Thread Partitioning and Value Prediction for Exploiting Speculative Thread-Level Parallelism

Pedro Marcuello, Antonio González, *Member*, *IEEE Computer Society*, and Jordi Tubella

Abstract—Speculative thread-level parallelism has been recently proposed as a source of parallelism to improve the performance in applications where parallel threads are hard to find. However, the efficiency of this execution model strongly depends on the performance of the control and data speculation techniques. In this work, several hardware-based schemes for partitioning the program into speculative threads are analyzed and evaluated. In general, we find that spawning threads associated to loop iterations is the most effective technique. We also show that value prediction is critical for the performance of all of the spawning policies. Thus, a new value predictor, the increment predictor, is proposed. This predictor is specially oriented for this kind of architecture and clearly outperforms the adapted versions of conventional value predictors such as the last value, the stride, and the context-based, especially for small-sized history tables.

Index Terms—Speculative thread-level parallelism, value prediction, branch prediction, thread spawning policies, clustered architectures.

✦

1 Introduction

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

The amount of ILP that these processors can exploit is closely related to the size of the instruction window among other issues. However, the effective size of the instruction window is limited by the branch prediction accuracy since the number of correctly speculated control-flow instructions depends on the number of consecutive branches that have been correctly predicted. This is due to the sequential nature of the fetching mechanism of superscalar processors since a single mispredicted branch prevents the instruction window from growing beyond the branch, this being a significant limitation in nonnumeric programs, which have many difficult-to-predict branches. As a result of that, the performance achieved by such processors is as far away from the peak performance they can get and it results in a poor usage of the resources of the processor.

In order to make better use of the resources of the processors, some multithreaded architectures, such as the Simultaneous Multithreading Processors [29], propose executing different threads coming from either parallelized applications or multiple applications. The select and the wake-up logic of the processor can search for independent instructions among all threads running in parallel. These kinds of architectures usually provide a better usage of the resources and a higher throughput, but it may increase the execution time of sequential applications (compared with their execution alone in a processor with the same amount of resources) since the resources are shared with other programs that are running simultaneously [28].

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

These microarchitectures provide support for multiple contexts and appropriate mechanisms to forward or to predict values produced by one thread and consumed by another. They differ in how programs are split into threads. In several proposals, such as the Multiscalar [6], [24], the SPSM architecture [5], and the Superthreaded architecture [30], the compiler splits the program into threads, whereas others rely only on hardware techniques. Examples of the latter group are the Dynamic Multithreaded Processor [1] and the Clustered Speculative Multithreaded Processor [14], [15].

All these works have shown that speculative thread-level parallelism has significant potential to boost performance. Most of them use different heuristics to partition a sequential instruction stream into speculative threads. Little insight exists to explain the source of the advantages for each particular partitioning approach and its interaction with value prediction and other microarchitectural components. In all cases, significant benefits have been reported,

• *The authors are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Jordi Girona, 1-3 Mòdul D6, 08034 Barcelona, Spain. E-mail: {pmarcue, antonio, jordit}@ac.upc.es.*

Manuscript received 14 june 2001; revised 19 July 2002; accepted 19 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114350.

but the absolute figures are not comparable due to very different architectural assumptions.

In this work, the performance benefits of different approaches of thread-level parallelism are studied. Various speculative thread spawning policies—loop iterations, loop continuations, subroutine continuations—are analyzed. Results for the SpecInt95 will show that spawning threads at loop iterations with an unrestricted thread ordering policy achieves a 3.1 speed-up for 16 thread units for perfect input register value prediction and the degradation suffered by this mechanism when a realistic increment value predictor (which is a special value predictor especially targeted for this kind of architecture) is considered on this architecture is only 10 percent, on average, being higher than any other spawning policy and value predictor considered. We also show that value prediction is a must for this type of microarchitecture and benchmarks. Without value prediction, a speculative multithreaded processor has a performance very close to a conventional superscalar one.

Below, Section 2 reviews related work. Different speculative thread spawning policies are described in Section 3 and their performance potential is analyzed in Section 4. The impact on the performance of several key parts of the processor is then analyzed, starting with the branch prediction scheme in Section 5 and the value predictor in Section 6. The overall processor performance is analyzed in Section 7. Conclusions are summarized in Section 8.

2 Related Work

Several multithreaded architectures providing support for thread-level speculation have been proposed. Pioneer work on this topic was the Expandable Split Window Paradigm [6] and the follow-up work, the Multiscalar [24]. In this microarchitecture, the compiler is responsible for partitioning the program into threads based on several heuristics that try to minimize the data dependences among active threads or maximize the workload balance, among other compiler criteria [32].

Other architectures, such as the SPSM [5] and the Superthreaded architectures [30], also rely on the compiler to split the program into threads, but, in these cases, threads are assumed to be loop iterations instead of the more complex analysis of the Multiscalar compiler. A common feature of these architectures is the way to deal with data dependences since all of them stall the execution of the consumer of an interthread data dependent instruction until the value is produced and forwarded by the producer thread and no mechanisms for value prediction are considered.

On the other hand, some other architectures try to exploit thread-level parallelism speculating on threads dynamically created by the processor without any compiler intervention. The Speculative Multithreaded Processor [14] and its successor, the Clustered Speculative Multithreaded Processor [15], identify loops at runtime 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, but it speculates on loop and subroutine continuations. Moreover, the architectural design of the processor allows for out-of-order thread creation, which requires communication among all hardware contexts.

Trace Processors [20], [31] also exploit a certain kind of speculative thread-level parallelism. Its mechanism to split the sequential program into almost fixed-length traces is especially suited to maximize the workload balance among the different thread units by the help of the trace cache [21].

These last three multithreaded architectures provide support for value prediction in order to avoid serialization among concurrent threads, but the value predictors considered in each of them are very different. Whereas the mechanism provided in the Dynamic Multithreaded Processors is based on copying the register file of the parent thread into the child register file, a mechanism that can be quite suitable for spawning threads at subroutines, the Trace Processor and the Clustered Speculative Multithreaded architecture use more complex and general predictors, like the context-based and the increment predictors.

Several works on speculative thread-level parallelism on multiprocessor platforms have been performed. The I-ACOMA group [3], [12] and the STAMPede group ([27], [26] among others) have proposed compiler-based techniques to speculatively exploit thread level parallelism. In both works, loops are considered as the main source of speculative parallelism and the constraints for parallelizing them are relaxed, especially those related to memory disambiguation. In these architectures, the impact of memory value prediction is also studied, providing small benefits if the cost of misprediction is high or if some of those predicted values can be eliminated, such as loop induction variables. Some other proposals in this area can be found in [10], [11].

