# Clustered Speculative Multithreaded Processors

Pedro Marcuello and Antonio González

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

E-mail: {pmarcue,antonio}@ac.upc.es

## Abstract

*In this paper we present a processor microarchitecture that can simultaneously execute multiple threads and has a clustered design for scalability purposes. A main feature of the proposed microarchitecture is its capability to spawn speculative threads from a single-thread application at run-time. These speculative threads use otherwise idle resources of the machine.*

*Spawning a speculative thread involves predicting its control flow as well as its dependences with other threads and the values that flow through them. In this way, threads that are not independent can be executed in parallel. Control-flow, data value and data dependence predictors particularly designed for this type of microarchitecture are presented.*

*Results show the potential of the microarchitecture to exploit speculative parallelism in programs that are hard to parallelize at compile-time, such as the SpecInt95. For a 4-thread unit configuration, some programs such as ijpeg and li can exploit an average degree of parallelism of more than 2 threads per cycle. The average degree of parallelism for the whole SpecInt95 suite is 1.6 threads per cycle. This speculative parallelism results in significant speedups for all the SpecInt95 programs when compared with a single-thread execution.*

**Keywords:** Data value speculation, data dependence speculation, control-flow speculation, clustered processors, dynamically scheduled processors, simultaneous multithreaded processors.

## 1. Introduction

Technology evolution projections anticipate that in about 15 years around 1 billion transistors will be available in a single chip microprocessor. Computer architects have to cope with the challenge of making an effective use of such huge amount of logic. Scaling up current superscalar organizations may provide some benefits but will soon reach a point of diminishing returns, especially for non-numeric applications. Data dependences[1], instruction window size and wire delays are probably the most important hurdles that limit the performance of superscalar processors.

[1] In this paper, data dependences refer to RAW dependences

Data dependences cause just by themselves a severe limitation to performance. For instance, the SPECint95 benchmarks compiled with the DEC Alpha compiler with full optimization have a maximum instruction-level parallelism of 37 instructions per cycle (IPC), assuming an ideal machine with infinite resources and perfect branch prediction. For 2 out of the 8 benchmarks, the IPC is even lower than 20 [9]. If the effect of a limited instruction window is then considered, the maximum IPC significantly drops, approaching values that are not far from the peak IPC that may be exploited by current superscalar processors. On the other hand, as future designs employ smaller feature sizes, wire delays will not scale down to the same pace and will become the most critical component of the execution time [22].

Penalties due to data dependences may be relieved by data value speculation. This technique is based on predicting the input/output operands of instructions, which avoids the ordering imposed by data dependences otherwise. Recent studies have shown that the performance impact of data value speculation for superscalar processors is moderate [8], and its potential improvement approaches a linear function of the prediction accuracy. This suggests that one cannot expect much benefit by just improving the value predictor. However, its potential impact on a multithreaded architecture is much higher [9].

The effective instruction window size is mainly limited by the branch predictor accuracy. The amount of control-flow speculative instructions from the correct path that can be in the instruction window depends on the number of consecutive branches that can be correctly predicted, due to the sequential nature of the fetching mechanism of superscalar processors. As an approximation, if we assume that the probability of predicting correctly any branch is $p$, and that it does not depend on the success/failure in previous branches, we have that the average number of consecutive branches that are correctly predicted is $p/(1-p)$, and if $B$ is the average number of instructions between branches, the number of correctly speculated instructions in the instruction window will be on average $(p/(1-p))\cdot B$. For instance, if $p$=0.95 and $B$=5, the average number of correctly (control-flow) speculated instructions will be on average 9.5, which is a rather limited window size in spite of a such an accurate branch predictor.

Finally, the problem of wire delays may be handled by exploiting *communication locality.* The idea is to partition a full design into several blocks, including in the same block those components that frequently communicate and placing in different blocks those components that rarely exchange data. Sometimes this organization has been referred to as *clustered* or *decentralized microarchitecture.* Examples of such microarchitectures are the Alpha 21264[10], and those proposed in [5][6][13][22][23][27][33].

The processor microarchitecture that we propose in this paper attacks these three problems in the following way. First, it is a clustered microarchitecture in order to exploit communication locality. Second, the instruction window is not built by a sequential process. Instead, it consists of a collection of non-adjacent smaller windows. Each small window is composed of a subsequence of the dynamic instruction stream, and it is build by sequentially speculating on branches. Each window is initially started by speculating on highly predictable branches. In particular, we have considered those branches that close loops. However, other alternatives may be considered, like subroutine calls. Each instruction window is managed as a thread of control (thread for short), and it is not necessarily independent of other threads. Third, dependences among threads are predicted, as well as the values that flow through them. This allows each thread to execute at its own pace, without synchronizing with other threads, provided that dependent values are correctly predicted. As shown in [9], the potential of data value speculation for a multithreaded processor is higher than for a single-thread processor. Furthermore, we show in this paper that the approach taken to dynamically partition a program into threads favors the accuracy of value predictors. Inter-thread dependent values are more predictable than the average. Finally, a side-effect of the proposed thread partitioning is that concurrently active threads usually share the same code (different iterations of the same loop), which can be used to significantly reduce the fetch bandwidth requirements.

