved).

Compiler techniques usually work well for regular and numerical applications. These programs are loop-intensive and their memory access patterns are easy to detect at compile time and therefore, easy to disambiguate. However, compiler techniques usually fail to find thread level parallelism when they are applied to irregular or non-numerical applications. Such programs traditionally are subroutine-intensive and the loops are usually short, the number of iterations per loop execution is small, unpredictable and memory dependences are hard to disambiguate. Therefore, finding non-speculative parallel threads is almost impossible.

Speculative multithreading is a promising technique to reduce the execution time of applications by means of relaxing the constraints for partitioning the code into speculative parallel threads. The main difference between such speculative threads and traditional non-speculative threads lays in that the former are both data and control speculative on previous ones (the execution of the thread is not guaranteed to be correct).

In this Chapter, different approaches to split programs into speculative threads are studied and their effectiveness is evaluated. These mechanisms, which will be referred to as spawning policies, try to identify which parts of the code may be speculatively parallelized.

This Chapter is organized as follows: in section 4.2 the basis of the spawning schemes are studied. In section 4.3, the experimental framework is described. The spawning schemes based on heuristics are analyzed in section 4.4. In section 4.5, an automatic method to divide the program into threads is presented and discussed. Related work is commented in section 4.6 and in section 4.7 the main conclusions of the Chapter are summarized.

4.2. SPAWNING AND CONTROL QUASI-INDEPENDENT POINTS

The partitioning process of programs into speculative threads tries to identify which parts of the code are the most suitable to be executed by speculative threads, at which points of the program these speculative threads should be spawned and where the validation of the speculation should be done. The different ways these issues are solved result on the different spawning policies.

A thread spawning operation is implemented by identifying in the code an instruction that when reached it fires the creation of a new thread, an instruction where the spawned thread starts and an instruction where the speculation is fully validated. Note that the validation can be done when the non-speculative thread reaches the first instruction of the next thread. In this way, the instruction where the speculative thread starts and where the speculation is validated may be the same. In this way, the partitioning process just searches for pairs of instructions that are referred to as *spawning pairs*.

A spawning pair is a couple of instructions that identifies the thread spawning operation and are referred to as *spawning point* and *control quasi-independent point*. The spawning point is the instruction that when reached by the processor, it fires the creation of a new thread, whereas the control quasi-independent point is the instruction where the spawned thread starts. The spawning and the control quasi-independent point can be special instructions such as *fork* and *spawn* or conventional instructions in the instruction set with special marks. This simple model of thread-spawning operations may be complicated considering for each spawning point not a single control quasi-independent point but a set of them. In this latter case, one of them is chosen for each particular spawn execution based.

A *spawning policy* is the scheme for selecting the spawning pairs from the potential set of any pair of instructions of the program. Spawning policies can take into account parameters such as the type of the instruction at the spawning and the control quasi-independent points, the distance among them, the dependences between the instructions executed by one thread and the other, etc.

There are different ways to implement the selection rules that define a spawning policy. It can be done dynamically with some special hardware support that spawns a new thread every time a particular instruction is reached or with compiler support that analyzes the code and identifies those pairs that provide better performance. In the case that the spawning policy is fully implemented by the hardware, an existing instruction is used to fire the creation of a new thread and it should include the address of where the new speculative thread will start or it should be easily computable. In the latter case, new instructions to the instruction set architecture may be added. Compiler techniques can also take benefit of additional resources like profile information to select the best spawning pairs.

To determine which of the best spawning policy, different metrics can be considered:

- Coverage: It refers to the percentage of the program that may be executed in parallel with other threads. Thus, low coverage implies that most of the code will not benefit from exploiting speculative thread-level parallelism.
- Average Active Threads per Cycle (TPC): This metric refers to the average number of threads that are executing instructions simultaneously. Having a large coverage but a low TPC indicates that most of the time few threads are executing instructions and the rest of the contexts of the processor are idle. On the other hand, having a TPC close to the number of contexts of the processor indicates that near the maximum number of threads are running most of the time.

The previous metrics may help to determine the goodness of a spawning policy, but the metric that really determines if a spawning policy is better than another is execution time. If an application executed in a speculative multithreaded processor with a given spawning policy finishes earlier than with other spawning policy, the former is better. The reasons why a spawning policy with less coverage and worse TPC outperforms others in execution time is due in most of the cases to the quality of the speculative threads. For instance, if speculative threads are data dependent on previous ones, they may have to wait until the dependent values are produced and forwarded to them to continue execution whereas if speculative threads are almost data independent they may proceed in parallel most of the time.

Spawning pairs should meet some requirements in order to be effective:

- Control independence: The probability of reaching the control quasi-independent point after visiting the spawning point should be high in order not to waste resources executing instructions that may never be reached.
- Thread size: The distance between the spawning point and the control quasi-independent point should not be too small nor too large to keep the thread size into a certain limit. Small threads result in too much spawning overhead and large threads may require large storage for speculative state.
- Workload balance: Threads that are executed by around the same time should have approximately the same size since threads are committed in sequential order and resources are not freed until commit.
- Data independence: Instructions after the control quasi-independent point (the instructions that will correspond to the speculative thread) should have few dependences with instructions above it.
- Data predictability: If the mechanism considered to deal with data dependences among concurrent threads is value prediction, it is desirable that the dependent values among threads are predictable to reduce the cost of misspeculations.

The previous list does not try to be an exhaustive list. Other criteria may try to hide the cost of hard-to-predict branches, load misses, etc. For instance, good candidates can be spawning pairs with high control independence with respect to hard-to-predict branches or load misses between the spawning point and the control quasi-independent points since this may help to hide the latency of possible mispredictions.

Basically, two families of spawning policies can be considered. One is based on heuristics and it has been widely studied in the latest years. This family of spawning policies does not consider the whole set of instructions pairs as candidates for spawning pairs but a small subset of them. Such candidate spawning pairs are usually well-known program constructs. Examples of this family are the loop-iteration, the loop-continuation and the subroutine-continuation spawning schemes, which will be analyzed in detail in section 4.4.

On the other hand, in this thesis a new family of spawning policies is proposed. These spawning schemes do not take into concrete program constructs as opposed to the previous family of spawning policies. In this family, spawning pairs are selected from the whole set of pairs of instructions and only those that meet some criteria (i.e. control quasi-independence, thread size, data independence among others) are considered. This method introduces spawn and fork instructions in the code based on an off-line analysis with support of profiling data. This off-line analysis is implemented as a stand alone module. This inclusion in a compiler can be even more effective but this is not considered in this thesis.

4.3. 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 unrestricted spawning model has been considered and the cost of spawning a speculative thread is assumed to be 0. The impact on the performance of having a non-zero initialization overhead will be investigated in the next Chapter.

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

In order to evaluate the potential of the spawning policies, we have first considered an idealized scenario. Three different approaches to deal with data dependences among instructions in different threads are evaluated. In the first model, all values corresponding to inter-thread dependences through both registers and memory are assumed to be correctly predicted. This model is referred to as *perfect register and memory prediction*. In the second model, inter-thread dependent register values are assumed to be correctly predicted but inter-thread dependent memory values must be forwarded from the producer to the consumer and the delay has been assumed to be 3 cycles. This model is referred to as *perfect register prediction*. Finally, the last model, which is called *synchronization* model, assumes a perfect synchronization mechanism. In this last model, the delay for forwarding from the producer thread unit to the consumer is assumed to be 3 cycles for memory values and 1 cycle for registers. In the following Chapter, realistic value predictors will be considered and their implications on the performance will be analyzed.

Performance is by default reported as the speed-up over a single-threaded execution.

4.4. THREAD-SPAWNING POLICIES BASED ON HEURISTICS

In this section, the spawning policies based on heuristics are analyzed. In this family of spawning policies, speculative threads are usually assigned to common program constructs. Examples of this kind of program constructs are loops and subroutines. In this section, we thoroughly analyze three of these possible spawning policies. Speculative threads in these policies are started at:

- The next loop iteration: This scheme will be referred to as the *loop-iteration spawning scheme*. Here, the spawning point and the control quasi-independent point may be any instruction of a loop. For instance, if the first instruction of a loop is chosen, every time an iteration of the loop is started, a new speculative thread is spawned at the same instruction in order to execute the next iteration of the loop.
- The continuation of a loop: This scheme will be referred to as the *loop-continuation spawning scheme*. Here, the spawning point is the first instruction of the loop and the control quasi-independent point the following instruction in static order of the backward branch that closes the loop. In this scheme, when the first instruction of a loop is reached, a new speculative thread is spawned in such a way that the thread that has fired the creation of the speculative thread will execute all the iterations of the loop whereas the spawned thread will execute the code after the loop.
- The continuation after a subroutine return: This scheme will be referred to as the *subroutine-continuation scheme*. The spawning point is the subroutine-call instruction and the control quasi-independent point the following instruction in static order after the spawning point. In this scheme, when the subroutine-call instruction is reached, a new speculative thread is created just after the call. In this way, the thread that has fired the creation of the speculative thread will execute the body of the subroutine whereas the spawned thread will execute the code after the subroutine returns.

These program constructs are common enough to potentially offer a good coverage and lots of opportunities for thread creation. Additionally, their corresponding spawning pairs meet quite well the control independence criterion. For instance, it is very likely that when a subroutine-call instruction is reached, the following instruction in static order will be reached once the subroutine returns.

An additional advantage of this family of policies is that is quite straight-forward to be implemented only by hardware mechanisms since those program constructs are delimited by easily detectable instructions such as subroutine-call and return instructions and backward branches.

In the following subsections, each of these schemes is studied. Finally, a scheme that combines the three previous ones is considered.

4.4.1. Loop-Iteration Spawning Scheme

The loop-iteration spawning scheme is based on assigning different iterations of a loop to speculative threads. The idea is very simple and it has been widely used to parallelize non-speculatively numerical applications. In the case of irregular or non-numerical applications, loops are usually smaller, with less number of iterations, with more dependences among them and harder to disambiguate.

Anyway, spawning speculative threads at loop iterations initially seems to be an excellent choice to obtain significant benefits. Practically, all the dynamic instructions of a program are executed inside a loop in such a way that it may guarantee very high coverage. In addition to the coverage, loops are quite common in the code so the number of threads that might be spawned is also very high.

Regarding the effectiveness of the loop-iteration spawning scheme, the spawning pairs of this policy have significant control independence. For instance, consider the first instruction of a loop as the spawning point and the same instruction as the control quasi-independent point. In this case, the probability that the control quasi-independent point is not reached once the spawning point has been visited is equal to the inverse of the average number of the remaining iterations of such loop (i.e. if the remaining number of iterations of a loop is 100, the probability to spawn a speculative thread incorrectly is 0.01). In other words, once the first instruction of a loop iteration is reached, it is very likely that this same instruction will be executed in a near future by the following iteration of the same loop independently of the control flow taken between them.

In addition to that, loop iterations may have a similar size, which favors workload balance, although this may not true when concurrent iterations of the loop follow very different paths.

Regarding dependences and the predictability among speculative threads based on loop-iterations, it has been shown in the previous Chapter that the number of dependences is not high and the values that flow from one thread to the other are quite predictable.

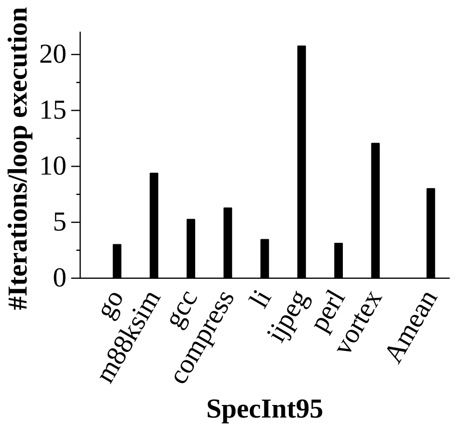


Figure 4.1. Average number of iterations per loop execution.