A different approach for on-chip multiprocessor for exploiting speculative thread-level parallelism is taken by the Atlas multiprocessor [4]. Here, the different processing elements are interconnected by means of a bidirectional ring topology and to which support for thread and value speculation has been added. Speculative threads are obtained by means of the MEM-Slicing algorithm, which, instead of spawning threads at points of the program with high control independence, spawns threads at memory-access instructions.

Finally, some work comparing different spawning policies has been done in [4], [19]. In both cases, reported results show that spawning threads at subroutines achieves higher performance than spawning at loop iterations or loop continuations. However, both studies consider as their baseline architecture an on-chip multiprocessor where each processing unit was single-issue and instructions were issued in program order. With these assumptions, the existing interactions between fine and coarse-grain parallelism are not considered.

Some insight into the behavior of Speculative Multithreading has been done in [34]. In that work, the impact of several parts of the microarchitecture is evaluated for the module-level execution model, but it is not focused on evaluating different spawning mechanisms.

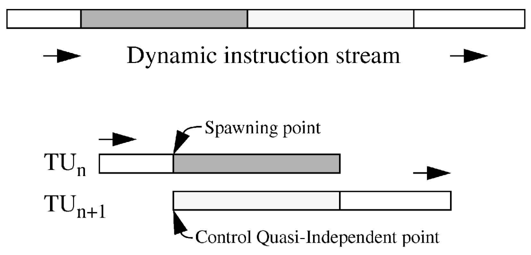

Fig. 1. Spawning and control quasi-independent points.

3 Dynamically Exploiting Speculative Thread-Level Parallelism

For performance reasons, speculative multithreaded processors could try to split a program into threads in such a way that they are control and data independent since, then, the processor would be able to execute them concurrently. Unfortunately, these kinds of threads are very hard to find, especially in nonnumerical programs.

The approach used to identify the points where the speculative threads will be spawned becomes one of the critical issues of this kind of architectures. Since spawning threads only at control independent points strongly constrains the potential thread-level parallelism, speculative multithreaded processors create new threads at points of the program that are very likely to be executed in the near future, although the path to reach them may vary a lot and may be rather unpredictable. In this paper, these points will be referred to as control quasi-independent points. Besides, speculative threads should have few dependences with previous threads in the program order that are executed in parallel with or later that itself. Fig. 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 continues executing instructions up to the Control Quasi-Independent Point, whereas the following Thread Unit spawns a new thread, speculatively starting at this instruction.

Candidates, among others, to be control quasi-independent points are:

- The first instruction in static order of a loop since it is very likely to be executed in following iterations almost independently of the outcome of the branches inside the loop. Here, the spawning point and the control quasi-independent point will be the target of a backward branch.
- The following instruction in static order after a backward branch that closes a loop since it is very likely to be executed when all the iterations have been performed and it is also almost independent of the control-flow inside the loop. Here, the spawning point will be the target of the backward branch and the control quasi-independent point the following instruction in static order of a backward branch.
- The following instruction in static order of a subroutine call since it is very likely to be executed when the subroutine returns, independently of the path followed inside it. Here, the spawning point will be the call instruction and the control quasi-independent point the following instruction in static order.

In this paper, the performance potential of these spawning policies based on these control quasi-independent points is evaluated. Note that other control quasi-independent points can be assumed (e.g., [17]). In this paper, the spawning policies associated with the above three types of control quasi-independent points will be referred to as *loop-iteration*, *loop-continuation*, and *subroutine-continuation*, respectively.

3.1 Experimental Framework

For the experiments in this work, we will consider a Clustered Speculative Multithreaded Processor [15]. This microarchitecture is made up of several thread units, each one being similar to a superscalar processor core. Each thread unit concurrently executes different threads of a sequential program. These threads are dynamically obtained by a control speculation mechanism based on identifying spawning and control quasi-independent points creating a new thread every time a spawning point is reached and there are thread units available. 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. A clustered design is selected due to its scalability.

There are multiple ways to interconnect the different thread units that form the multithreaded processor. Two extremes scenarios are: 1) a ring topology interconnection network in which one thread unit can only send data to the immediate successor and can only receive data from its immediate predecessor and 2) a full-connectivity architecture in which any thread unit can communicate with anyone else. Other scenarios could also be considered. Fig. 2 shows a Clustered Speculative Multithreaded Processor fully interconnected. The speculation engine is responsible for allocating the new spawned threads in those thread units that are available as well as all the tasks related to thread initialization.

Performance statistics are obtained through trace-driven simulation of the whole SpecInt95 benchmark suite. SpecFP95 has not been considered for its evaluation in this paper since conventional compilers can easily extract nonspeculative thread-level parallelism in significant parts of those programs. Some other authors [3] have investigated schemes to speculatively parallelize some portions of these codes that are hard to parallelize.

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 [25]. For the statistics, we simulated 300 million instructions after skipping initializations. The programs are executed with the *ref* input data since they reflect a more realistic workload, in particular, for some parameters such as the number of loop iterations.

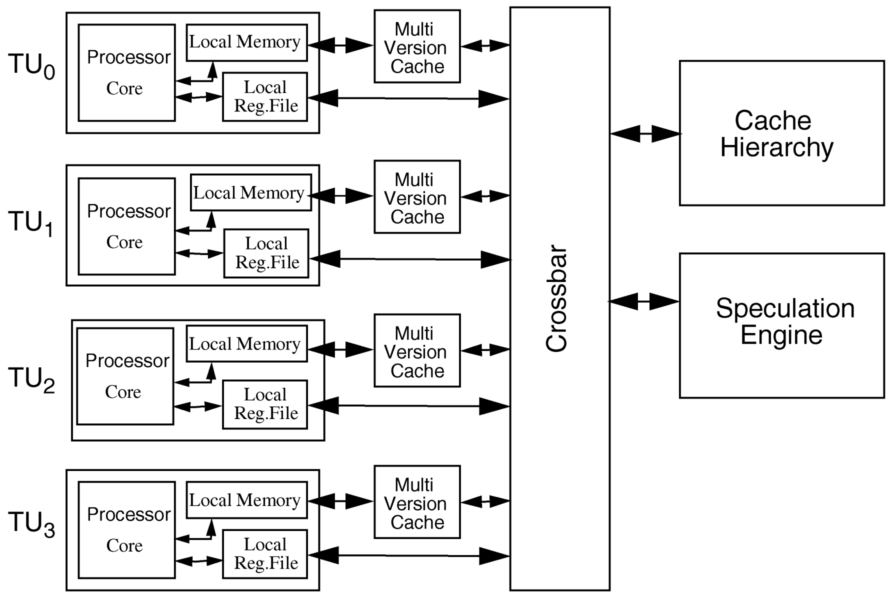

Fig. 2. A clustered speculative multithreaded processor with four thread units fully interconnected.