The rest of this paper is organized as follows. Section 2 describes the microarchitecture of the clustered speculative multithreaded processors, with especial emphasis on the control-flow, data value and data dependence speculation mechanisms. Performance figures are discussed in section 3 and the related work is reviewed in section 4. Finally, the main conclusions are summarized in section 5.

## 2. Clustered Speculative Multithreaded Microarchitecture

The clustered speculative multithreaded processor has roots on the Multiscalar [27], with some notable differences: a) thread partitioning and data dependence enforcement are performed by run-time mechanisms, without any compiler support; and b) data speculation techniques are used to avoid the serialization imposed by data dependences.

The clustered speculative multithreaded microarchitecture was initially proposed in [20]. That preliminary proposal showed a high performance for FP codes but the performance for integer codes was rather low. The main reason for this poor performance was that parallel threads, each one corresponding to a different innermost iteration[1] of the same loop, were constrained to follow the same control flow. Figure 1 shows the average number of consecutive innermost iterations that follow the same control flow for the SpecInt95 programs, which can be considered as an upper bound on the degree of thread-level parallelism that could be exploited by such microarchitecture. In fact the actual upper-bound is still lower, since for each change in the control flow, the processor requires two iterations to correctly predict the next threads. If we substract 2 to the numbers of Figure 1, we can conclude that the potential parallelism for many programs is extremely low. This figure points out that constraining all the threads to follow the same control flow causes severe limitations on the performance of integer codes.

All the statistics presented in this section, except for Figure 5, have been obtained by processing the trace corresponding to the whole execution of the SpecInt95 benchmark suite with the train

[1] We define an innermost iteration as an iteration that does not contain any loop. Note that a given loop may consist of some innermost iterations and some others that are not innermost, if the enclosed loop is conditionally executed.

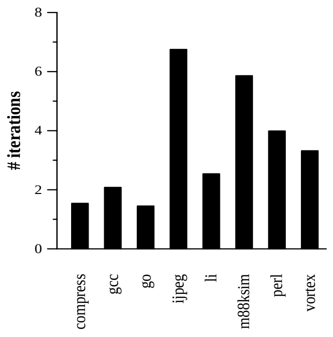


**Figure 1. Average number of consecutive innermost iterations with the same control flow.**

inputs listed in section 3. More details about the experimental framework can also be found in that section.

This paper extends the clustered speculative multithreaded microarchitecture to support parallel threads that follow different control flows. This implies significant changes in the instruction fetch and control speculation engines, as well as in the value prediction mechanism. This extension will significantly boost the performance of integer codes. We will show that this microarchitecture can exploit an important degree of thread level parallelism for programs that are known to be hard to parallelize at compile-time (1.6 threads per cycle on average for 4-thread units). We describe below the main features of the clustered speculative multithreaded microarchitecture.

The clustered speculative multithreaded microarchitecture is shown in Figure 2. It consists of several thread units, each of them being quite similar to a superscalar core, except for a set of live-in registers that are used to pass register values among neighbor thread units. Every thread unit has its own register file (local registers), register map table, instruction queue, functional units, local memory and reorder buffer. Thus, it is a multithreaded architecture in which most of the hardware resources are local to each thread unit. This approach is more scalable than sharing the resources.

A main feature of this processor is its capability to dynamically extract multiple threads from a sequential program and execute them in parallel without any support from the compiler and without requiring any extension to the ISA. Note that this processor can also exploit thread/task-level parallelism provided by the user/compiler/run-time system. However, when there are idle thread units due to the inability of software

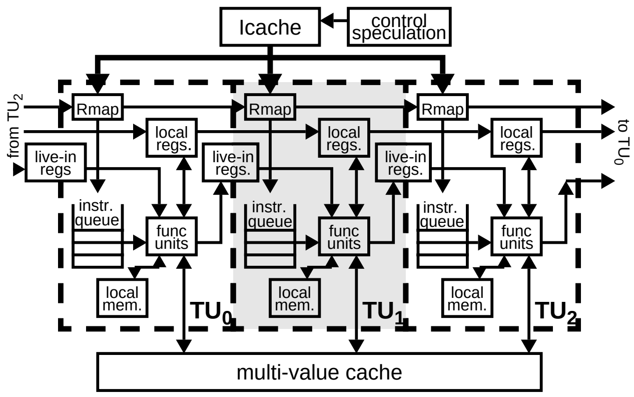


**Figure 2. A clustered speculative multithreaded processor with three thread units.**

techniques to extract parallelism (e.g., when executing a single non-parallelizable application), the processor can generate multiple speculative threads that are executed in the idle thread units and can speedup the execution of a single software thread. In this paper we focus on this latter capability, since executing multiple software threads in a multithreaded architecture has been extensively researched in previous works (see [31] for instance).

The speculative threads correspond to different consecutive innermost iterations that do not have to be independent. When the processor reaches the beginning of an iteration, idle thread units are allocated to speculatively execute following iterations. The register file of a speculative thread is initialized with the predicted output values of the previous thread and thus, the value predictor is one of the critical parts of the microarchitecture. Predicting the control flow correctly is useless if the input values (i.e. the values of registers that are live at the beginning of a thread) are mispredicted.

Memory values are handled by means of a special first-level cache organization, the *multi-value* cache, which manages multiple versions of each memory location. Each version corresponds to the state of a memory location in a given point in time of the execution, that is, the state of that memory location as it is viewed by each different thread.