The spawning pairs of this spawning policy are usually characterized by the fact that the spawning and the control quasi-independent point are the same static instruction. This instruction can be anyone in the loop body, but the selection will affect performance. For instance, choosing an instruction in the middle of the loop body that only belongs to a subset of the control flows in the loop will cause that the spawned speculative threads may execute more than one iteration and thus will affect the workload balance. To avoid this undesirable situation, the instruction selected to be the spawning and the control quasi-independent point is usually one of the loop body that is executed by all the iterations. Examples of these instructions are the first instruction in the loop body and the last one (the backward branch that closes the loop). The behavior of these two alternatives is slightly different since in the case of the last instruction of the loop, the creation of the new speculative thread is delayed until the first iteration of the loop is completed and such thread will execute the third iteration (the second will be executed by the thread that spawns the speculative thread) whereas in the case of the first instruction, the speculative thread will execute the second iteration of the loop while the spawner thread will execute the first. Figure 4.1 presents the average number of iterations of the loops from the SpecInt95. On average, this number is about 8 iterations per loop execution even though for most of the benchmarks, this number is quite low (less than 5 iterations). This suggests that missing an iteration per loop execution may result in a significant drop in performance.

Regarding the way the spawning pairs can be identified, there are basically two approaches. At compile time, loops are easy to detect and a new instruction for thread spawning can be inserted at the beginning of the loops or such instructions may be annotated. Alternatively loops can be detected at run time. This mechanism is based on the observation that loops are usually closed by a backward branch. This mechanism is based on the Loop Execution Table and the Current Loop Stack[77].

The Current Loop Stack is a small stack that keeps information about the nesting of the loops that are being executed. On the other hand, the Loop Execution Table keeps information about the number of iterations for every loop that has been found in the code. This information can be used to predict the number of iterations for future executions of the loop, which may avoid to spawn threads that execute non-existent iterations. These structures basically work as follows: when a taken backward branch is found, a loop is assumed to be found and the target address of the branch is considered the first instruction of the loop. The Current Loop Stack is looked up with this address. If it is found at the top of the stack, then a new iteration of the same loop is assumed to be found and their current counter of iterations is increased. If it is found, but it is not at top of the stack, then all the loops above of it in the Current Loop Stack are assumed to be finished and their number of iterations are updated in the Loop Execution Table. If the instruction is not found, a new entry at top of the Current Loop Stack is allocated to indicate a new execution of the loop. When a not taken backward branch is found the Current Loop Stack is also looked up. If it is found, then the execution of the corresponding loop has finished, the entry and all the entries above are eliminated and their corresponding entries in the Loop Execution Table are updated with the number of iterations.

Any time an instruction is fetched, it is compared to those instructions stored in the Loop Execution Table. In case of hit, a new execution of the loop is assumed to be found and as many speculative threads as free thread units or the number of iterations, whichever is lower, are created.

Since more speculative threads are squashed by less speculative ones, most of the time, this spawning policy only speculates on innermost loops, since speculative iterations of outer loops would be cancelled by the iterations of the innermost ones. Outer loops are larger, so they require larger storage for speculative state and more live-in values to be predicted. On the other hand, larger threads result in smaller overhead due to thread spawning activities.

Mechanisms to avoid firing speculative threads on those spawning pairs that could provide small benefits or limit the benefits produced by other spawning pairs will be later presented in subsection 4.5.3. The performance results presented in subsection 4.4.4 do not take into account these mechanisms to inhibit the spawning process.

Finally, there is a beneficial side-effect for those loops whose iterations follow the same control-flow. In this case, those speculative threads that share the same control-flow can fetch the same instructions once and be used by all the thread units that require them. This phenomena may be interesting to increase the fetch bandwidth for those speculative multithreaded processors whose fetch unit is shared among the thread units. This situation is really common for regular and numerical applications, but not common for

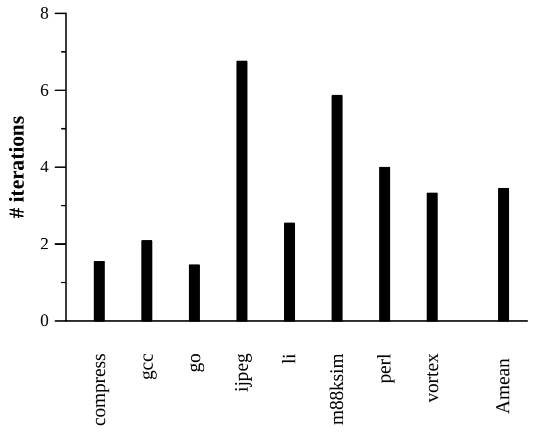


Figure 4.2. Average number of consecutive iterations that follow the same control-flow.

irregular and non-numerical programs. Figure 4.2 shows the average number of consecutive iterations that follow the same control-flow for the innermost loops in the SpecInt95. We can observe that on average there are only less than 4 consecutive iterations and for programs like `go` and `gcc` it is lower than 2. On the other hand, the number of different control flows followed by every consecutive 8 iterations is quite low as can be observed in Figure 4.3. In fact, excepting `go`, `compress` and `gcc`, the average number of iterations with different control flows is lower than 2.

A similar feature is exploited by the CONDEL architecture [81] and the dynamic vectorization approach proposed in [80]. However, those approaches are more restrictive since the former is limited to loops whose static body does not exceed the implemented instruction window, whereas the latter is feasible only if the dynamic sequence of instructions executed by the loop are the same for all the iterations and they fit into a single instruction cache line (the instruction cache organization that they use is the trace cache [60])

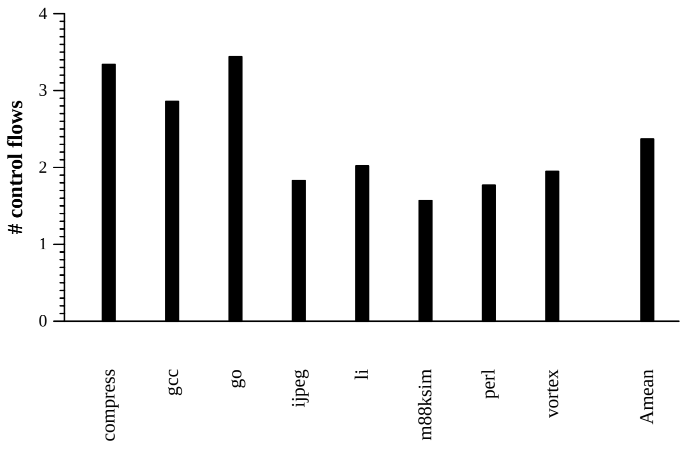


Figure 4.3. Average number of different control flows in the last 8 iterations of innermost loops.

4.4.2. Loop-continuation spawning scheme

The loop-continuation spawning scheme spawns a new speculative thread starting at the end of the loop while the spawner thread continues with the whole execution of the loop. This results in larger threads than those generated by the loop-iteration scheme.

Similarly to the loop-iteration scheme, the abundance of loops in the code might result in a high coverage and lots of opportunities to spawn speculative threads. However, the effectiveness of this policy is not as evident as for the previous scheme. It seems quite obvious than in general, once all the iterations have been executed, the instructions beyond the backward branch that closes the loop will be executed, so the reaching probability is high. Problems regarding control independence may come from possible optimizations of loops with different exits since the control quasi-independent point selected could be never reached. On the other hand, workload balance among threads is harder to obtain since the speculative threads that run in parallel may have quite different sizes since they may belong to executions of different loops that might have different code or just executions of the same loop but each of them executing different number of iterations.

The spawning pairs used by this spawning scheme are made up by the first instruction of a loop, as the spawning point, and the following instruction in static order of the branch that closes the loop, as the control quasi-independent point.

In the same way of loop-iteration scheme, this spawning policy can be implemented with or without compiler support. At compile-time, a spawning instruction can be added at the beginning of the loops for creating a new speculative thread at the point that the loop finishes. Alternatively, this partitioning mechanism can be dynamically implemented using the Current Loop Stack and the Loop Execution Table described in the previous subsection.

4.4.3. Subroutine-continuation spawning scheme

The subroutine-continuation spawning scheme triggers the spawning process at subroutine call instructions and creates a speculative thread at the point the subroutine returns, that is, the following instruction in static order after the call instruction. The pairs of instructions that make up the spawning pairs for this spawning policy consist of the subroutine-call instruction, as the spawning point, and its following instruction in static order, as the control quasi-independent point.

Subroutines are also quite abundant in many codes so coverage is high. In addition to that, control independence is almost guaranteed except for very rare cases where a subroutine does not return to the

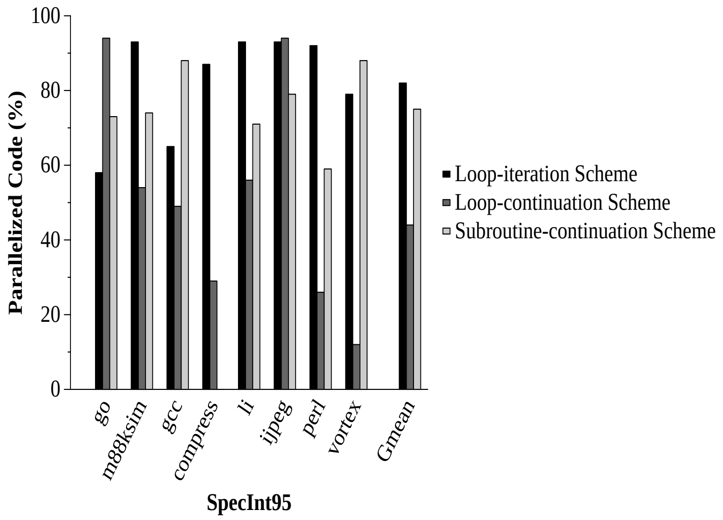


Figure 4.4. Percentage of code executed in parallel with other threads for each spawning policy.

point where it was invoked. However, it is not clear that this policy provides good workload balance among the threads since subroutines may have very different sizes.

Subroutines may be nested into other subroutines. Recursive subroutines may require a special treatment since they may result in an overpopulation of threads.

4.4.4. Performance figures

The following statistics have been collected from executing 300 million of instructions after skipping initializations. Speculative threads have been created without any compiler support and the spawning process only relies on hardware techniques. No statistics are reported for `compress` since no subroutines were found during the execution of those 300 million of instructions.