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

- Fetch: up to four instructions per cycle or up to the first taken branch, whichever is shorter.
- Issue bandwidth: Four instructions per cycle.
- Functional Units (latency in brackets): two simple integer (1), two memory address computation (1), 1 integer multiplication (4), two simple FP (4), one FP multiplication (6), and one FP division (17).
- Reorder buffer: 64 entries.
- Local branch predictor: 14-bit gshare. Local branch prediction tables are assumed to be copied from the parent thread to the new spawned thread at spawning time. Other policies are evaluated in Section 4.2.
- 32 KB nonblocking, 2-way set-associative local, L1 data cache with a 32-byte block size and up to four outstanding misses. The L1 latencies are three cycles for a hit and eight cycles for a miss.

We will consider two different spawning ordering policies: a sequential thread ordering policy and an unrestricted ordering policy. For the former, threads are created in a sequential order in such a way that one thread cannot be spawned between two current threads. The latter policy allows the creation of new threads without any constraint.

In order to evaluate the potential of this architecture, the following section studies the performance on an ideal scenario. In this first analysis, three different approaches to deal with data dependences among instructions in different threads (interthread dependences for short) are considered. In the first model, all values corresponding to interthread 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, interthread dependent register values are assumed to be correctly predicted, but interthread dependent memory values must be forwarded from the producer to the consumer and the delay has been estimated as three cycles. This model is referred to as *perfect register prediction*. Finally, the last model considers that the interthread dependences cause a serialization between the producer and the consumer instructions and is called the *synchronization* model. In this last model, the delay of forwarding the model from the producer thread unit to the consumer is assumed to be three cycles for memory values and one cycle for registers. In Section 6, realistic value predictors will be considered and their implications on the performance will be analyzed. Some other approaches could also be considered between the two extreme scenarios, that in which all dependent interthread register and memory values are always correctly predicted and that in which all interthread dependent values have to wait for their computation in the producer thread.

In the next figures, the cost of spawning threads is assumed to be zero. In the next section, the impact of assuming an 8 and 16-cycle initialization overhead is analyzed. Memory dependence violations are detected by means of a MultiVersion Cache which implements a cache coherence protocol based on the Speculative Versioning Cache [9].

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

4 Dynamic Thread Partitioning

In addition to the thread spawning policy, the performance of a speculative multithreaded processor strongly depends on the underlying hardware architecture, in particular, the interconnection capabilities among different hardware contexts. Here, we consider two extreme scenarios: a ring topology interconnection network and a full-connectivity architecture. Obviously, in the case of a ring topology

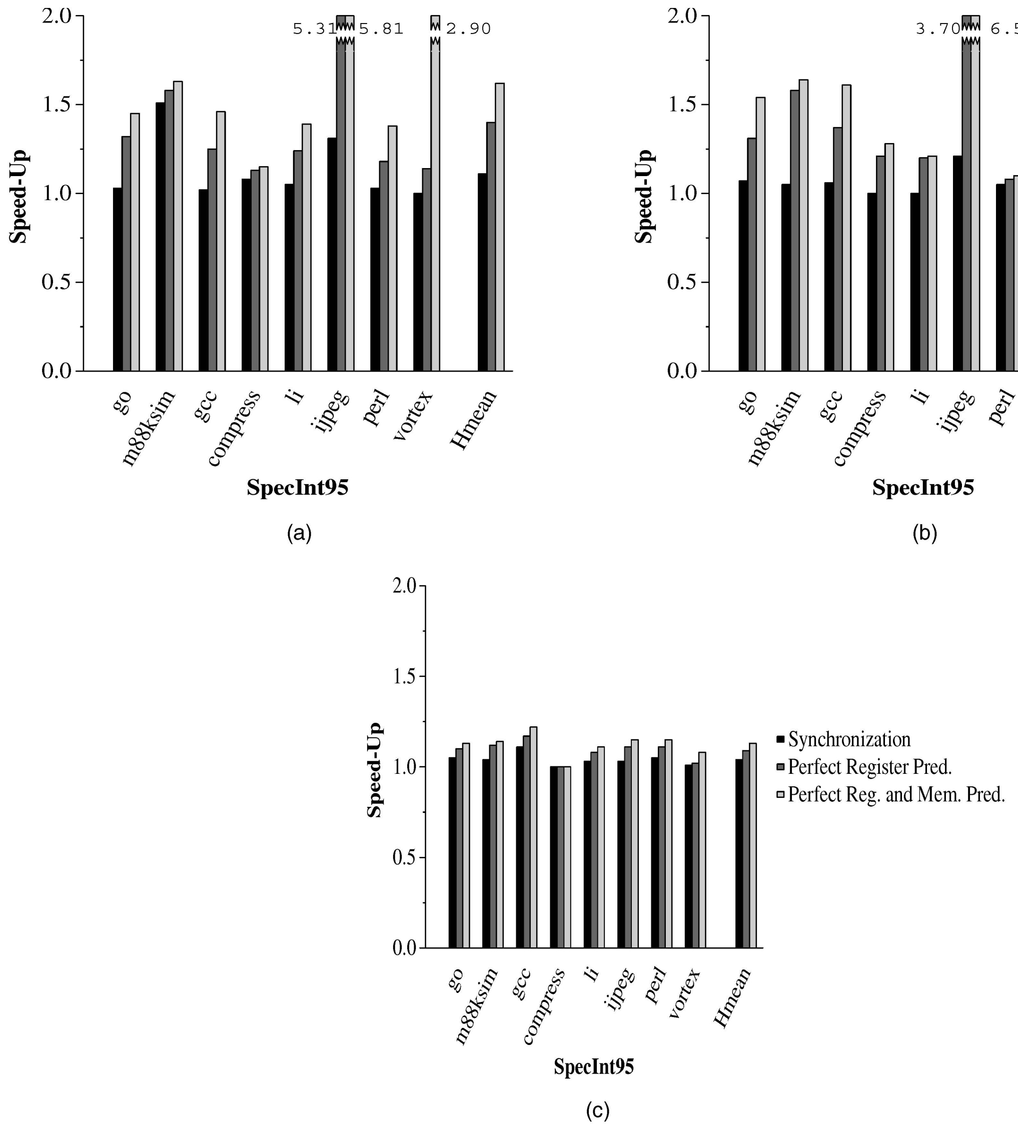

Fig. 3. Speed-ups for the three different spawning policies. (a) Loop-iteration. (b) Loop-continuation. (c) Subroutine-continuation for the sequential thread ordering scheme.

interconnection, the penalty due to forwarding register values from the producer thread unit to the consumer one is one cycle per hop.