Data dependences among instructions in different threads (inter-thread dependences for short) are also predicted. Data dependence prediction may significantly reduce the number of data dependence misspeculations for large instruction windows (see [21] among others). Dependences through registers are predicted by predicting the number of writes to each logical register that each thread will perform. Once this number of writes are performed, the register contents are forwarded to the following thread, either to check if the predicted value of that register was correct or to pass the value when it was not predicted. Memory dependences are predicted in a similar way, that is, by predicting the memory locations that each thread will write. An address predictor is used for this purpose. The multi-value cache is initialized with this information and memory values are passed among threads by enforcing the dependences implied by the predicted writes.

Threads are allocated to thread units in sequential order following a ring topology. The first thread is called the *non-speculative* thread whereas the other ones are *speculative* threads. When the non-speculative thread reaches the point where the first speculative thread started, it terminates and the first speculative thread becomes non-speculative. Precise exceptions are supported by means of the local reorder buffers, which allow the processor to retire the instructions of each thread in order. Values produced by speculative threads are held either in their local register files/local memories or in the multi-value cache and are not allowed to propagate to lower levels of the memory hierarchy.

Values and dependences are predicted by means of a history table (the *loop iteration table*) that holds information regarding some of the last executed loops. Details about the mechanisms to predict values and dependences, as well as the management of threads are provided in [20][30].

Another important feature of the clustered speculative multithreaded processor is its low instruction fetch bandwidth requirements. In the original proposal, since all simultaneously active threads followed the same control flow, a simple fetch engine could feed the processor by fetching each instruction just once and replicating it for all active threads. This results in an average IPC for the SpecFP95 of 5.2 with a fetch unit with a peak bandwidth of 4 instructions per cycle. The new microarchitecture proposed in this paper can execute multiple threads that follow

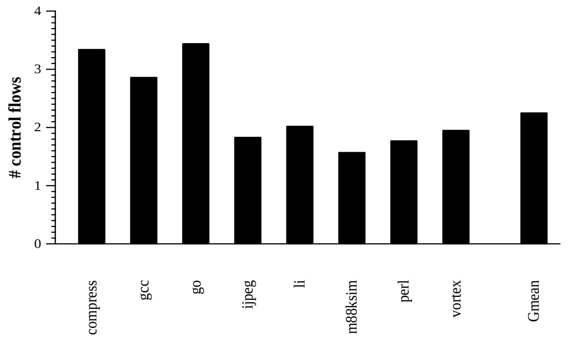


**Figure 3. Average number of different control flows for every 8 consecutive innermost iterations.**

different control flows, which will significantly speedup integer codes. However, this will also reduce the benefit of saving instruction fetches by replicating instructions for those threads that follow the same control flow. Nevertheless, the benefits will still be important since we have measured that the number of different control flows is very low, in spite of the fact that the control flow changes frequently. For instance, Figure 3 shows the average number of different control flows for every 8 consecutive innermost iterations. On average, in every 8 consecutive innermost iterations there are only 2.3 different control flows. This implies that if these iterations were executed in parallel, the fetch bandwidth requirements could be reduced in a factor of 3.5.

## 2.1 Support for Multiple Control Flows

As discussed above, threads that follow different control flows should be allowed to be executed simultaneously in order to obtain significant speedups for integer codes. For this purpose, each entry of the loop iteration table is associated to a particular control flow followed by some iterations of a particular loop. We will refer to the sequence of instructions of a particular control flow of a given loop as a *loop trace*. For instance, if a loop has just a conditional branch (in addition to the loop closing branch), the iterations of this loop may follow up to two different control flows and therefore, up to two different loop traces may be found during the execution of that loop

A loop trace is a particular type of traces [24] that comprises a whole loop iteration (including subroutine activations) and can also include multiple indirect branches. It is identified by the PC of its first instruction and a sequence that defines a particular control flow. This sequence consists of all branches/jumps, indicating for each conditional branch whether it is taken or not, and the target addresses of indirect jumps.

The loop iteration table provides support for data and data dependence speculation, that is, it is used to predict dependences and the values that flow through such dependences. Dependence and value prediction is not based on a stride-predictor of live-in[1] values as proposed in [20] since different loop traces may have different live-ins, and thus, it is unlikely that live-ins follow a stride pattern. We present new value and dependence predictors in the next section. Very few entries are enough to achieve a significant hit ratio in the loop iteration table, as shown in Figure 4 for an 8-entry table.

The ability of simultaneously execute threads that follow different control flows has an impact on the design of the fetch

[1] We refer to a register/memory location that is live at the beginning of a loop trace as a *live-in* or an *input.*

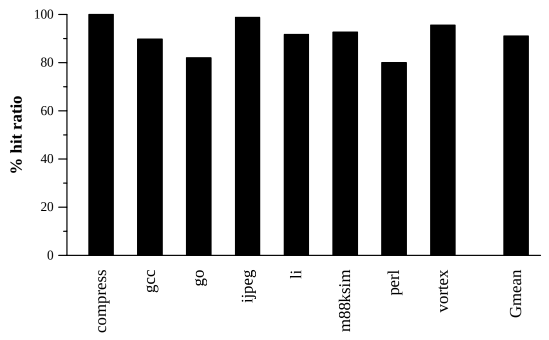


**Figure 4. Hit ratio of an 8-entry loop iteration table**

engine. The fetch engine manages multiple PCs, one for each different control flow. Each cycle, a single fetch from a PC is performed and the obtained instructions are forwarded to all threads that follow this control flow. The particular PC used in each cycle is selected among those in which the instruction queues of the associated threads are not full, following a round-robin policy.