Regarding coverage, figure 4.4 shows the percentage of code that is being executed in parallel with the code of any other thread on a 16-thread unit configuration and perfect register value prediction. We can observe that this fact produces a high percentage of code that is executed in parallel. Observe that for codes such as `m88ksim`, `li`, `perl` and `ijpeg` practically the whole code is executed in parallel with some other part of the code when speculating on loop iterations. On average, 83% of the code is executed in parallel for this scheme. The coverage of the subroutine spawning policy is also quite high (76% on average) and it suffers a significant drop for the loop-continuation scheme. There are several reasons that justify the low coverage presented in this figure for the loop-continuation scheme. The first one is branches inside the loop that exit the loop but do not jump to the instruction after the backward branch that close the loop. On the other hand, a less important reason is the granularity of threads. It has been evaluated that the size of

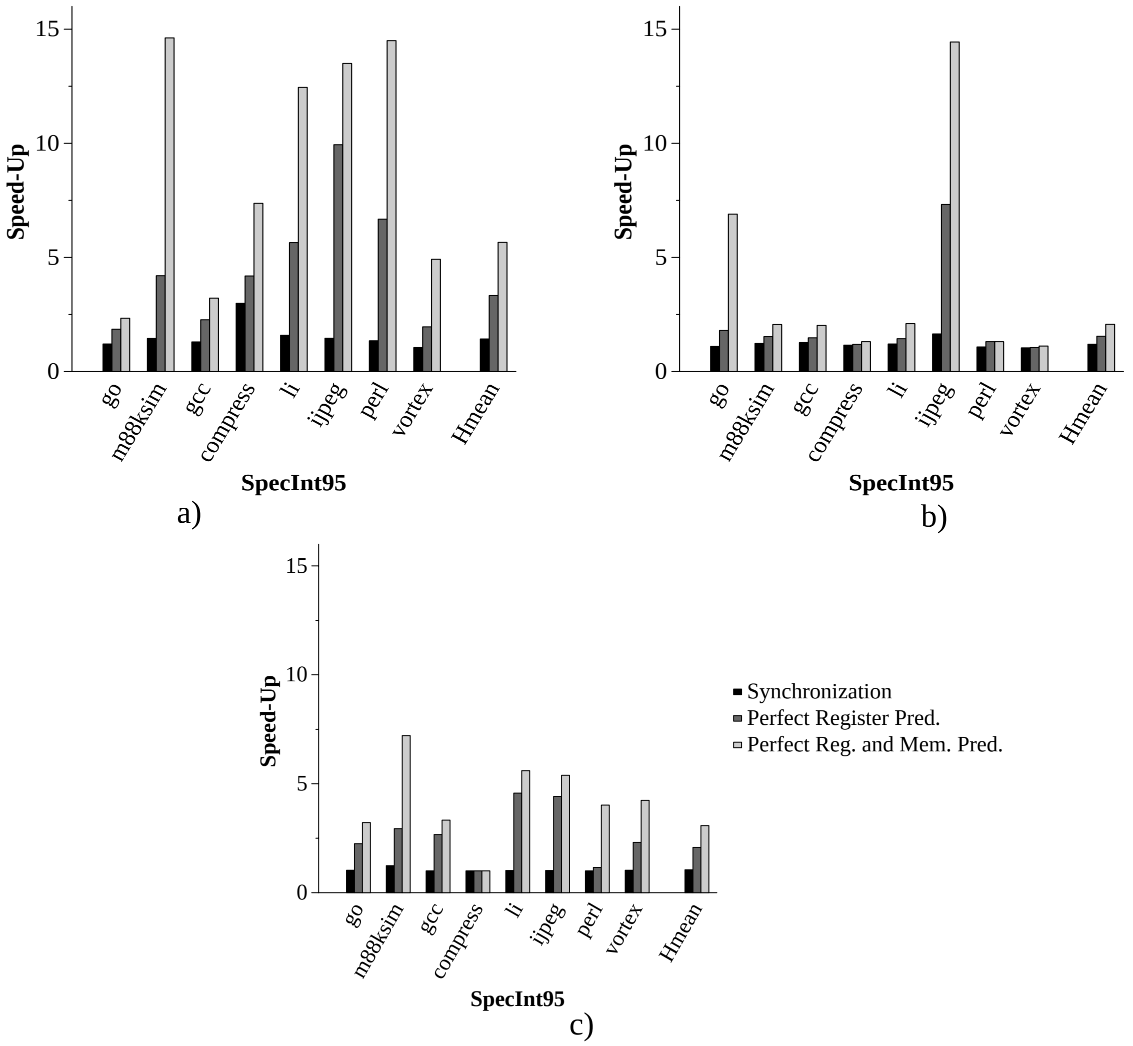


Figure 4.5. Speed-ups for the three different spawning polices a) loop-iteration, b) loop-continuation and c) subroutine-continuation for the unrestricted thread ordering scheme.

the speculated threads in the loop-continuation scheme is significantly greater than for the loop-iteration scheme. Then, speculating too far may cause that the thread units have to store large amount of speculative state and in case that it exceeds the storage capacity, the speculative thread stalls execution.

These high percentages of parallelized code are translated in high speed-ups as shown in Figure 4.5, especially for the models that speculate on loop iterations and subroutines. For these two scenarios, the processor can achieve an average speed-up of 5.7 and 3.1 respectively with 16 thread units. The loop-continuation scheme has the lowest performance (only 2.1x on average). This is due to its relatively low coverage and the workload imbalance caused by loops with different number of iterations.

4.4.5. Combining Schemes

The set of spawning pairs for the subroutine-continuation scheme is quite different to the others whereas for the loop-iteration and the loop-continuation scheme the spawning pairs have some intersection. In fact considering a combination of both loop-based policies produces quite similar results to those obtained by the loop-iteration scheme alone since the execution model will eliminate in most of the cases the additional speculative threads created by the loop-continuation scheme. This is due to the fact that iterations are less speculative in program order than the continuation of the loop and in case of no idle thread units, the most speculative threads are squashed to give the thread unit to the less speculative thread.

However, a synergistic effect is expected from combining the subroutine-continuation scheme with the others since the domain of the selected spawning pairs for subroutines is different from those of the spawning policies based on loops. Because of this, in this section a new spawning policy based on the previous ones is presented. The set of spawning pairs for this policy is the result for applying the union of the sets of spawning pairs of the spawning policies individually. That is, this policy spawns new speculative threads any time a subroutine-call is found starting at the following instruction in static order and any time a first instruction of a loop is reached. In this latter case, two speculative threads, one for the next iteration of the loop and another one for the continuation of the loop are spawned. The speculative thread that executes the second iteration of the loop will spawn another speculative thread for the third iteration, and so on. So, if the loop has more iterations than idle thread units, the thread created at the loop continuation will be squashed to allow a less speculative thread to be executed.

The only constraint applied to the spawning pairs of this policy is that the number of instructions between the spawning and the control-quasi independent point should be at least 32 instructions in order to avoid that the benefits of a parallel execution are offset by the thread initialization overhead.

Figure 4.6 presents the average speed-up assuming perfect value prediction for register values and synchronization for the memory values. These statistics have been collected for the whole execution of the SpecInt95 with the train input set. It can be observed that the results presented by this spawning policy are quite impressive, reporting an average speed-up higher than 6.5 and are especially high for `li`, `ijpeg`, `m88ksim` and `perl`. Mechanisms to improve the performance of this spawning policy will be further detailed in next subsection.

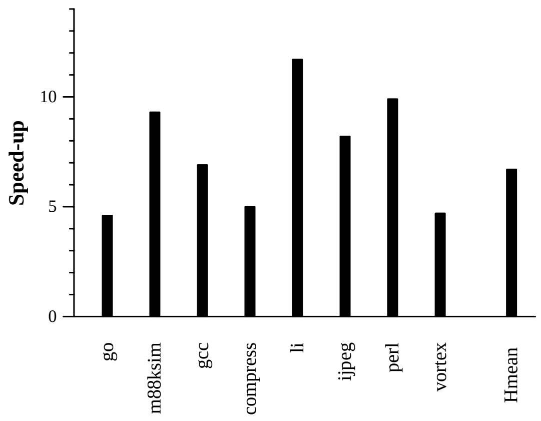


Figure 4.6. Speed-up of the combination of heuristics compared with a single-threaded execution.

Figure 4.7 shows the average number of active threads per cycle. As it can be expected, the average number of active threads per cycle is quite related to the speed-up. On average, the number of active threads is 5.9 and for `li` it is higher than 9.5.

Figure 4.8 shows the percentage of the code that is executed overlapped with other active threads. On average, the percentage of code is about 88% and higher than 90% for the benchmarks with the largest speed-ups such `m88ksim`, `li` and `perl`. The largest percentage corresponds to the `m88ksim` with a 94% and the lowest to `vortex` with a 78%.

4.5. Thread-Spawning Policies Not Based on Heuristics

The spawning policies presented in the previous section have shown impressive benefits. However, the way the spawning pairs are selected by such schemes is quite simple, they choose the spawning pairs by the fact they belong to concrete program constructs.

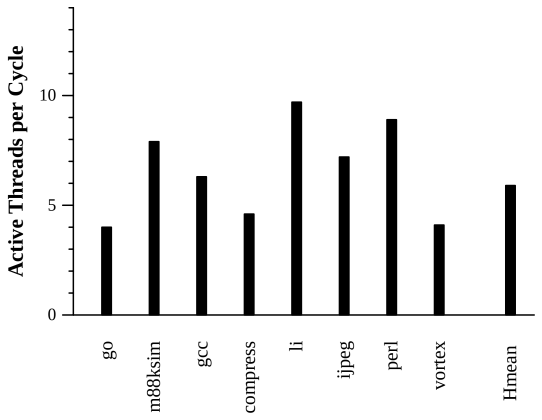


Figure 4.7. Average number of active threads per cycle.

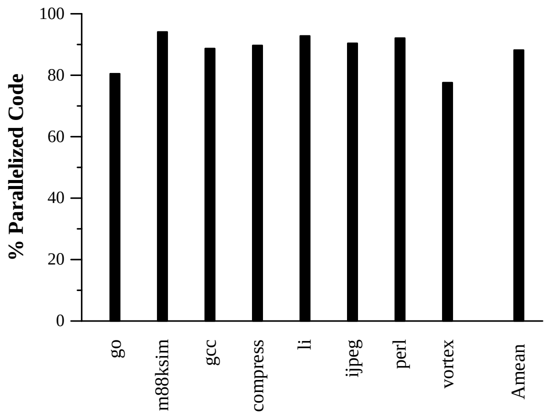


Figure 4.8. Percentage of parallelized code.

In this section, a systematic approach to identify spawning pairs is presented. This scheme is based on quantifying the relevant properties of every section of the code, in order to identify those points that offer the highest potential.

Some of the criteria to be taken into account to get effective spawning pairs have been pointed out previously in this chapter:

- The probability to reach the control quasi-independent point after visiting the spawning point should be very high in order not to spawn speculative threads which are very unlikely to be needed.
- The distance between the spawning point and the control quasi-independent point (i.e. the average number of instructions between both points) should not be too small or too large since small threads have a too high initialization overhead whereas large threads require a too large speculative storage.
- Instructions of the speculative threads should have few dependences with instructions of previous threads or, at least, the values that flow through such dependences should be predictable.

The loop-iteration, the loop-continuation and the subroutine-continuation schemes meet the first criterion but not necessarily the others.

The family of spawning policies presented here are based on identifying spawning pairs anywhere in the code. In this way, speculative threads are not necessarily associated with any particular program construct (e.g. loop iteration) and any instruction can be a spawning point or a control quasi-independent point. Thus, unlike to the spawning policies based on heuristics, an only-hardware implementation is more difficult for this family of spawning policies due to the fact that a more complex analysis to determine the effectiveness of any pair of instructions must be performed.