4.1 Sequential Thread Ordering

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

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

Fig. 3 shows the speed-up of each benchmark over a single-threaded execution, as well as the harmonic mean, for a 16 thread unit configuration. The three bars for each program correspond to the three different approaches to deal with interthread dependences (synchronization, perfect prediction of register values, and perfect prediction of register and memory values). We can observe that the speed-up achieved for the loop-iteration and loop-continuation spawning policies is somewhat higher than for the subroutine-continuation scheme and it is especially significant for `ijpeg`. The reason is that, in these two models, there are more threads that are created in their sequential order than for the subroutine-continuation model and, thus, they generate fewer squash operations. The low performance of the subroutine-continuation scheme is due to the thread ordering scheme imposed by the ring topology, which restricts the thread speculation mainly to leaf subroutines. To illustrate this problem, let us assume a

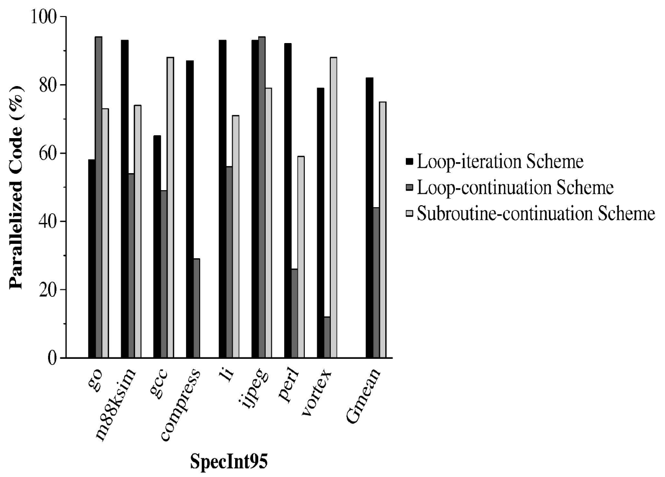

Fig. 4. Percentage of code executed in parallel with other threads for each spawning policy.

subroutine A, which is being executed in thread unit 1, that calls subroutine B, which in turn calls subroutine C. When the call to subroutine B is found, a speculative thread that executes the continuation of B (rest of subroutine A) is spawned and allocated to thread unit 2, whereas thread unit 1 proceeds with the execution of subroutine B. Then, when subroutine B calls subroutine C, thread unit 1 proceeds with subroutine C and the speculative thread corresponding to the continuation of C (rest of B) is allocated to thread unit 2, which causes the thread running the continuation of B to be squashed.

Note that the subroutine-continuation scheme does not improve the single-threaded execution of `compress` since the evaluated part of the program does not contain any subroutine call that returns to the following instruction in static order. Therefore, for the average calculation of this scheme, `compress` is not considered

4.2 Unrestricted Thread Ordering

In opposition to the ring topology, we can think of other topologies that allow each thread unit to communicate with anyone else. In this case, when a new speculative thread is created, any thread unit can be allocated for its execution. An idle unit will be chosen if any; otherwise, the unit running the most speculative thread will be chosen and the thread currently running on it will be squashed. This thread management is more complex than that of the sequential ordering scheme because the hardware must be aware of the logical (sequential) order of the speculative threads running in parallel since, now, the logical order no longer corresponds to the physical order of thread units. Besides, the hardware must provide full connectivity among thread units. On the other hand, the processor can speculate at different loop nesting levels, that is, the processor can simultaneously execute different iterations of different nested loops or different nested subroutines instead of the single level of speculation that the sequential thread ordering scheme permits.

Fig. 4 shows the percentage of code that is being executed in parallel with the code of some other threads on the 16-thread unit configuration speculating at different loop/subroutine levels and assuming perfect register value prediction. We can observe that a high percentage of code 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 percent of the code is executed in parallel for this scheme. The percentage of code for the subroutine spawning policy is also quite high (76 percent on average) and this percentage suffers a significant drop for the loop-continuation scheme, which explains its much lower performance.

These high percentages of parallelized code are translated into high speed-ups, as shown in Fig. 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 5.66 and 3.08, respectively, with 16 thread units. Nonetheless, observe that the speed-up achieved by the model that speculates on loop continuations is lower than those of the two other policies (only 2.07 on average). This is due to the fact that this policy is very similar to speculating on loop iterations for any noninnermost loop because, in a loop nest, the continuation of a given loop corresponds to an iteration of the next outer loop in such a way that the granularity of the speculated threads will be larger than for the loop iteration model. The size of the threads depends on the number of iterations of the innermost loop and differences among different threads cause load imbalance, which significantly penalizes performance.

We can observe, comparing Figs. 3 and 5, that the unrestricted thread ordering clearly outperforms the sequential thread ordering. Even for the best performing thread speculation model (loop-iteration) with perfect value prediction for both interthread register and memory dependences, the speed-up achieved by 16 thread units is lower than 1.7, whereas the unrestricted thread ordering scheme has a much higher performance potential (i.e., for perfect value prediction, the best performing thread speculation model is again the loop-iteration and the average speed-up is close to 6 for 16 thread units). In addition, it is also remarkable that the subroutine-continuation scheme, which provides the lowest speed-up in the restricted thread ordering model, is better than the loop-continuation scheme in the unrestricted thread ordering model.

Fig. 6 shows the average speed-up for a number of thread units ranging from 2 to 16, for the loop-iteration and subroutine-continuation spawning schemes. For each scheme, the figure shows the speed-up achieved when considering a perfect predictor for either register values or both register and memory values. Observe that performance scales quite well for all cases, especially when all data values are predicted. Moreover, the performance for the loop-iteration scheme is always higher than for the subroutine-continuation approach.

5 The Impact on Branch Prediction Accuracy

A centralized branch predictor would not be adequate for speculative multithreaded processors since it would require a large number of ports. In addition, branches are not fetched in sequential order and, thus, the history information, especially global history registers, would be significantly degraded by