To confirm the potential performance of a clustered speculative multithreaded processor with support for different control flows to speedup non-numerical codes, we initially evaluated a 4-context processor with a perfect loop trace predictor. That is, the processor was assumed to be able to always predict correctly the control flow that the next iteration will follow, its input values and its dependences. The remaining processor parts were realistically modelled. Details about the experimental framework can be found in section 3. Figure 5 shows the degree of thread level parallelism that would be exploited by such a processor as well as the speedup over a single-threaded execution. It can be seen that all the benchmarks show significant speedups. On average, the processor may have 2.1 active threads which results in a 70% increase in performance over a single thread.

## 2.2 Loop Trace Prediction

The process of spawning speculative threads comprises two important elements: a) deciding *when* speculative threads are spawned; and b) predicting the loop trace associated to those threads.

Speculative threads are tried to be spawned when the non-speculative is in an innermost loop and there are idle thread units. The processor knows when the non-speculative thread enters an innermost loop by searching the loop iteration table for an entry with a PC that matches the current PC. If such an entry is found,

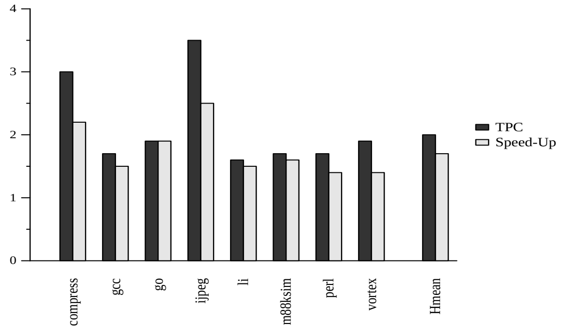


**Figure 5. Average number of threads per cycle (TPC) of a 4-context clustered speculative multithreaded processor and speedup over a single-threaded execution for a perfect loop trace predictor**

then the speculation engine spawns as many threads as idle thread units provided that it can predict their associated loop trace. Every time that the non-speculative thread finishes (and the first speculative thread becomes the non-speculative one), its thread unit becomes idle and it is allocated to a new speculative thread.

To spawn a thread, the speculation engine must predict its associated loop trace, which involves two different tasks: a) predicting its control flow; and b) predicting its input values and inter-thread dependences. These tasks may be carried out by the same predictor or by different ones, as described below.

### *2.2.1 Value and Dependence Prediction*

Predicting the input values of a loop trace when it is spawned is a key component of the proposed microarchitecture since it allows a thread to be executed without waiting for previous threads to produce such values. Note that previously proposed value predictors may not be adequate since they try to exploit features of individual instructions (e.g., constant, stride or repeated sequences). What we require here is a predictor whose predictions are associated to traces instead of individual instructions. It must predict the input values of a loop trace before having fetched all the instructions of previous traces. Note that the inputs of a trace can also be predicted by predicting the outputs of previous traces.

We could try to adapt previously proposed predictors to predict the input/output values of a trace by regarding a loop trace as a single unit. However, they will unlikely work well. For instance, it is unlikely that the input or output registers of a loop trace follow a stride sequence, since several other traces that modify these registers may have been executed in between two consecutive executions of the same loop trace.

We propose a predictor that is based on the observation that the value of each output register of a loop trace that is live for a following loop trace usually is equal to its value at the beginning of the trace plus a constant increment that is the same as the last time the same loop trace was executed. The proposed predictor consists of a table that is indexed by the loop trace identifier and it stores for each output register the difference between its value at the beginning and its value at the end of the trace for the last time the loop trace was executed. Next time the same trace is encountered, the output registers are predicted to be updated in the same way as they were last time. We will refer to this predictor as an *increment* predictor.

Every time a new thread is started, its output values are immediately predicted and they will speculatively be used as input values by the next thread when it is spawned. When a thread finishes, its output predictions are verified and mispredictions are handled by selective re-execution.

We have evaluated the potential of this predictor by measuring the percentage of correctly predicted inputs that would be obtained by predicting the outputs of a loop trace as their value at the beginning of the trace plus the same increment as the last execution of the same loop trace. Results, which are shown in Figure 6, show that this approach may predict 92% of all input registers on average. The hit ratio is quite high (higher than 88%) for all the programs. The actual hit ratio when the mechanism is included in a timing simulator is slightly lower since it is negatively affected by loop trace mispredictions and by predicting the outputs using predicted inputs instead of actual values. Note however that these figures are much higher than those reported when predicting all instructions, even if infinite tables are considered [26]. This suggests that a proper selection of the values to be predicted may have an important impact on the performance of data value speculation techniques.