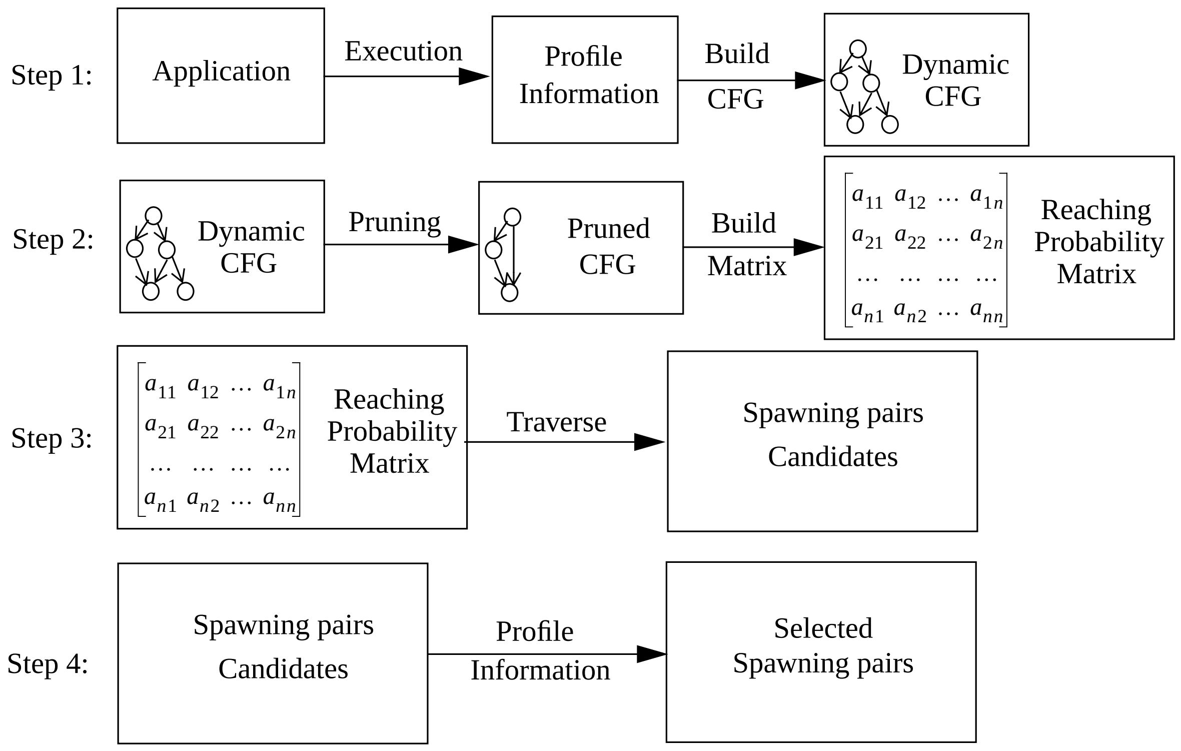


Figure 4.9. Steps of the profile-based spawning scheme.

In this thesis, an off-line analysis with the support of profiling information is presented.

4.5.1. Profile-based Spawning Scheme

In this subsection, a technique to identify spawning pairs based on a profile-based analysis of the properties of any potential section of code is presented. Any basic block of the program can be either a spawning and control quasi-independent point. A basic block is a set of instructions that are either all executed or none of them.

Even though the criteria considered here can be applied to instruction granularity, in order to reduce the amount of computation, the proposed analysis looks for pairs of basic blocks instead of pairs of instructions. The result of this analysis will be spawning pairs made up of the first instructions of the basic blocks considered as spawning and control quasi-independent points.

Figure 4.9 illustrates the different steps followed by this spawning scheme. Initially, a dynamic control flow graph of the program is built from the profile information obtained through an execution of the program. Each node of the graph represents a basic block and edges represent possible control flows among blocks. Edges are weighted with the frequency of the corresponding control flow obtained during profiling.

To reduce the size of the graph, the least frequently executed basic blocks of the dynamic control flow graph are pruned. To do this, basic blocks are ordered by execution count and they are chosen from highest to lowest count until a certain percentage 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 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 one is computed. We will refer to these probabilities as *reaching probabilities*. These probabilities are stored in a two-dimensional square matrix (i.e. the *reaching probability 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 edges that exist from the source node to the destination node. 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 (the spawning node and the control quasi-independent node can be the same, and any other basic block can appear more than once in the sequence.)

This constraint also reduces the control logic of the processor since otherwise, the identification of the starting and ending points of each thread would be quite cumbersome. This restriction also simplifies the execution model of the architecture since if the spawning point may appear more than once in the path, the processor should know that speculative threads only have to be spawned at the first instance of the spawning point and not for the following *n*. Similarly if a control quasi-independent point appears more than a once, a thread would not finish its execution when it is reached the first time but at the n^{th} appearance.

Once the reaching probabilities are computed, the pairs that do not accomplish the minimum requirements to be considered as good candidates are eliminated. The first property that they must satisfy is that their associated reaching probability should be very high, i.e. higher than a given threshold.

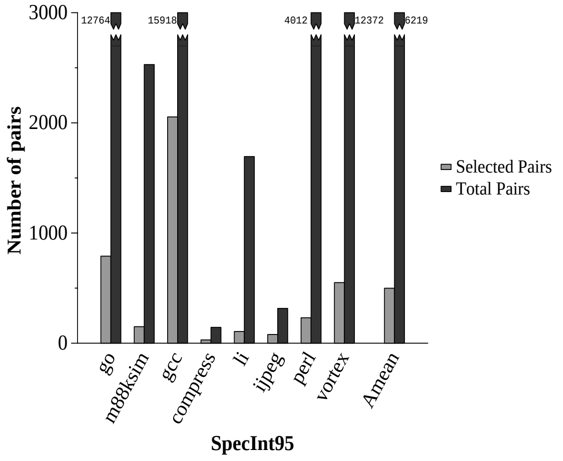


Figure 4.10. Number of pairs of basic blocks selected and number of selected pairs that have different spawning points.

A second requirement 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, the average number of instructions between the source node and the destination node is also calculated. The average is calculated as the sum of the number of instructions executed by each sequence of basic blocks multiplied by their frequency.

For a given spawning point, there may be several good candidates for its associated control quasi-independent point (i.e. for a given row of the probability matrix, there may be more than one element that exceeds the minimum probability and the minimum size). Figure 4.10 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. For this experiment, the minimum distance between the spawning point and the control quasi-independent point is 32 instructions and the minimum reaching probability is 0.95.

When the processor reaches the spawning point, it starts 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 their expected benefits. Several alternative criteria can be considered to produce such an ordering. In this thesis, three criteria have been evaluated:

- Maximizing the distance between the spawning and the control quasi-independent point. This distance can be considered as 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.

- Maximizing the number of instructions of the spawned thread that are independent of previous instructions.
- Maximizing the number of instructions of the spawned thread that are either independent of or dependent on predictable values generated outside the thread.

Other examples of criteria that could be taken into account are the number of live-in values or the impact of branch mispredictions and cache misses.

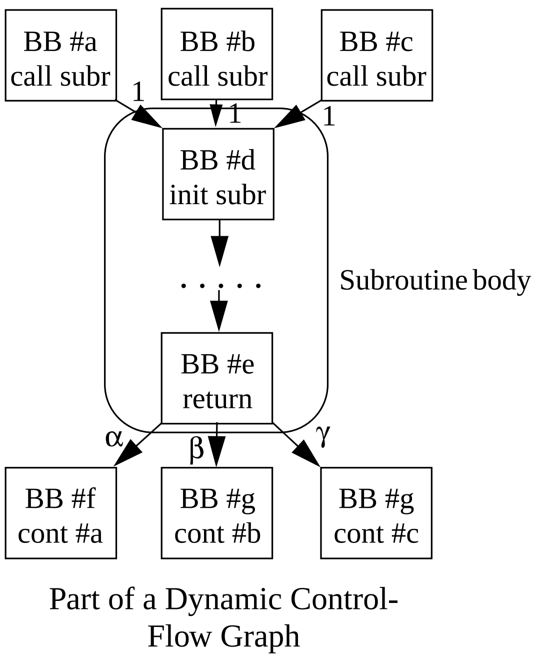
When ordering the spawning pairs with the same spawning point, each of the spawning pairs is considered as if it was the only spawning pair, so no interactions with other close spawning pairs are considered. More complex analysis schemes that take into account interactions with other threads is left for future work.

4.5.2. Performance evaluation

The following statistics have been collected after executing of the SpecInt95 benchmarks until completion with the train input set. Profile analysis has been performed on the same input set. Unlike the spawning schemes based on heuristics, here, the speculative threads have been created through an off-line analysis that introduces fork and release instructions at the appropriate locations of the code.

We have considered two different spawning policies. In the former, spawning pairs have been selected with the technique proposed in the previous subsection and the order policy among the spawning pairs maximizes the distance between the spawning and the control quasi-independent point. The minimum reaching probability considered is 0.95 and the minimum distance between the points is 32 instructions.

In the latter scheme, in addition to the spawning pairs selected in the previous scheme, all call-return pairs (pairs of a subroutine-call and the corresponding return point) are added if they satisfy the minimum size constraint since some of them may not have been selected by the previous algorithm. Those spawning pairs have a very high reaching probability (they are almost control-independent) but due to the way the reaching probabilities are computed, many of them are missed. This problem is illustrated in figure 4.11. This figure shows a piece of the dynamic control-flow graph of a program where a subroutine is called from multiple locations. The subroutine-return basic block has as may successors as different locations from which the subroutine has been invoked. The weights in each edge correspond to the frequency the subroutine has been called from each respective location (i.e. α, β and γ). Thus, when the reaching probability matrix is computed, as the calculation only takes into account the information edge profiling as



Computed Reaching Probability Submatrix

$$\left[\begin{array}{c|ccc} \dots & f & g & h \\ \hline a & \alpha & \beta & \gamma \\ b & \alpha & \beta & \gamma \\ c & \alpha & \beta & \gamma \end{array}\right]$$

Real Reaching Probability Submatrix

$$\left[\begin{array}{c|ccc} \dots & f & g & h \\ \hline a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ c & 0 & 0 & 1 \end{array}\right]$$

Figure 4.11. Computed and Real reaching probability submatrix for a subroutine invoked from more than one place in the code.

opposed to path profiling, if the values from α, β and γ are lower than the threshold, the probability from going to the caller block to the corresponding continuation block will be computed as α, β and γ respectively, whereas the correct value should be 1.

Figure 4.12 shows the speed-up over a single-threaded execution on a 16-thread unit fully-interconnected Clustered Speculative Multithreaded Processor with a perfect inter-thread register prediction. On average, the speed-up achieved is 5.9 without considering call-return pairs and it is improved up to 7.2 when they are taken into account.

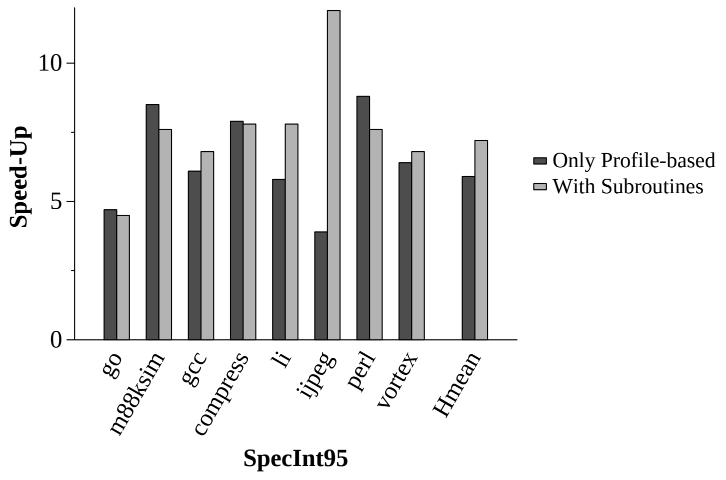


Figure 4.12. Speed-up over a single-threaded execution obtained for 16 TU.

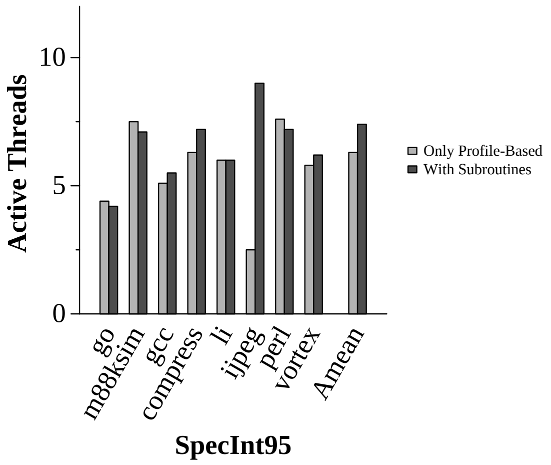


Figure 4.13. Average number of active threads per cycle.