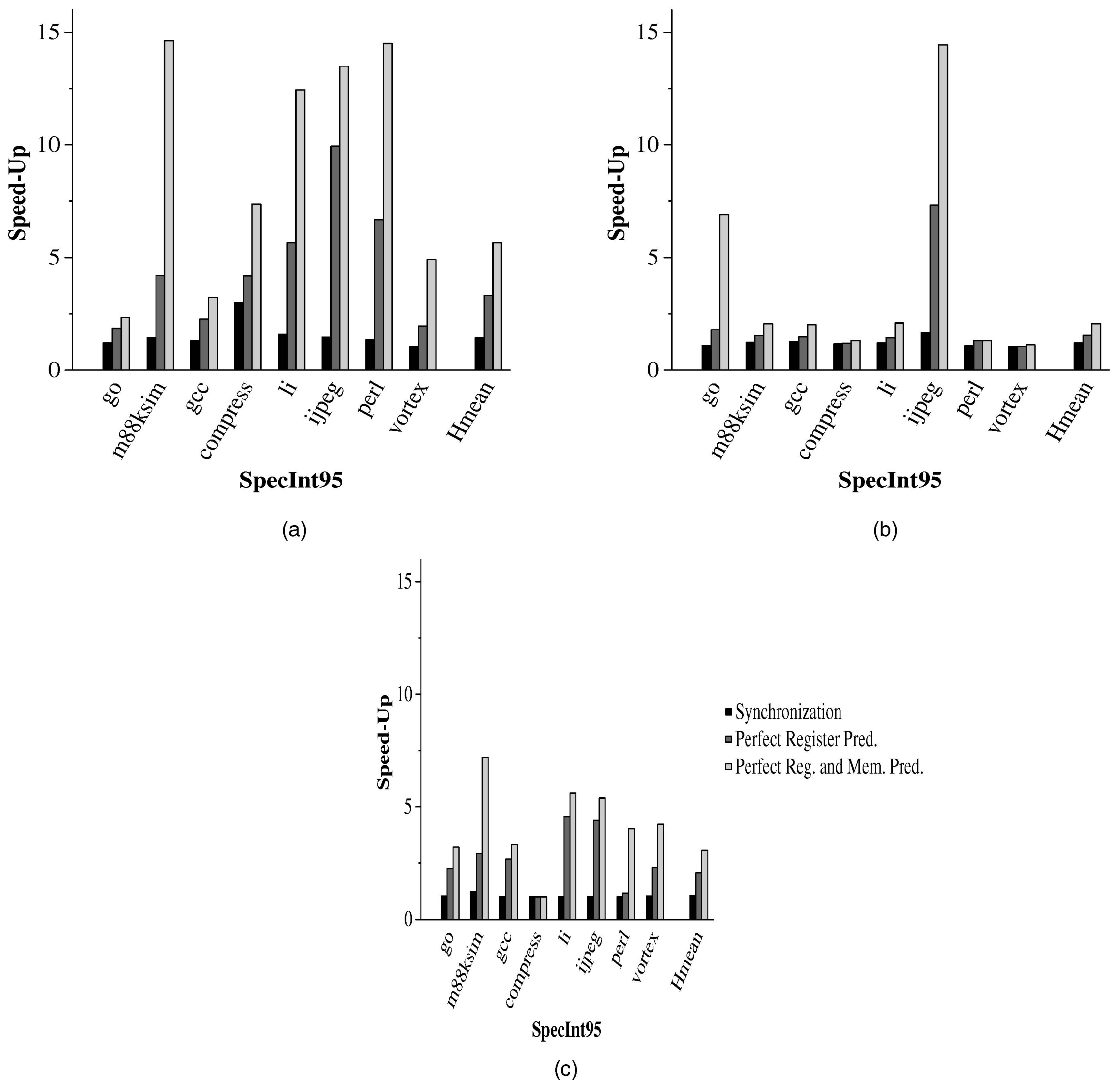

Fig. 5. Speed-ups for the three different spawning policies. (a) Loop-iteration. (b) Loop-continuation. (c) Subroutine-continuation for the unrestricted thread ordering scheme.

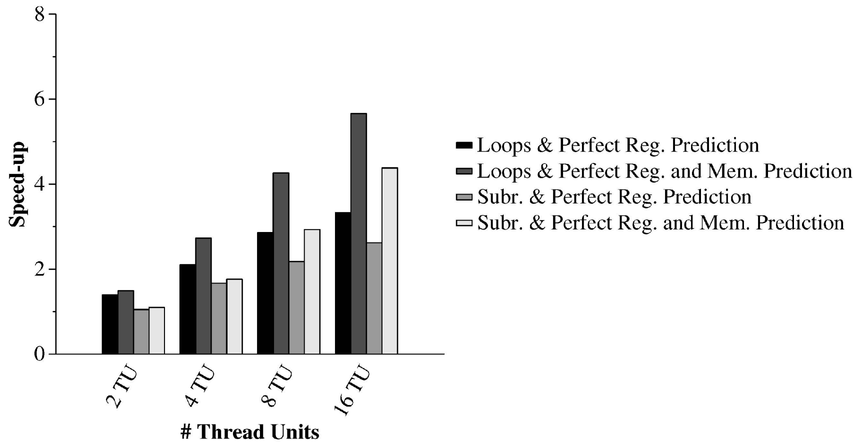

Fig. 6. Speed-up for different number of thread-units.

this. For the experiments in Section 4, a distributed branch prediction scheme was assumed. It consisted of a local branch predictor for each thread unit that worked completely independent of the other predictors once a thread is started. We also assumed that, when a thread is initialized, its branch prediction table is copied from the table of the parent thread. This may imply a too high initialization overhead. Alternatively, branch prediction tables could be

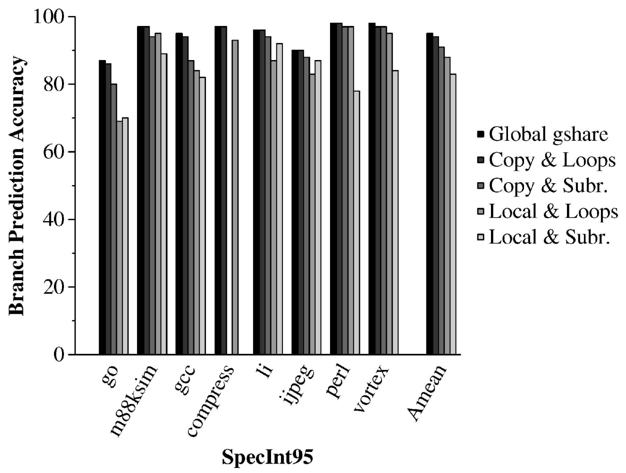

Fig. 7. Branch prediction accuracy.

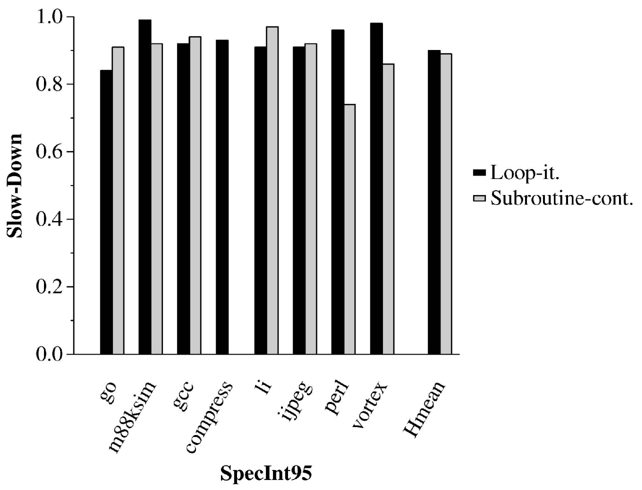

Fig. 8. Slow-down when independent local branch predictors are used.

not initialized at thread creation. Instead, when a new thread is started in a thread unit, it simply inherits the prediction table as it was left by the previous thread executed in that unit.