Regarding memory dependences, as previously anticipated, they are predicted by predicting all memory locations that will be

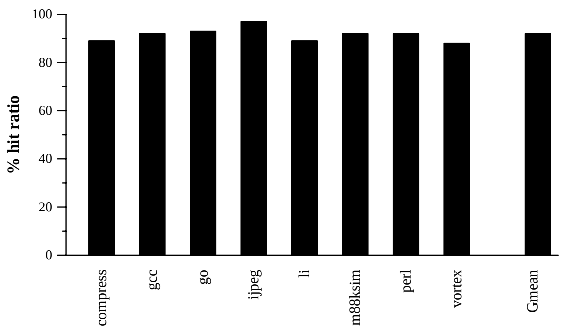


**Figure 6. Potential accuracy of the value predictor.**

written by a given loop trace. The effective addresses of store instructions are predicted based on the prediction made for registers. For each store in a loop trace, the loop iteration table holds the logical register used to compute its effective address and the offset relative to the value the register had at the beginning of the loop trace in its last execution. When a new thread is spawned, the effective address of each store instruction is predicted as the value predicted for its associated logical register plus the offset obtained from loop iteration table.

### *2.2.2 A Combined Control Flow and Value/ Dependence Predictor*

Since a loop trace is a particular type of trace, the control flow of speculative loop traces could be predicted by means of predictors previously proposed for traces, like the *path-based next trace* predictor [12]. We have considered the hybrid scheme proposed in that paper modified to include a different history register per each loop, instead of a single one, since it has a higher performance.

When a new speculative thread is spawned, the trace predictor is first used to predict its control flow, which identifies a particular loop trace. Then, the value predictor is used to predict its outputs, so that the next thread, if any, can immediately start since all its input values are available. The value predictor is implemented in the loop iteration table by including in each entry the last observed increment for each output register.

### *2.2.3 Decoupling Control Flow Prediction from Value/Dependence Prediction*

In order to execute a thread, predicting the whole control flow that it will follow is not necessary. This may be convenient if the processor has a *trace cache* that stores the instructions of loop traces since it would allow the processor to fetch the whole trace immediately. However, if we consider a conventional instruction cache, the only requirement to start a thread is to know its initial PC. Each thread can execute at its own pace by fetching instructions starting at that PC and predicting branches just when they are fetched, through a local branch predictor.

Each local predictor is updated with the branches that it processes. When there is a single active thread, the local predictors of idle thread units are also updated for each branch, so that, when threads are spawned, they reflect the branches processed up to this point.

Using local branch predictors is simpler than having just a shared predictor because the number of ports of the tables is significantly reduced, and it is a more scalable design. Moreover, a shared predictor would be negatively affected by the fact that branches are predicted and processed out of order, due to the parallel execution of multiple threads.

A loop trace predictor is still required by the value predictor, since output values are predicted by adding an increment to each output register and this increment is associated to a particular loop trace. Thus, the main difference between this approach and the one presented in section 2.2.2 relies on the way the control flow of loop traces is predicted. In the combined predictor of section 2.2.2, it is predicted by a path-based next trace predictor, whereas in the decoupled approach it is predicted by local conventional branch predictors.

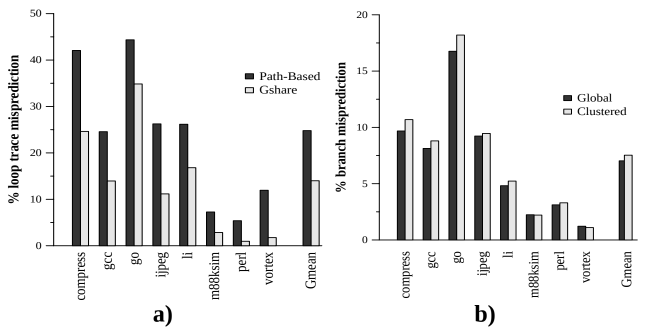


**Figure 7. Control flow prediction accuracy of different approaches.**

To choose between the two schemes, we have compared the control-flow prediction accuracy of both schemes. We have assumed the hybrid scheme of the path-based next trace predictor proposed in [12] with infinite tables and history depth of 3. On the other hand, we have evaluated the trace hit ratio of 4 local gshare predictors [18], each one belonging to a different thread unit. Each of these predictors has a 14-bit history register and a $2^{14}$-entry pattern history table. We consider that the gshare correctly predicts a loop trace when it hits in the prediction of all the branches in the trace.

Figure 7.a compares the loop trace misprediction rate of the path-based scheme and the local gshare predictors. We can observe that the local gshare approach significantly outperforms the path-based predictor, reducing the average loop trace misprediction rate from 24.8% to 14.0%.

Finally, we have evaluated the prediction degradation caused by having local predictors instead of a global one. Figure 7.b compares the branch misprediction rate (for individual branches) of a global gshare with $2^{16}$ entries against the misprediction rate of the 4 local gshare predictors with $2^{14}$ entries. For the global predictor, we have assumed that branches are processed in their sequential order (this is an optimistic assumption that would not be realizable). The results show that the degradation due to using local predictors is negligible. The average misprediction rate just grows from 7% to 7.5%.

## 3. Performance evaluation

In this section, we present performance figures of the clustered speculative multithreaded processor. The objective is to demonstrate its potential to speedup applications that are hard to parallelize at compile-time, such as the SpecInt95, by exploiting speculative thread-level parallelism.