It is especially remarkable the significant improvement of the `ijpeg`. In this case, most of the code is in small loops inside subroutines. Such loops have a huge number of iterations, but the body of the loop is smaller than 32 instructions. Then, even though the reaching probability is very high, the distance between the spawning point and the control quasi-independent point is lower than the threshold and such spawning pairs are eliminated. Moreover, the subroutines that contain these loops are called from different locations in such a way that the computed reaching probability is lower than 95%. Adding the spawning pairs associated to the subroutines allow the processor to parallelize such subroutines.

Call-return pairs achieve some improvements also for `gcc`, `li` and `vortex` and a slight slow-down for the rest of benchmarks. The main reasons for such slow-down will be deeply analyzed in the next subsection, but it can be anticipated that increasing the number of spawning pairs does not always imply better performance since spawning too many threads may cause overpopulation. This overpopulation can provoke that more useful speculative threads are not spawned since the processor has the thread units busy with other speculative threads that will provide less performance.

Figure 4.13 shows the average number of active threads per cycle. As it is expected, the average number of active threads per cycle is increased when subroutine pairs are added. This is specially remarkable for `ijpeg`. On average, the profile-based spawning mechanism has 6.3 active per cycle and it grows up to 7.4 on average when subroutine pairs are added.

Finally, figure 4.14 shows the coverage of both policies. No important differences exist among them with the exception of the `ijpeg`. Note that even though the coverage for the `m88ksim` is higher, for the profile-based scheme with subroutines, the performance showed in figure 4.12 is lower, and the opposite is true for `gcc`.

4.5.3. Improving the performance

In spite of the excellent performance results reported above, on average more than 50% of the thread units remain idle or executing threads that will be later squashed. This may be due to application's features, but also to some limitations in the spawning policy.

There are several undesirable situations that can reduce the performance of the speculative multi-threaded processor:

- The processor is executing for a long period of time just a few threads. This situation may obey to different causes. The most frequent one is the lack of confident spawning pairs in that part of the code. In this situation, the approach to select the spawning pairs may use more relaxed constraints (e.g. reducing the minimum reaching probability between the spawning and the control quasi-independent point). The other situation may be due to load imbalance. Since speculative threads must commit in program order, thread units become available in the same order. Threads that finish earlier than their predecessors make their thread unit to remain idle until they can commit.
- There are some spawning pairs that never spawn useful speculative threads. This may happen when another thread has previously spawned a speculative thread at the same control quasi-independent point.

In this subsection some mechanisms to reduce those sources of inefficiency are presented.

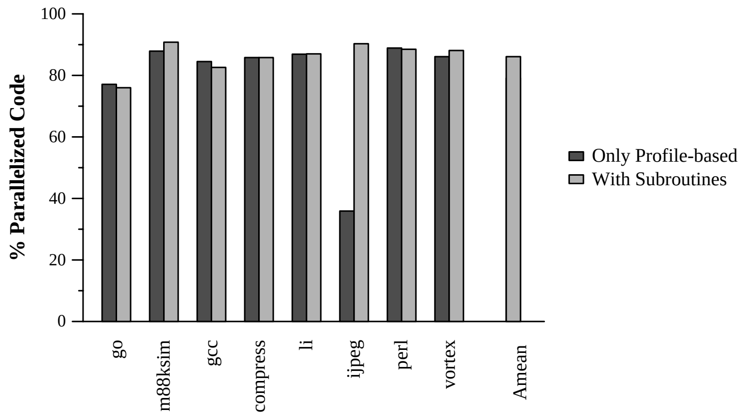


Figure 4.14. Percentage of code that is executed in parallel with some other code.

4.5.3.1. Cancellation Spawning Policy

Figure 4.15 shows the utilization of the thread units per each benchmark of the SpecInt95 for the profile-based spawning scheme with the call-return pairs. As it is expected, these figures show a behavior close to a Gauss function centered on the average active threads per cycle. In the case of the benchmark

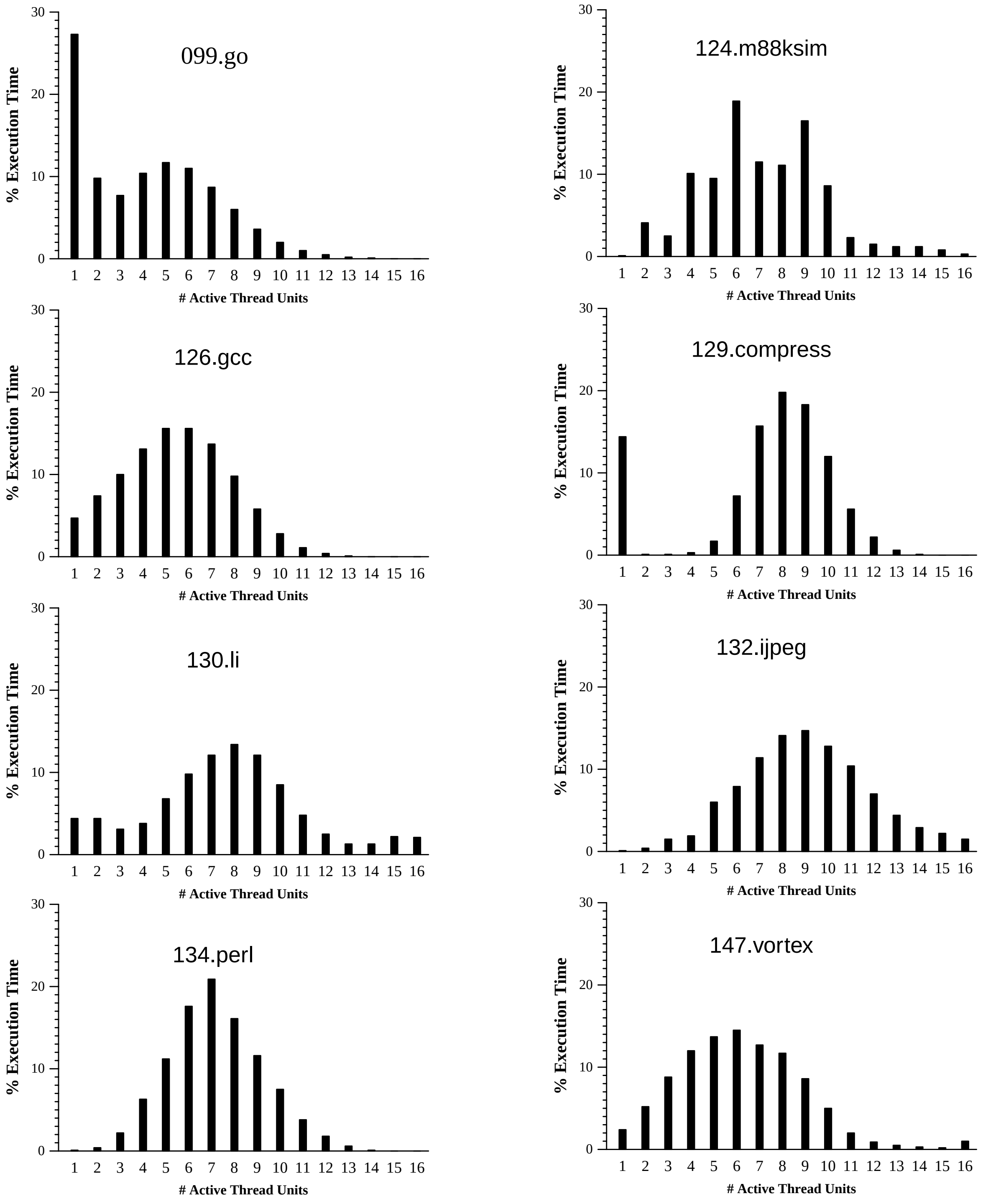


Figure 4.15. Thread Unit Utilization for the Profile-based spawning scheme with the call-return pairs.

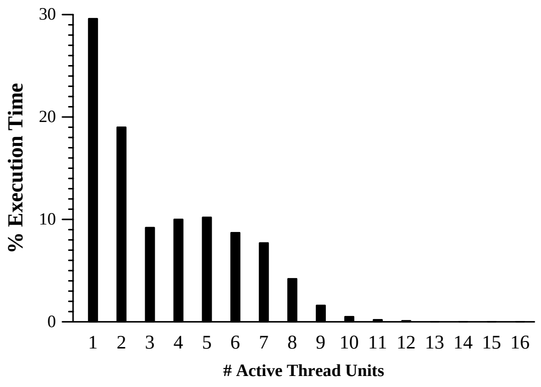


Figure 4.16. Percentage of time the spawning pair <9360-9361> of the go benchmark is executed simultaneously with another threads.

go, during more than 25% of the execution time, just one thread is being executed. For compress, its corresponding graph approaches a Gauss function excepting for the peak at the single threaded execution, which represents almost 15% of the execution time.

Observing the individual contribution of each of the spawning pairs, we have observed that when the processor is executing a few number of threads, in many cases the implicated spawning pairs are usually the same. In fact, 80% of the time that go is executing 1 or 2 threads, only 2 particular spawning pairs are involved. In the case of vortex, 85% of the execution time with less than 3 threads is due just to 3 spawning pairs and for compress 2 spawning pairs are responsible for the 99% of the single-threaded execution.

The cause for such low average of active threads per cycle for such spawning pairs can be due to different factors. For instance, it may be possible that after the control quasi-independent point or between the spawning and the control quasi-independent point there are no other spawning pairs in such a way that no other speculative thread are spawned.

An example of this is showed in figure 4.16. This figure focuses on the spawning pair of go that has its spawning point at the basic block 9360 and its control quasi-independent point at basic block 9361 (it corresponds to a call-return point). The figure represents the percentage of time the speculative thread created by such spawning pair is executing with some other threads. The speculative thread created at that point represents the 5% of the total instruction count of the benchmark and it represents the 45% of the total of time the benchmark is executing alone and the 39% when is executing two threads. It can be observed that almost half of the time, the speculative thread created by this spawning pair is executing together with a few number of threads.

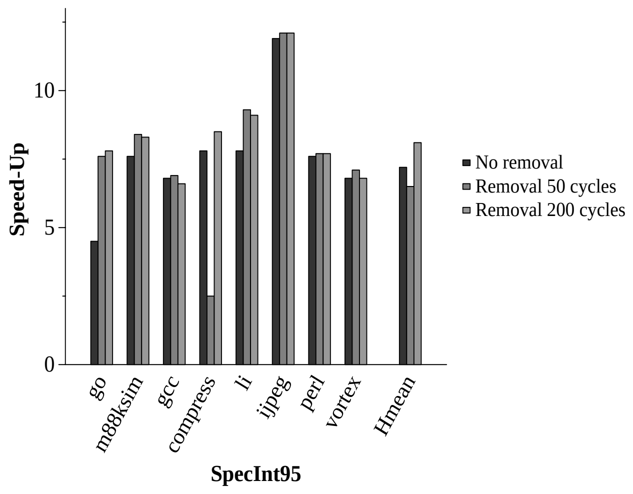


Figure 4.17. Speed-ups achieved by the different spawning pair removal scheme for different number of cycles executing alone.

A proposal to avoid this situation is to dynamically eliminate those spawning pairs that provide small-benefit. 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. This scheme is referred to as the *cancellation scheme*.

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 4.17 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. On average, the speed-up achieved for the 200 cycle scheme is higher than 8, which represents a 10% improvement compared with the non-removal spawning scheme. Most of this improvement comes from the `compress` whereas for the rest of the benchmarks a small slow-down is produced.