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

Fig. 7 compares the branch prediction accuracy of branch predictors initialized from the parent at thread creation and that of a noninitialization policy considering the fully connected Clustered Speculative Multithreaded Processor with 16 thread units and for the perfect register prediction configuration. In addition, it also shows the prediction accuracy of a centralized predictor that processes all branches in sequential order as a superscalar microprocessor does, as a baseline for comparison. Observe that the degradation suffered when the copy mechanism is implemented is very low (only 1 percent for a loop-iteration spawning policy and 4 percent for the subroutine-continuation one), but it is significant when predictors are independently managed (higher than 10 percent on average).

Fig. 8 shows the impact of this loss in branch prediction accuracy on the overall performance of the speculative multithreaded processor. This figure depicts the slow-down caused by not initializing the local predictors. On average, the slow-down is close to 10 percent and significant for some programs such as `perl` for the subroutine-continuation spawning policy.

6 The Performance of Value Prediction

From the study in Section 4, we concluded that value prediction of values produced by one thread and consumed by others is crucial for speculative multithreaded processors. Without value prediction, the contribution of thread-level parallelism would be almost negligible for nonnumeric applications.

This section is devoted to analyzing the performance of value prediction in the context of a Clustered Speculative Multithreaded processor fully interconnected. For this analysis, we consider the loop-iteration scheme when spawning speculative threads since, as shown in the previous section, this approach provides the highest speed-up among the three analyzed alternatives. The performance of this type of architecture strongly depends on the ability to predict values flowing through interthread data dependences. Thus, we start by studying the accuracy of different value predictors, including a scheme especially targeted for such architectures, for values produced by one thread and consumed by a different one.

6.1 Value Predictors

Predictors usually exploit the correlation with past values of the instruction operand to be predicted. This approach is followed by the so-called instruction-based predictors. Examples, among others, of instruction-based predictors are the last value (LV) [13], stride (STR) [7], [8], [22], context-based (FCM) [23], and hybrid schemes [2], [33] that include multiple predictors and a selector. In particular, we will consider the stride-context (HYB-S) predictor, which consists of both a stride predictor and a context-based predictor.

The performance of instruction-based predictors can be improved if information about the trace to which the instruction operand belongs is also used [16], [18]. This results in the so-called trace-based value predictors. The increment predictor (INC) [16] is one scheme that follows this strategy. In our particular study, a trace always refers to the code executed by a particular loop iteration. Nevertheless, this predictor can be used together with other criteria to divide the dynamic sequence of instructions into traces.

The INC predictor predicts only data computed inside a trace. It predicts a trace output value as the value of that storage location at the beginning of the trace plus an increment. This increment is computed as the value at the end of the trace minus the value at the beginning of the trace in previous executions of the same trace. The predicted increment is updated when a new increment is seen twice in a row. The main difference between such a predictor and the stride value predictor is that the stride predictor computes a difference between two consecutive values of an operand at the same instruction address.

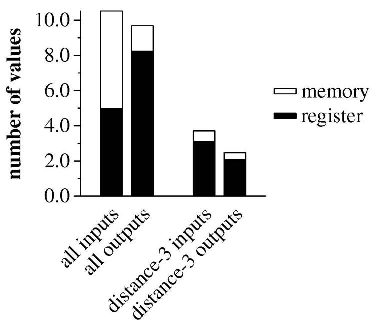

Fig. 9. Average number of inputs/outputs and distance-3 inputs/outputs per trace.

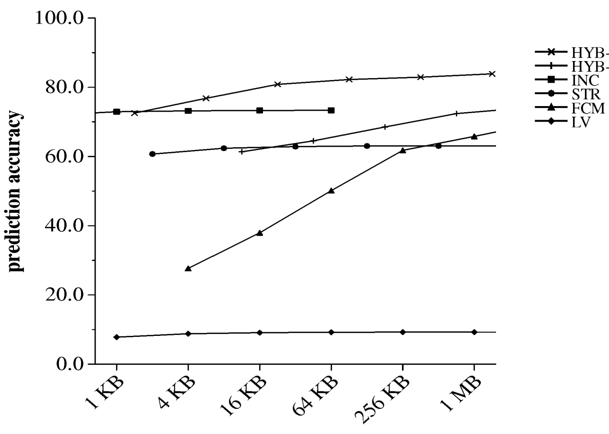

Fig. 10. Predicting distance-3 values of loop traces.

Writes to the same storage location produced between two instructions affect the accuracy of the predictor. Instead, we have observed that, in general, it is better to base the value prediction of a storage location on the difference (the increment) of its value between two given points of the execution that always correspond to the same high-level structure, such as the beginning and the end of a thread.

A hybrid scheme composed of an increment predictor and a context-based predictor (HYB-I) will also be analyzed. For hybrid predictors, the choice between the two predictions is guided by confidence fields located in each individual predictor which are implemented by means of 3-bit up/down saturating counters. The prediction with the highest counter is chosen.

Trace-based value predictors access the history tables through an identifier of the control flow followed by the trace and an operand identifier (e.g., register identifier). Since, in our particular case, a trace is a loop iteration, we have considered that the identifier of the control flow of an iteration consists of the instruction address of the first instruction of the loop along with a bit vector with the result of all conditional branches inside. This is not a unique identifier because target branch addresses of indirect unconditional branches are not considered.

6.2 Prediction Accuracy

This section analyzes the accuracy of the above described value predictors when they are used by a speculative multithreaded processor.

When a thread is spawned to execute a loop iteration, the values flowing through interthread dependences should be predicted. One possibility is to predict all input values needed by the new speculative thread. Another possibility is to predict all output values produced by the previous thread in sequential order. We have computed the average number of input and output thread values (see the two leftmost bars of Fig. 9) and we have observed that there are slightly fewer output than input values.

Moreover, there is no need to predict all input or all output values. Among all the inputs or outputs of a thread, only the prediction accuracy of those that are used speculatively will have an impact on performance. In other words, if a given input or output is already available at the time it is used or an output is never utilized, the performance of the processor will be the same, regardless of the result of its prediction. In other words, performance is related to the prediction accuracy of those input values that are produced in the latest previous threads or those output values that are read by the earliest next threads. As an approximation of this, we have computed the average number of *distance-3 input values* (input values produced by any of the previous three threads) and *distance-3 output values* (output values consumed by any of the following three threads). We can observe in Fig. 9 (the two rightmost bars) that there is a lower number of distance-3 outputs than distance-3 inputs. This led us to use value predictors for thread output values. Fig. 9 also shows that the average number of distance-3 memory inputs and outputs is rather low. Because of that, we have focused on the prediction of register values.

Fig. 10 shows the prediction accuracy for distance-3 output register values and different predictors. The capacity of the history tables is depicted along the X-axis.