## 3.1 Experimental framework

The clustered speculative multithreaded processor has been evaluated through trace-driven simulation of the SpecInt95 benchmark suite. The programs have been compiled with the DEC compiler for an AlphaStation 600 5/266 with full optimization (-O4) and instrumented by means of the Atom tool. For statistics that do not require a cycle-level simulator of the whole processor we run the programs with the train inputs until completion. To obtain accurate timing results, such as execution time, a cycle-by-cycle simulation is performed. Because of the detail at which simulation is carried out the simulator is slow, so we have simulated 50 million of instructions for each benchmark after skipping the initial part that corresponds to initialization of data structures. In this type of experiments we have used the *ref* inputs, since they may reflect a more realistic workload, in particular for some parameters such as number of iterations per loop. Table 1 lists the input set for each experiment.

| Benchmark | Train input | Ref input |
|---|---|---|
| go | 9 9 | 50 21 |
| m88ksim | /train/input/ | /ref/input/ |
| gcc | genrecog.i | 1amptjp.i |
| compress | 40000 e 2231 | 14000000 e 2231 |
| li | train.lsp | /ref/input/*.lsp |
| ijpeg | vigo.ppm | penguin.ppm |
| perl | train/input/ | /ref/input |
| vortex | train/input | /ref/input |

**Table 1: Benchmark summary**

We have evaluated a clustered speculative multithreaded processor with 2 and 4 thread units. The fetch bandwidth of the architecture is up to 4 instructions per cycle or up to the first taken branch, whichever is shorter, and the fetch policy for threads with different control flow is round-robin. The MultiValue cache has 128 entries (4KB capacity) and a latency of 1 cycle. A 32KB non-blocking, 2-way set-associative L1 cache with an 8-byte block size and up to 4 outstanding misses is considered. The L1 latencies are 2 cycles for a hit and 6 cycles for a miss. An ideal L2 cache memory is considered. An 8-entry Loop Iteration Table has been assumed. The Path-Based predictor is implemented by means of a 1K-entry secondary table and an 8K-entry correlating table. Each thread unit has the following parameters:

- Issue bandwidth: 4 instructions per cycle.
- Physical Registers: 64.
- Local Memory: 64 entries (512 bytes).
- Functional Units (latency in brackets): 2 simple integer (1), 1 integer multiplication (2), 2 simple FP (3), 1 FP multiplication (6) and 1 FP division (17).
- Reorder buffer: 64 entries.
- Branch predictor: $2^{14}$-entry gshare.

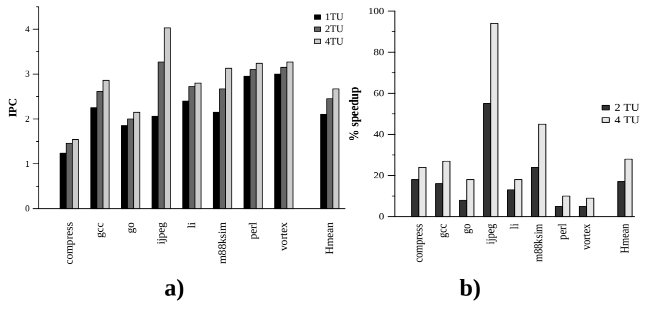


**Figure 8. a) IPC for 1, 2 and 4 thread units; b) speedup over a single-thread execution.**

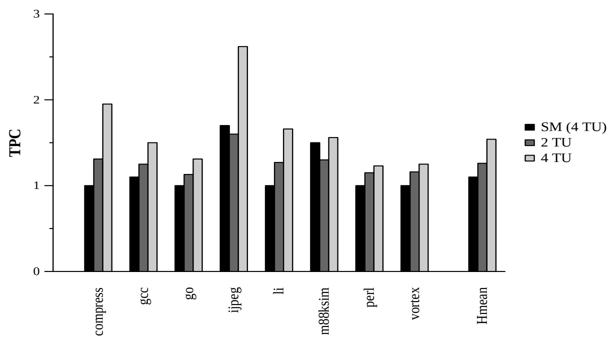


**Figure 9. Average number of active threads per cycle (TPC) for 2 and 4 thread units. For comparison purposes, it is also shown the TPC achieved by the SM architecture [20] with 4 thread units.**

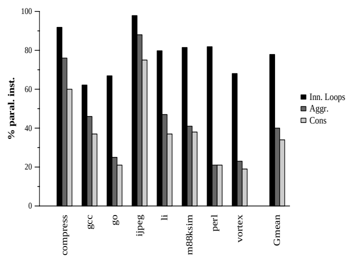
We have considered two different policies to spawn speculative threads. In both cases, threads are spawned only if their associated information to predict values and dependences is found in the loop iteration table. For the *conservative* policy, the confidence bits of the path-based loop trace predictor, which is used only for the value and dependence predictor, are considered. If the prediction is not confident, the thread is not spawned. For the *aggressive* policy, the confidence bits of the loop trace predictor are irrelevant. The aggressive policy may probably exploit more thread-level parallelism but incurs in a much higher number of mispredictions.