Figure 4.18 shows the percentage of spawning pairs removed for each of the benchmarks for different cancellation policies. It can be observed that on average, almost the 20% of the spawning pairs are removed for the 50 cycle scheme. More than 40% of the pairs are removed for `gcc`, `compress` and `li`. For `li`, this pair removal represents almost 20% speed-up whereas for `gcc` the improvement is much lower and for `compress` it causes a drastic drop of performance. It is also remarkable that for `go`, removing 7.6% of the spawning pairs represents almost a 40% speed-up. Regarding the 200 cycle scheme, it reduces the percentage of spawning pairs eliminated and in most of the cases the performance achieved by the previous one too. On average, the percentage of spawning pairs removed is 4%. It can be observed in figure 4.17 that for `go` just removing the 3% of the spawning pairs produces the highest benefits. For `com-`

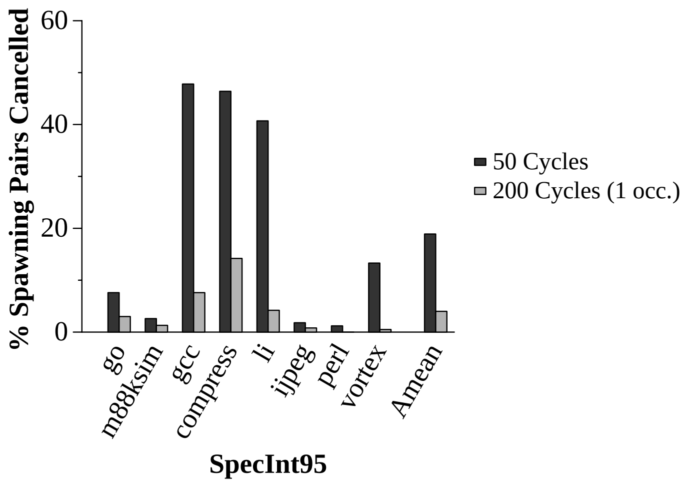


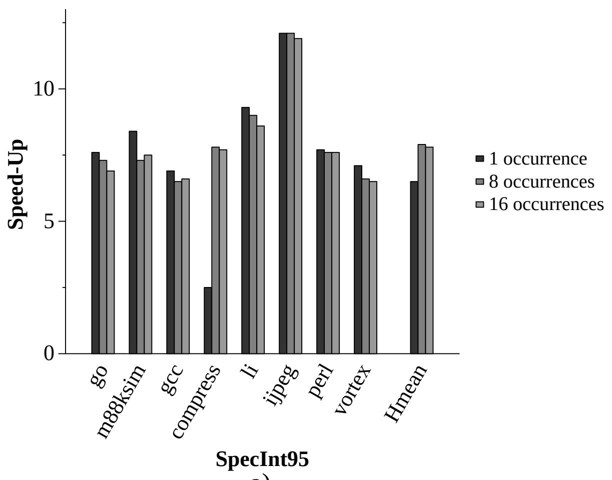
Figure 4.18. Average number of spawning pairs removed by the cancellation policy.

`press`, removing the 14% of the spawning pairs achieves almost a 10% speed-up. For the rest of benchmarks, slight slow-downs are produced.

An alternative way to moderate the removal mechanism is to hold off cancelling a spawning pair until the speculative thread is executing alone a minimum of occurrences. Figure 4.19a shows the performance for a cancelling policy with 50 cycle alone scheme when the number of occurrences is 1, 8 and 16. On average, delaying the removal decision results in an improvement, but it is basically due to the huge improvement achieved for `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.

For the 50-cycle-alone cancellation policy that removes the spawning pairs after a given number of occurrences, figure 4.19b shows that the average number of spawning pairs removed is much lower, 3% for 8 occurrences and 1% for 16. This explains why there is almost no gain in figure 4.19, excepting for `go`, which removes 3.2% of the pairs for the 8-occurrence policy and 2.4% for the 16-occurrence policy.

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. Figure 4.20 shows the speed-up obtained by removing a spawning pair if it is executing with 2 or less threads for 50, 200 and 2000 cycles. For comparison, the cancellation policy that removes the spawning pairs when they are executing alone for more than 50 cycles is also depicted in the figure. This results in a small improvement on average, although most of the benefit comes from `compress`. Some benefit is also reported for `gcc` and `m88ksim`. For the rest of the benchmarks, some slow-downs can be observed, especially for `ijpeg`. Moreover, it can be observed that for all



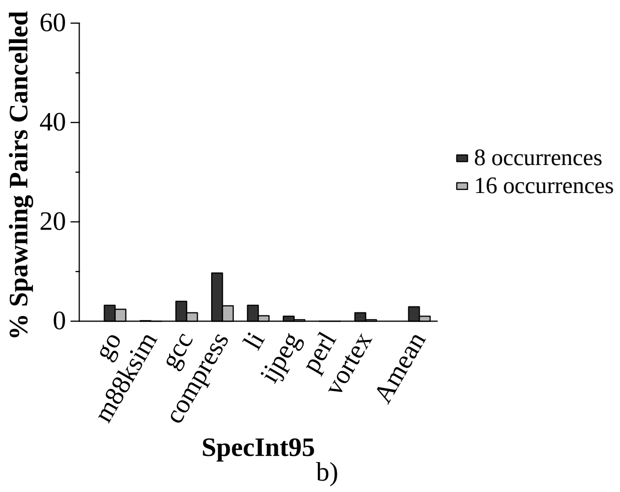


Figure 4.19. a) Speed-ups achieved by the different spawning pair removal scheme for different number of occurrences before cancelling for the 50-cycle removal scheme. b) Percentage of cancelled spawning pairs for the cancellation scheme after 8 and 16 occurrences.

the benchmarks excepting gcc, the most conservative spawning policy, that is, the one that removes later the spawning pairs, presents better results than the more aggressive one. This trend is the opposite to the one showed in figure 4.17a, where on average the more aggressive policies present better results.

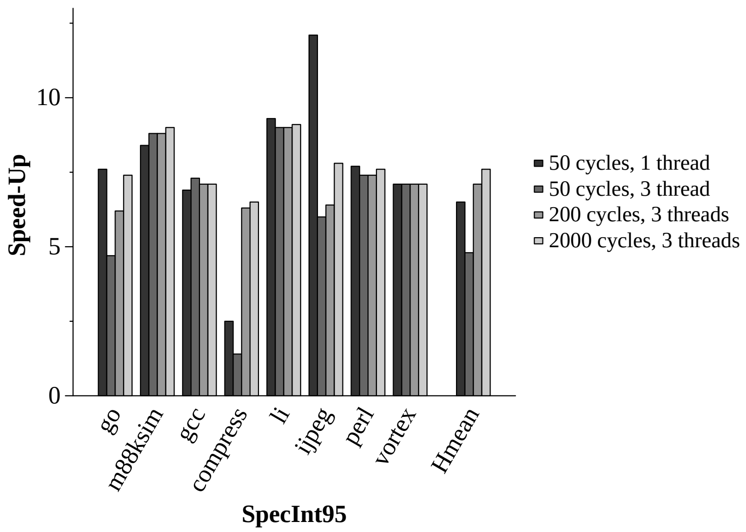


Figure 4.20. Speed-up for the cancellation policies that remove spawning pairs are executing together with 2 or less parallel threads.

Besides, a scheme that reconsiders an eliminated spawning pair after a certain period of time has been evaluated. Results can be observed in figure 4.21. In this case, the cancelling policy removes a spawning pair after executing 50 cycles alone. A spawning pair is reconsidered when their spawning pair has been visited 8 times. For comparison, the non-cancelling policy and cancelling after 50 cycles is shown in the figure. It can be observed that on average this policy provides better results than the previous one, but all the benefit comes from `compress` while some slow-down is showed for the rest of benchmarks.

Finally, to summarize the benefits of the cancellation policy, the average number of active threads for the best cancellation spawning policy are shown in 4.22. The policy corresponds to cancelling the spawning pair the first time the associated thread is executing 50 cycles alone. It can be observed that on average all the benchmarks increase their average number of active threads per cycle and that, excepting for `compress` and `m88ksim`, the percentage of time the clustered processor is executing a single thread has been reduced. In the case of the Motorola simulator, this significant increment in the percentage of single-threaded execution is more than compensated by the significant increment for the parallel execution of 7 and 8 threads.

The case of `compress` is different since the cancellation policy is too aggressive and most of the spawning pairs are eliminated so almost no speed-up is achieved. This situation is clearly reflected in the figure 4.22 where it can be observed that the percentage of single-threaded execution grows from 14% to almost 90%. Figure 4.17a has showed that some additional benefits can be obtained when considering a less aggressive cancellation policy for this benchmark. In fact, delaying the removal of a spawning pair

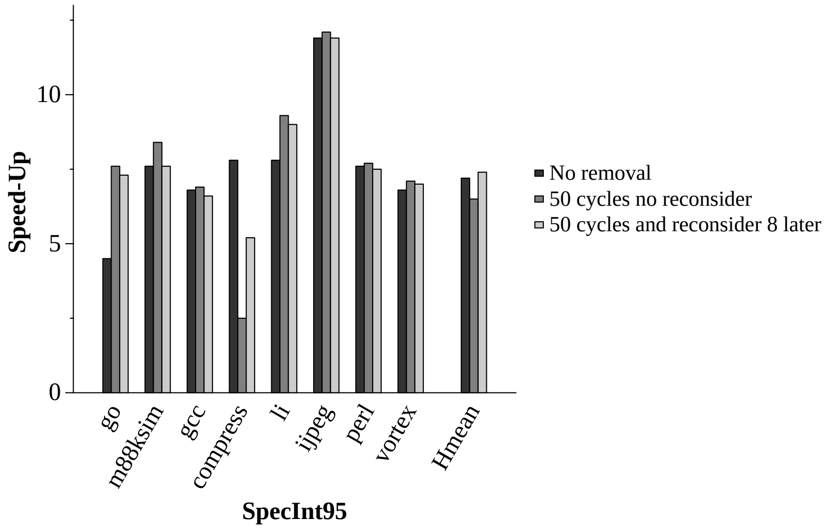


Figure 4.21. Speed-up of the cancellation policy that reconsiders an eliminated spawning pair after visiting it 8 times.

until it has been executing alone for 200 cycles provide a 10% speed-up compared with the non-cancellation policy. For this cancellation policy, the time distribution of the active threads is depicted in figure 4.23. It can be observed an important increment in the percentage of single-threaded execution, but quite far from the one obtained for the 50-cycle cancellation policy. However, as for `m88ksim`, the average number