As observed for superscalar processors [23], an FCM predictor can achieve a high prediction accuracy, but it requires very large history tables. The STR predictor can achieve better accuracy for small-sized tables. The LV predictor is the least accurate.

A remarkable fact is that the INC predictor significantly outperforms the STR predictor for distance-3 output values. This is explained by the fact that the stride predictor suffers from interferences from other instructions with different addresses that write to the same storage location, whereas these interferences are avoided by a trace-based predictor such as the INC.

The INC predictor obtains a similar performance (73 percent hit rate for the whole range of table capacity) and the HYB-I predictor can achieve an 80 percent hit ratio with relatively small tables (16 KB in total).

As conclusions up to this point, for a speculative multithreaded architecture based on loop iterations, the INC predictor for small sized tables and its hybrid version, the HYB-I predictor, for larger tables outperform the other value predictors. An increment predictor can achieve a quite high hit rate with very small tables (73 percent for a 1 KB table). Moreover, this predictors can be easily added to other multithreaded architectures.

For the four predictors with higher accuracy (HYB-I, HYB-S, INCR, FCM), Fig. 11 shows the percentage of traces whose all distance-3 output values are correctly predicted.

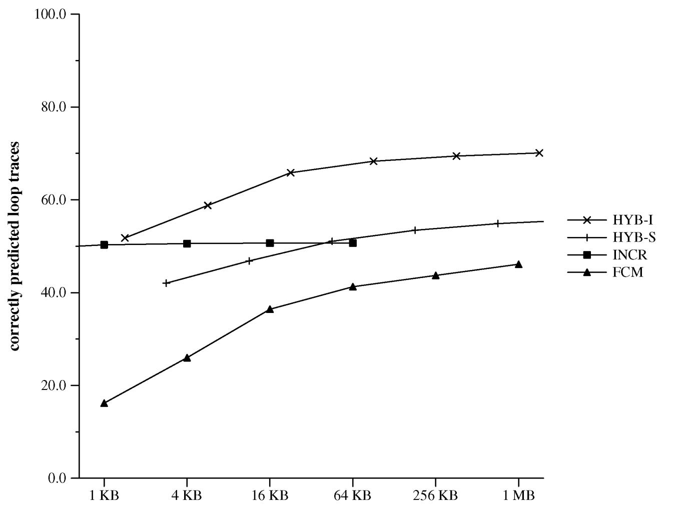

Fig. 11. Percentage of traces that have all their distance-3 output values correctly predicted.

This gives an estimation of the percentage of threads that can be executed as if they were parallel. Note that many traces can be parallelized due to value prediction, even with small predictors (50 percent for 1-KB INCR predictor). With large history tables, this percentage can be as much as 70 percent (with a HYB-I predictor). Note also that the percentage of correctly predicted traces is strongly correlated with the prediction accuracy for individual values.

7 Overall Performance

Fig. 12 shows the speed-up achieved by a Clustered Speculative Multithreaded processor with 16 thread units fully interconnected over a single-threaded execution for different register value predictors. The misprediction penalty considered in this work is the elapsed time until the correct value is available plus an extra cycle to forward the correct value plus one cycle. Note that the average number of cycles waiting for the computation is, in general, significantly larger than the other two factors. Moreover, a selective reissue mechanism is also considered in such a way that only dependent instructions of the mispredicted value have to be reexecuted. Memory values are not predicted and dependent values are forwarded from the producer to the consumer with a delay of three cycles by means of the cache coherence protocol based on the Speculative Versioning Cache [9].

On average, the losses due to a realistic value predictor in comparison with perfect register value prediction are 16 percent for the loop-iteration spawning scheme. Overall, the benefits of speculative thread-level parallelism are still quite high. The loop-iteration model achieves an average speed-up of 2.84.

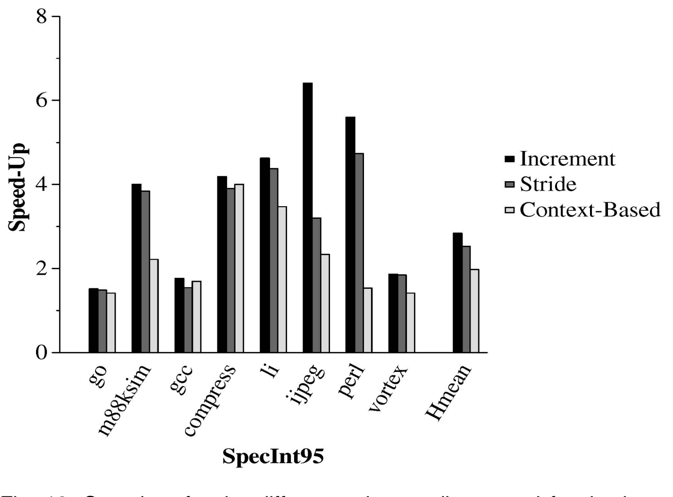

Fig. 12. Speed-up for the different value predictors and for the loop-iteration spawning policy.

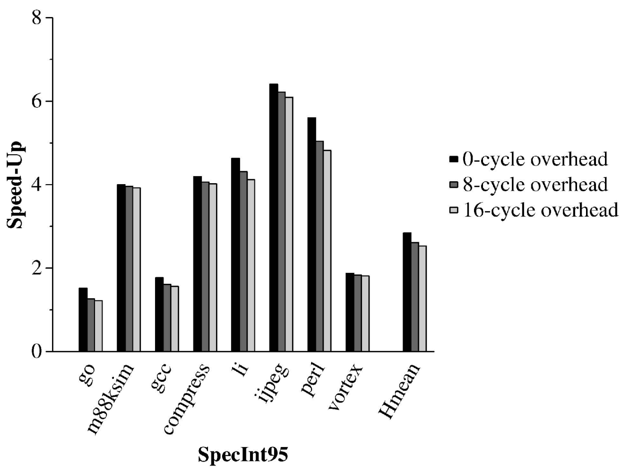

Fig. 13. Speed-up when an overhead penalty is considered.

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

Fig. 13 shows that the impact of the initialization overhead for a penalty of either 8 or 16 cycles is evaluated, which may be reasonable for 32 integer and 32 FP registers. On average, the performance loss is about 8 percent for the loop-iteration spawning scheme with an increment register value predictor, assuming an 8-cycle overhead penalty, and 10 percent when a 16-cycle overhead penalty is considered.

8 Conclusions

In this work, some main design parameters that strongly affect the performance of speculative multithreaded processors have been analyzed, such as the spawning policy and the impact of value prediction. We have first shown that a ring topology has a very limited performance since it can only speculate successfully when speculative threads are created in their sequential order. On the other hand, a full-connectivity architecture with an unrestricted thread ordering scheme provides very high performance benefits. Then, we have shown that the loop-iteration speculation scheme is the one with the highest performance if all values corresponding to interthread dependences could be correctly predicted.