## 3.2 Performance figures

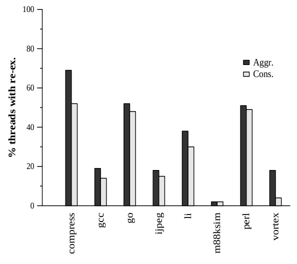
We first evaluate the clustered speculative multithreaded processor with an aggressive spawning policy. Figure 8.a shows the instructions committed per cycle (IPC) for each program when it is executed on a single, two and 4-thread units whereas Figure 8.b shows the speedup over a single-thread execution. We can see that a speculative multithreaded execution provides significant speedups over a single-thread execution for all the programs. For 4-thread units, the highest speedups are exhibited by *ijpeg* and *m88ksim*, which achieve 94% and 45% respectively, and the lowest speedups are 9% and 10%, which correspond to *perl* and *vortex* respectively. On average, the speedup is 17% and 28% for 2 and 4 thread units respectively. It is remarkable the high IPC of some programs, which approximate the bound due to the fetch bandwidth. In particular, *ijpeg* achieves an IPC of 4.03 with 4-thread units, which is slightly higher than the fetch bandwidth, which confirms the benefits of the fetch sharing approach.

Figure 9 shows the degree of thread-level parallelism exploited by the clustered speculative multithreaded architecture. It can be observed that speculative multithreading is an important source of parallelism for all SpecInt95 programs, which are known to be hard to parallelize at compile-time. For 4-thread units, the average number of threads per cycles (TPC) is higher than 1.2 for all programs. On average, the TPC is 1.3 and 1.6 for 2 and 4-thread units respectively. *Ijpeg* and *compress* show a remarkable degree of speculative parallelism, with a TPC of 2.6 and 2.0 respectively. Note that the speedups are correlated with the TPC figures, being the latter always lower than the former, mainly due to misspeculations. Figure 9 also shows the TPC for the initial microarchitecture proposed in [20], which was able to exploit speculative parallelism only when threads followed the same control flow. It can be seen that the benefit of supporting multiple control-flows is very significant.

Finally, we compare the performance of the *conservative* and



a)



b)

**Figure 10. a) Percentage of dynamic instructions that are parallelized; and b) percentage of threads with re-executions due to value/dependence mispredictions.**

*aggressive* spawning policies. Figure 10.a shows the percentage of dynamic instructions that are parallelized. The first bar shows the percentage of instructions in innermost iterations. It can be seen that the part of the code that is candidate to be parallelized is quite high for all the programs. The other two bars show the percentage of dynamic instructions that are actually parallelized by the conservative and aggressive policies. As one may expect, the aggressive policy provides more speculative parallelism (about 40% of the code is parallelized). However, it incurs in a higher number of re-executions due to value/dependence misspeculation, as shown in Figure 10.b. The combination of these two opposite effects is that the aggressive approach achieves a slightly higher performance in terms of IPC for 4 programs (*compress, gcc, go* and *ijpeg*), whereas for other 3 programs (*m88ksim, perl* and *vortex*) the performance is very similar, and finally, for *li* the best performance is obtained by the conservative policy (see figure 11).

## 4. Related work

This work is inspired in previous proposals for dynamically scheduled processors with support for multiple speculative threads, such as the Expandable Split Window paradigm [6]; Multiscalar processors [27]; the SPSM architecture [4]; the Superthreaded architecture [29]; the Multithreaded Decoupled architecture [3]; Trace processors [23] [33]; the Dependence Speculative Multithreaded Architecture [19]; and Dynamic Multithreaded processors [1]. The Clustered Speculative Multithreaded architecture differs from previous proposals in the way that the speculation mechanisms are implemented. The control-flow, data value and data dependence speculation mechanisms are new contributions of this microarchitecture. Other important differences are:

- Multiscalar, SPSM, Superthreaded, Multithreaded Decoupled and Dependence Speculative Multithreaded architectures do not use data value speculation.
- Trace processors speculate on arbitrary traces. Loop traces have more predictable inputs/outputs and control flows since they are associated to a part of the program that corresponds to a high-level construct. Moreover, trace processors have a global register file, which limits its scalability.
- The Dynamic Multithreaded Processor does not predict dependences and the value prediction scheme is very simple: the register file of the parent thread is copied into the child register file. Moreover, it spawns a speculative thread to execute the continuation of a loop whereas the non-speculative one proceeds with the whole loop. It also spawns a speculative thread at subroutine invocations.

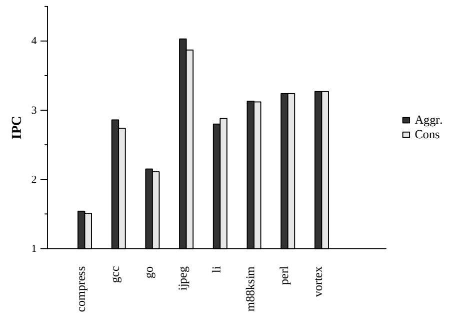


**Figure 11. IPC of the conservative and aggressive spawning policies for 4 thread units.**

Support for multiple speculative threads has recently been proposed as a way to reduce branch misprediction penalty by spawning speculative threads that execute both paths of conditional branches that are difficult to predict ([15][32][34] among others).

Multiple speculative threads can be also effective for a multiprocessor platform, as researched in [11][14][16][28]. In these works, an "always-independent" dependence prediction scheme is used and they do not include value prediction mechanisms.

Finally, predicting data values (see [17][26][35] among others) and data dependences (see [2][7][21] among others) is currently an active research.

## 5. Conclusions

We have presented a processor microarchitecture that supports the simultaneous execution of multiple threads. A clustered microarchitecture, in which most of the resources are local to each thread unit, allows the scalability of the system.