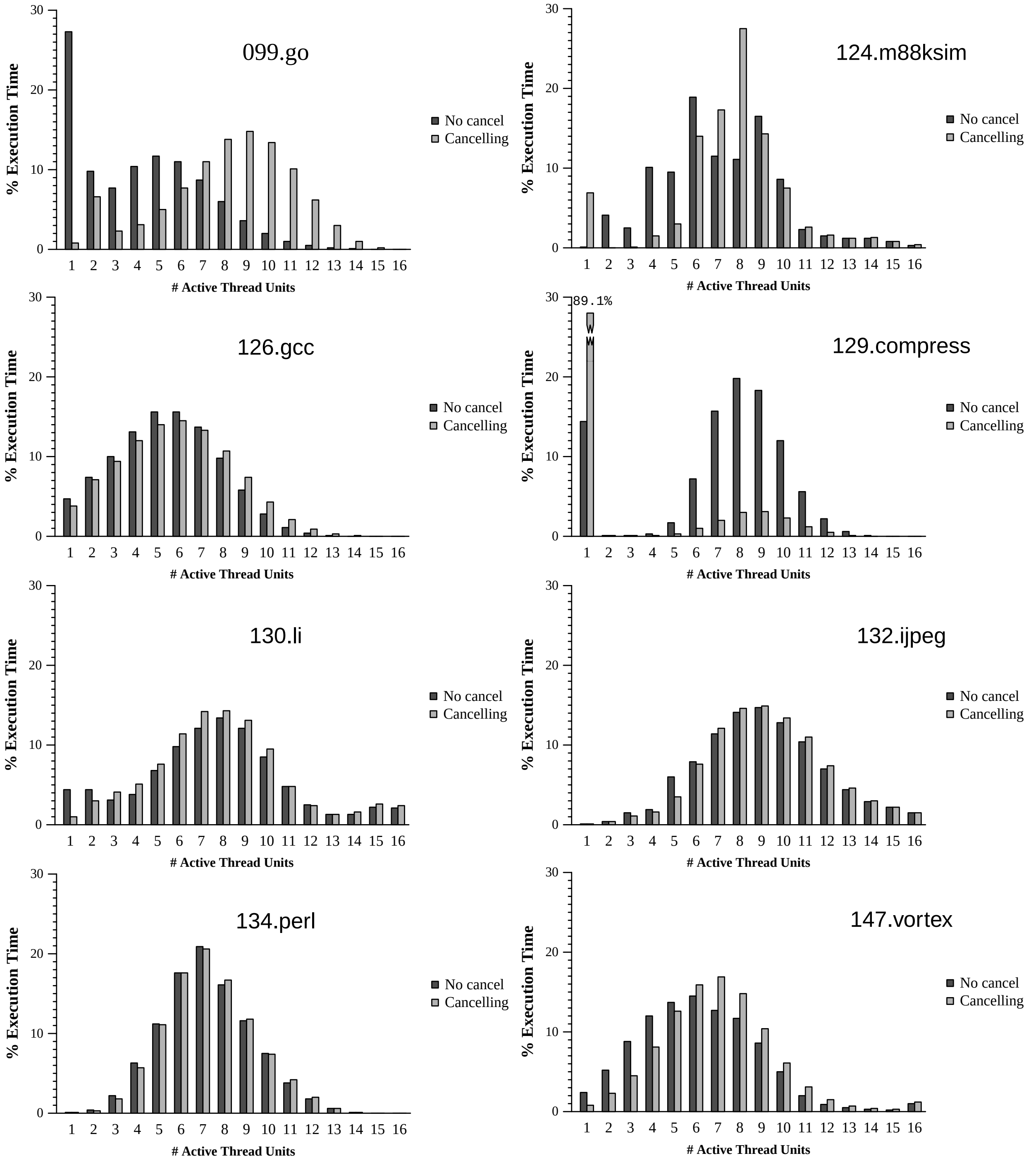


Figure 4.22. Thread Unit Utilization for the Profile-based spawning scheme with the call-return pairs for the best cancellation policy (50 cycles executing alone).

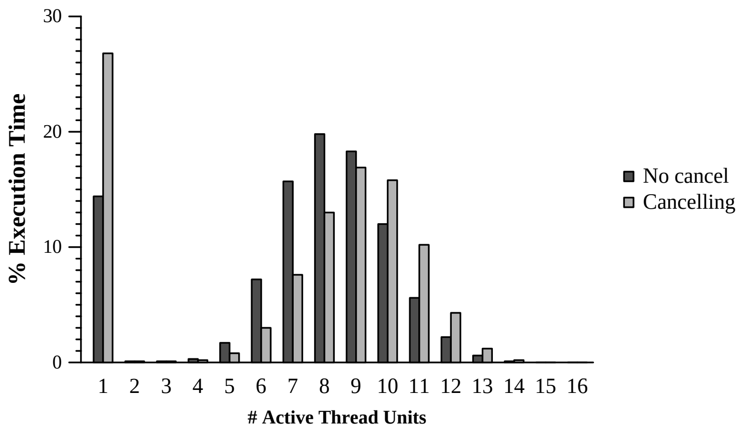


Figure 4.23. Thread unit utilization for `compress` when the cancellation policy is applied after 200 cycles of execution alone.

of active threads per cycle is higher than the one obtained by the non-cancelling policy (7.6 for the cancellation policy and 7.2 for non-cancelling) which results in a better overall performance.

4.5.3.2. Reassign Spawning Policies

Figure 4.24 shows the percentage of spawning pairs (not taking into account the call-return pairs) that create at least a speculative thread. It can be observed that on average only 42% of the spawning pairs are used.

There are different reasons that can cause that a spawning pair never tries to create a new thread. One of them is that when the spawning point is reached, all the thread units are used by other less speculative threads. Another reason is that whenever a thread reaches a spawning point, there is already a more specu-

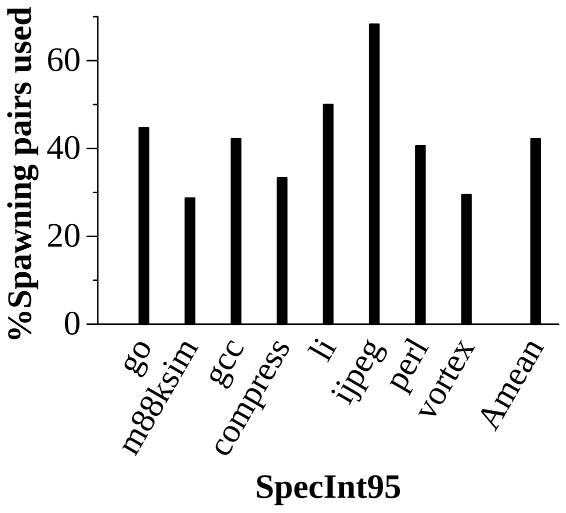


Figure 4.24. Percentage of spawning pairs that create speculative threads (not taking into account the call-return pairs).

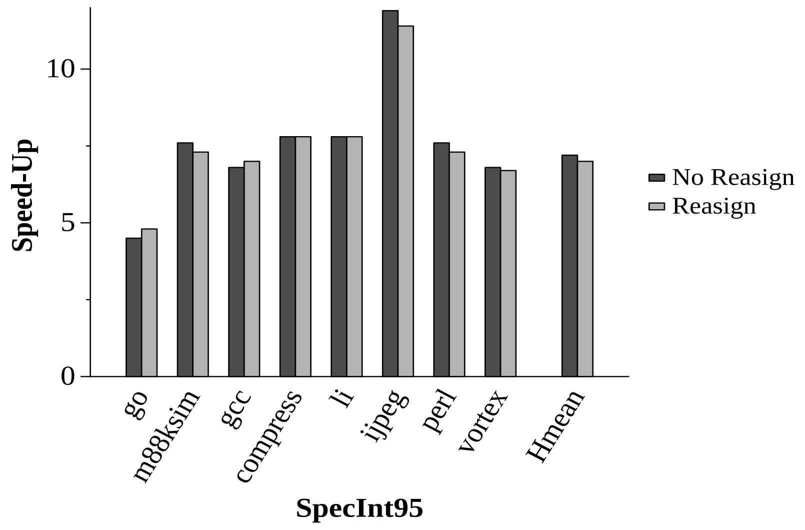


Figure 4.25. Speed-up for the reassign policy.

lative thread started in the same dynamic instance of the control quasi-independent point. That is, as speculative threads are disjunct, threads executing the same pieces of the dynamic instruction stream are not allowed. This situation can be detected with the order predictor explained in Chapter 2: a thread that starts in a given control quasi-independent point can never be compared with another thread that starts at the same control quasi-independent point.

Thus, an alternative policy may be considered. That says whenever a spawning point is 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 criterion. This family of spawning policies will be referred to as *reassign spawning policies*.

Figure 4.25 shows the performance obtained by this reassign policy. On average, reassigning produces a 3% slow-down compared with no reassign.

On the other hand, these reassigning approach can be used with a different flavour. Whenever a spawning pair is removed, the next most convenient pair with the same spawning point can be considered for the next time the spawning point is reached.

The result of these modifications are shown in Figure 4.26, together with the cancellation spawning scheme that just considers a single spawning pair per spawning point. It can be observed that the results are again worse for the reassign policy and the degradation is quite significant.

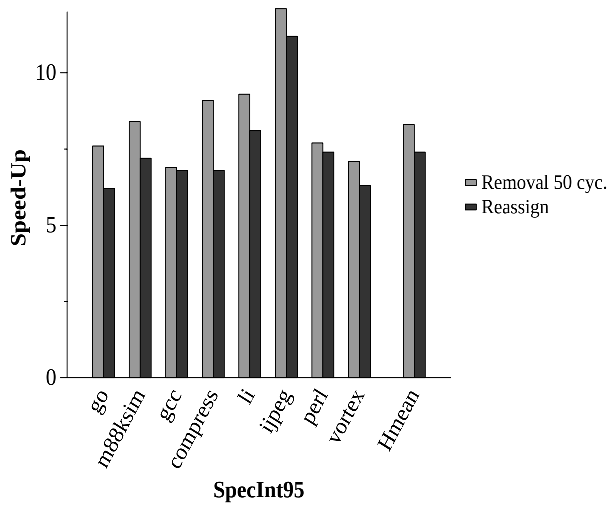


Figure 4.26. Speed-up of the reassign spawning policy compared with the 50-cycle removal policy (for compress, 200 cycles).

The main reason for the performance degradation suffered for the reassign spawning policies is the fact that whenever a control quasi-independent point cannot be chosen, the next control quasi-independent point is usually too close. Note that the criterion to order the control quasi-independent points for a given spawning point is maximizing the distance between both points. Then, it is very likely that if the profile-based spawning scheme has chosen a given basic block to be the control quasi-independent point, the next candidate in order will be the previous basic block. Thus, when the speculative thread is created at such new control-quasi independent point, few instructions later it will stop its execution since it has reached the starting point of another speculative thread (the one that has prevented the creation of the speculative thread at the primary control quasi-independent point). That is, the processor is generating very small threads that do not provide any benefit. Even worse, they may cause to waste the resources of the processor.

4.5.3.3. Minimum Thread Size Spawning Policies

Figure 4.27 shows the average number of instructions that are between the spawning and the control quasi-independent point. This average is almost 100 instructions. On the other hand, the average number of instructions that are executed on average by each thread unit is close to 35 instructions and is about the third part of the expected thread size extracted from the profile analysis. The most remarkable case is the average thread size for benchmarks such as `m88ksim` (21 instructions), `gcc` (30 instructions), `compress` (24 instructions), `li` (17 instructions) and `perl` (29 instructions). Note that the minimum distance between the spawning and the control quasi-independent point should be at least 32 instructions, but clearly it is much lower.

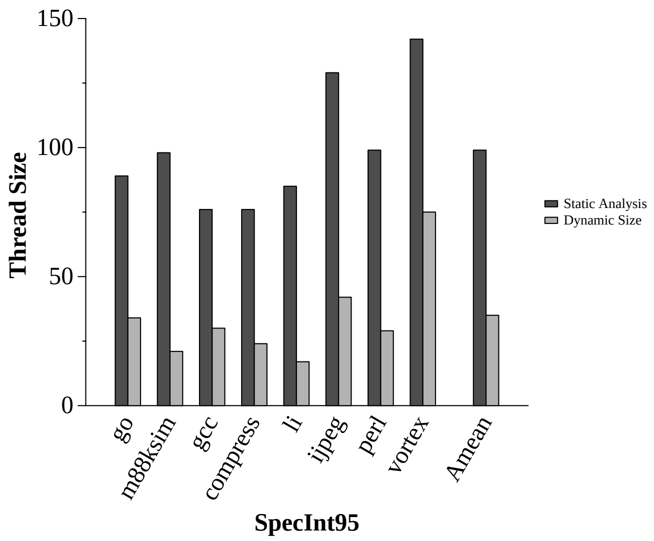


Figure 4.27. Average number of instructions between the spawning and the control quasi-independent point statically compared with the number of dynamic instructions executed at each thread unit.