For this spawning policy, the impact of branch predictor and value prediction have been analyzed. First, we note that a centralized branch predictor is not appropriate since branches are not fetched in program order. However, initializing the local branch prediction tables with the contents of the parent table every time a thread is created is very costly and it can dramatically increase the overhead initialization penalty. Then, we have shown that using the previous contents of the branch prediction tables only degrades the performance by 10 percent on average.

On the other hand, the performance of different value predictors has been studied. Experimental results have shown that the increment predictor, which is a special predictor targeted to this kind of microarchitectures, obtains the highest prediction accuracy for small-sized history tables when it is compared with other value predictors. This accuracy is increased for larger history tables by means of a hybrid predictor that combines an increment and a context-based predictors. Average accuracy for SpecInt95 ranges from 73 percent to 84 percent, depending on the capacity of the history table.

Finally, the effect of the thread initialization overhead has been studied. For an 8-cycle initialization overhead, the degradation suffered by the processor for the best value predictor is only 8 percent and it is 10 percent for 16 cycles.

Overall, we conclude that the loop-iteration spawning policy with an increment predictor, 8-cycle spawning thread overhead and a full-connectivity architecture is an effective organization to exploit speculative thread-level parallelism in nonnumerical applications. On average, a 16-thread configuration provides a speed-up of 2.70 for the SpecInt95 suite, which consists of programs which are very hard to parallelize. Finally, note that several spawning policies may be implemented in the same microarchitecture together with some heuristics that identify the most effective one for each particular section of code. We are currently investigating appropriate techniques for this hybrid spawning scheme.

ACKNOWLEDGMENTS

This work has been supported by grants CICYT TIC 511/98, ESPRIT 24942, and AP96-52274600. The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA). The authors would like to thank Bodo Parady (Sun Microsystems, Inc.) for his insightful comments on this paper.

REFERENCES

[1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proc. 31st. Ann. Int'l Symp. Microarchitecture,* 1998.
[2] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," *Proc. 26th Int'l Symp. Computer Architecture,* 1999.
[3] M. Cintra and J. Torrellas, "Eliminating Squashes through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture,* pp. 36-47, 2002.
[4] L. Codrescu and D. Wills, "On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 40-46, 1999.
[5] P.K. Dubey, K. O'Brien, K.M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 109-121, 1995.
[6] M. Franklin and G. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism," *Proc. Int'l Symp. Computer Architecture,* pp. 58-67, 1992.
[7] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction," Technical Report #1080, Technion, 1996.
[8] J. González and A. González, "Memory Address Prediction for Data Speculation," Technical Report UPC-DAC-1996-51, Universitat Politècnica de Catalunya, 1996.
[9] S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi, "Speculative Versioning Cache," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture,* 1998.
[10] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 1998.
[11] G.A. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing," *Proc. Int'l Conf. Parallel Processing,* pp. 239-246, 1996.
[12] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor," *Proc. ACM Int'l Conf. Supercomputing,* pp. 85-92, 1998.
[13] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 138-147, Oct. 1996.
[14] P. Marcuello, A. González, and J. Tubella, "Speculative Multithreaded Processors," *Proc. 12th Int'l Conf. Supercomputing,* pp. 77-84, 1998.
[15] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors," *Proc. 13th Int'l Conf. Supercomputing,* pp. 365-372, 1999.
[16] P. Marcuello, J. Tubella, and A. González, "Value Prediction for Speculative Multithreaded Architectures," *Proc. 32nd Int'l Conf. Microarchitecture,* pp. 230-236, 1999.
[17] P. Marcuello and A. González, "Thread Spawning Schemes for Speculative Multithreaded Architectures," *Proc. Eighth Int'l Conf. High Performance Computing Architecture,* 2002.
[18] T. Nakra, R. Gupta, and M.L. Soffa, "Global Context-Based Value Prediction," *Proc. Fifth Int'l Conf. High Performance Computing Architecture,* pp. 4-12, 1999.
[19] J. Oplinger, D. Heine, and M. Lam, "In Search of Speculative Thread-Level Parallelism," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 303-313, 1999.
[20] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith, "Trace Processors," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 138-148, 1997.
[21] E. Rotenberg, S. Bennet, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. Microarchitecture,* 1996.
[22] Y. Sazeides, S. Vassiliadis, and J.E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," *Proc. 29th Int'l Symp. Microarchitecture,* Dec. 1996.
[23] Y. Sazeides and J.E. Smith, "Implementations of Context-Based Value Predictors," Technical Report #ECE-TR-97-8, Univ. of Wisconsin-Madison, 1997.
[24] G. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. Int'l Symp. Computer Architecture,* pp. 414-425, 1995.
[25] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. Int'l Conf. Programming Panguages Design and Implementation,* 1994.
[26] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "Improving Value Communication for Thread-Level Speculation," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture,* pp. 58-68, 2002.
[27] J. Steffan and T. Mowry, "The Potential of Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture,* pp. 2-13, 1998.

[28] D.M. Tullsen and P.J. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," *Proc. 34th Int'l Symp. Microarchitecture,* pp. 318-327, 2001.

[29] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. Int'l Symp. Computer Architecture,* pp. 392-403, 1995.

[30] J.Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 35-46, 1996.

[31] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 1-12, 1997.

[32] T.N. Vijaykumar, "Compiling for the Multiscalar Architecture," PhD thesis, Univ. of Wisconsin-Madison, 1998.

[33] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *Proc. 30th Int'l Symp. Microarchitecture,* 1997.

[34] F. Warg and P. Stenstrom, "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 221-230, 2001.

Pedro Marcuello received the degree in computer science in 1995 and the PhD degree in computer science in 2003, both from the Universitat Politècnica de Catalunya at Barcelona, Spain. He has been a member of the Computer Architecture Department at the Universitat Politècnica de Catalunya since 1997 and a full-time teaching assistant since 2000. His current research topics include speculative multithreading and value prediction.

Antonio González received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. He has been a faculty member of the Computer Architecture Department at UPC since 1986 and he is currently a full professor. He leads the Intel Barcelona Research Center at UPC, whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. He has published more than 150 papers in technical journals and symposia and has served on more than 40 program committees for international symposia in the field of computer architecture. Dr. González is a member of the IEEE Computer Society.

Jordi Tubella received the degree in computer science in 1986 and the PhD degree in computer science in 1996, both from the Universitat Politècnica de Catalunya, Barcelona, Spain. He has been a member of the Computer Architecture Department at the Universitat Politècnica de Catalunya since 1988 and has been an associate professor since 1998. His research interests focus on processor microarchitecture and parallel processing, with special interest on multithreading, value prediction, and instruction reuse.