The reason for that divergence between the expected and the real sizes of the threads is due to the fact that the profile analysis considers each spawning point individually and does not take into account any possible interaction among speculative threads. Figure 4.28 illustrates this problem. Assume that a dynamic instruction stream as the one shown in the figure is being executed in TU$_i$. When Spawning Point 1 (SP1) is reached, a new speculative thread is created on an idle thread unit (TU$_j$) starting at Control Quasi-Independent Point 1 (CQIP1) and both threads proceed in parallel. Later on, the non-speculative thread reaches

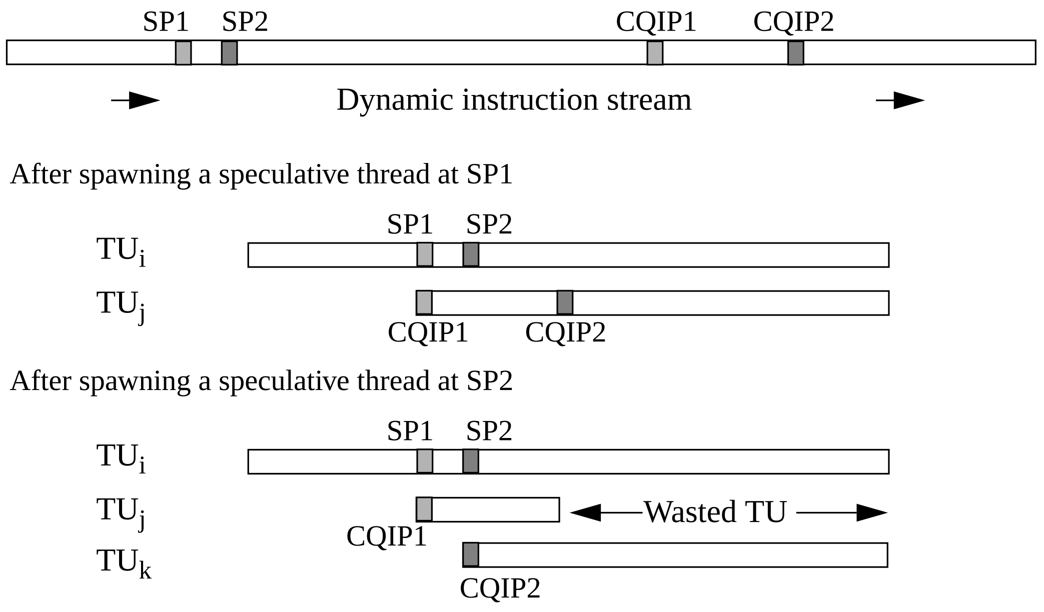

SP1
SP2
CQIP1
CQIP2
Dynamic instruction stream
After spawning a speculative thread at SP1
TUi
TUj
SP1
SP2
CQIP1
CQIP2
After spawning a speculative thread at SP2
TUi
TUj
TUk
SP1
SP2
Wasted TU
CQIP1
CQIP2

Figure 4.28. Example that justifies that thread sizes are lower than the expected.

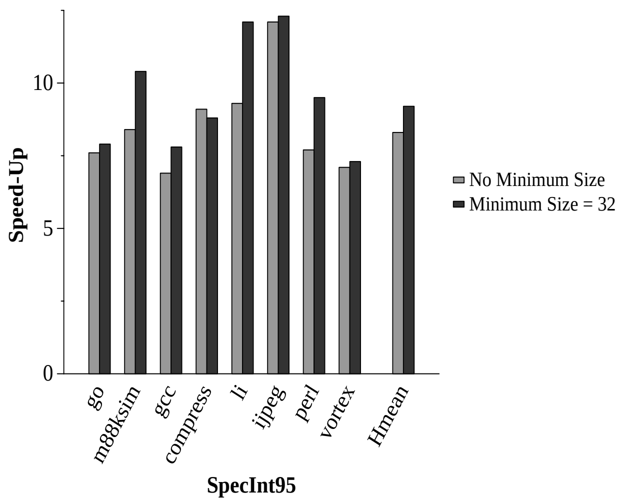


Figure 4.29. Speed-up achieved when a minimum thread size is considered to spawn new speculative threads.

the Spawning Point 2 (SP2) and creates a new thread on another idle thread unit (TU_k) starting at their corresponding Control Quasi-Independent Point 2 (CQIP2) and the three threads continue executing in parallel. When the thread that is being executed in TU_j reaches CQIP2, it finishes. As this thread has not become the non-speculative yet, it will have to wait for the finalization of the thread that is executing at TU_i to commit its values. During this time, TU_j is idle but not free and no other speculative threads can be spawned at this thread unit.

To avoid this undesirable situation, the processor may monitor the number of instructions that are executed by each thread unit. If this number is lower than a certain threshold (i.e. 32 instructions), the corresponding spawning pair is removed. We refer to this spawning policy as the *minimum thread size spawning policy.*

Figure 4.29 shows the performance for the spawning policy that removes those spawning pairs whose speculative threads execute less than 32 instruction in 8 consecutive occurrences (in addition to the traditional removal policy). For comparison, the left bar shows the speed-up obtained for the conventional removal policy (a 50-cycle threshold is considered for all the benchmarks, except for `compress`, which is set to 200 cycles). It can be observed that enforcing the minimum size of the threads and removing those spawning pairs that execute less instructions than 32 outperforms the conventional removal policy by 10% and achieves an average speed-up slightly higher than 9.

Some other different criterion to decide when a spawning pair must be removed (varying the number of occurrences, total number of occurrences instead of consecutive ones, etc....) have also been evaluated

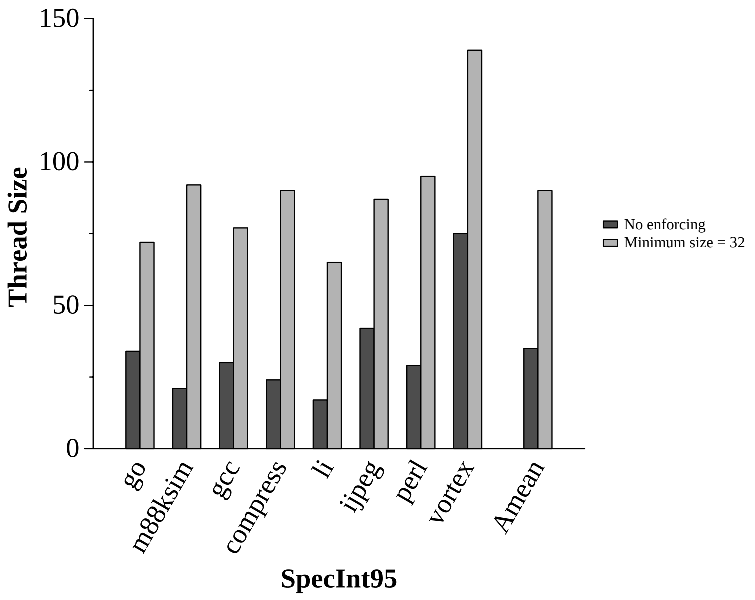


Figure 4.30. Thread size for the conventional removal policy and for the minimum thread size spawning policy.

and the results were similar or worse. Specially remarkable is the slow-down suffered (more than 35%) by the spawning policy that removes a spawning pair at the first time it executes less than 32 instructions.

The impact on the thread size of the minimum thread size spawning policy can be observed in figure 4.30. It can be observed that the average number of instructions is increased almost three times.

4.5.4. Profile-based vs. Heuristics

In the previous subsection, a spawning policy based on combining the different heuristics was proposed. In this spawning scheme, speculative threads are created at loop-iterations, loop-continuations and subroutine continuations. The results obtained by this partitioning mechanism were presented in subsection 4.4.5 and shown quite significant speed-ups. In this section, a different approach for obtaining speculative threads has been proposed. This scheme is based on a profile analysis of the code that selects the spawning pairs depending on their features, and some mechanisms to guarantee a better utilization of the thread units.

In order to compare both families of spawning schemes, the same dynamic optimizations considered for the profile-based spawning scheme are taken into account for the combined one. Then, if a speculative thread spawned by either an iteration, loop continuation or subroutine executes less than 32 instructions per iteration, this spawning pair is disabled for the future.

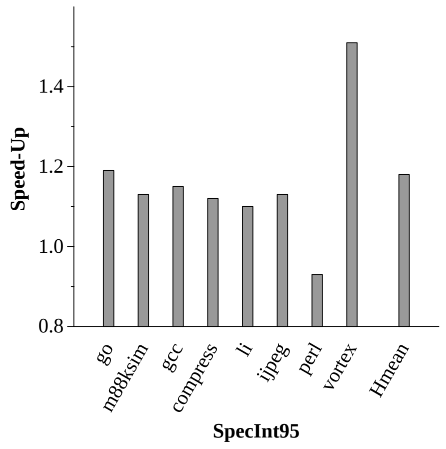


Figure 4.31. Speed-up of the profile-based spawning policy over the combination of heuristics.

Figure 4.31 shows the speed-up obtained by the profile-based spawning policy over the combination of loop-iteration, subroutine-continuation and loop-continuation spawning schemes when perfect register value prediction is considered for both spawning schemes. It can be observed that on average the improvement is close to 20%, being quite high for `vortex` and higher than 10% for the rest of the benchmarks (except for `perl`, which suffers a small slow-down (8%)). This fact is due to the workload imbalance.

Thus, we can conclude that the off-line analysis provides significant improvement over the heuristic-based approach.

4.6. Related work

The most usual spawning scheme used by speculative multithreaded architectures in the literature is based on spawning speculative threads at loop iterations, either at run-time by means of hardware mechanisms or with compiler support.

Examples of this last group that relies on the compiler for the partitioning process are the SPSM[14] and the Superthreaded architectures[76]. In this last architecture, some code reordering is performed by the compiler to obtain a fast computation of the dependent values and start the execution of following threads. In addition to this speculative multithreaded architectures, lots of works on-chip multiprocessors with support for speculative multithreading has chosen loop iterations as a main source for speculative threads such as the works from the I-ACOMA[5][34] group, the STAMPede[71][72] among others. In all cases, programs are split by the compiler.

A different heuristic is used by the Dynamic Multithreaded Processor[2] which speculates on loop and subroutine continuations and allows the speculative threads to be created in any order.

On the other hand, Trace Processors[59] partition a sequential program into almost fixed-length traces which is specially suited to maximize the workload balance among the different thread units with the help of the trace cache[60].

A different proposal to divide the program into threads was done by Codrescu *et al*. The MEM-slicing scheme[9] is also based on profile analysis, but the spawning algorithm spawns a new thread when a memory instruction is reached, and the speculative thread starts at some distance where values are expected not to depend on that memory value.

A more complex approach was taken by the Multiscalar[16][68] architecture. In that architecture, the compiler is responsible for dividing the code into tasks[84]. 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. The way tasks are built is incremental. The compiler starts at one basic block and looks for other basic blocks to be added to such task until it becomes too large or it has too may predecessors or successors or too many data dependences.

Finally, some works comparing different spawning policies have been performed for on-chip multiprocessors[9][53]. The spawning policies considered are based on assigning speculative threads to loop iterations, loop continuations and subroutine continuations.

4.7. CONCLUSIONS

In this chapter, the impact of the spawning schemes on the overall performance of the speculative multithreaded processors has been analyzed. Two different approaches for partitioning programs into threads have been studied. In the first one, spawning pairs are obtained through several heuristics that basically look for common program constructs. Examples of this partitioning policy are those that spawn threads at loop-iterations, at loop-continuations and at subroutine continuations.

The second family of partitioning mechanisms is based on an off-line analysis with support of profiling. This approach looks for threads that accomplish some properties that usually make them more effective. Schemes that maximizes the distance between the spawning and the control quasi-independent point, maximize the number of independent instructions between the spawner and the spawned thread and maximizes the number of independent and predictable instructions between the threads are proposed and evaluated.

Both families have been compared, showing that the off-line analysis provides better results on average than the one based on simple heuristics (almost 20% higher with a 16-thread unit speculative multithreaded processor).