A main feature of the proposed microarchitecture is its capability to exploit speculative thread parallelism from a single-thread program. These speculative threads use otherwise idle resources of the processor, when enough threads are not provided by the compiler/run-time system.

Parallel threads are speculative in the sense that its control flow is predicted as well as their dependences with previous threads and the values that flow through them. We show that data speculation can significantly increase the performance of a multithreaded architecture. Control-flow, data value and data dependence predictors especially oriented to a speculative multithreaded architecture are proposed.

Experimental results show that the proposed microarchitecture can significantly improve the performance of programs that are hard to parallelize, such as the SpecInt95. On average, about 1.6 threads can be executed in parallel for the whole benchmark suite, and for particular programs such as *ijpeg,* this number grows up to 2.6, in a 4-thread unit machine.

The proposed microarchitecture relieves the fetch bandwidth requirements, which is one of the critical issues of multithreaded processors, by fetching the instructions that belong to threads with the same control flow just once and dispatching them to all the involved threads. Several SpecInt95 programs achieved an IPC close to 4 (for one of them was even higher than 4) with a simple fetch engine that can fetch just up to 4 consecutive instructions per cycle.

## 6. Acknowledgements

We would like to thank José González for his insightful comments on this paper. This work has been supported by the Spanish

Ministry of Education under contract CYCIT TIC98-0511, the grant AP96-52274600 and the ESPRIT project MHAOTEU (EP 24942). The research described in this paper has been developed using the resources of the European Center of Parallelism of Barcelona (CEPBA).

## 7. References

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

[2] G.Z. Chrysos and J.S. Emer, “Memory Dependence prediction Using Store Sets”, in *Proc. of the Int. Symp.on Computer Architecture*, pp. 142-153, 1998.

[3] M.N. Dorojevets and V.G. Oklobdzija, “Multithreaded Decoupled Architecture”, *Int. J. of High Speed Computing, 7(3)*, pp. 465-480, 1995.

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

[5] K. Farkas, P. Chow, N. Jouppi and Z. Vranesic, ”The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, in *Proc. of the 30th Int. Conf. on Microarchitecture*, pp. 149-159, 1997.

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

[7] J. González and A. González, “Memory Address Prediction for Data Speculation”, in *Proc. of EURO-PAR 97 Workshop on ILP*, pp. 1084-1091, 1997

[8] J. González and A. González, “The Potential of Data Value Speculation to Boost ILP”, in *Proc. of the 12th Int. Conf on Supercomputing*, pp. 21-28, 1998.

[9] J. González and A. González, “Limits of Instruction Level Parallelism with Data Speculation”, in *Proc. of the VECPAR Conf.*, pp. 585-598, 1998.

[10] L. Gwennap, “Digital 21264 Sets New Standard”, *Microprocessor Report*, 10(14), Oct. 1996.

[11] L. Hammond, M. Willey and K. Olukotun, “Data Speculation Support for a Chip Multiprocessor”, in *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 1998

[12] Q. Jacobson, E. Rotenberg and J.E. Smith, “Path-Based Next Trace Prediction”, in *Proc. of the 30th Int. Symp. on Microarchitecture*, pp.14-23, 1997.

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

[14] I.H. Kazi and D.J. Lilja, “Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors”, in *Proc. of the 12th Int. Conf. on Supercomputing,* pp. 93-100, 1998

[15] A. Klauser, A. Paithankar and D. Grunwald, ”Selective Eager Execution on the PolyPath Architecture”, in *Proc of the Int. Symp. on Computer Architecture*, pp. 250-261,1998.

[16] V. Krishnan and J. Torrellas, “Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor”, in *Proc. of 12th Int. Conf. on Supercomputing,* pp. 85-92, 1998

[17] H.M. Lipasti and J.P. Shen, “Exceeding the Dataflow Limit via Value Prediction”, in *Proc of Int. Symp. on Microarchitecture*, pp. 226-237, 1996.

[18] S. McFarling, “Combining Branch Predictors”, Technical Report #TN-36, Digital Western Research Laboratory, 1993.

[19] P. Marcuello and A. González, “Control and Data Dependence Speculation in Multithreaded Processors”, in *Proc. of the Workshop on Multithreaded Execution Architecture and Compilation* held in conjuction with HPCA-4, 1998

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

[21] A.Moshovos, S.E. Breach, T.N. Vijaykumar and G.S. Sohi, “Dynamic Speculation and Synchronization of Data Dependences”, in *Proc. of Int. Symp. on Computer Architecture*, pp.181-193, 1997.

[22] S. Palacharla, N.P. Jouppi and J.E. Smith, “Complexity-Effective Superscalar Processor”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 206-218, 1997.

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

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

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

[26] Y. Sazeides and J.E. Smith, “The Predictability of Data Values”, in *Proc. of 30th Int. Symp. on Microarchitecture*, 1997.

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

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

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

[30] J. Tubella and A. González, “Control Speculation in Multithreaded Processors through Dynamic Loop Detection”, in *Proc. of 4th. Int. Symp. on High-Performance Computer Architecture (HPCA-4)*, pp. 14-23, 1998.

[31] D.M. Tullsen, S.J. Eggers and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 392-403, 1995.

[32] K.Uht and V. Sindagi, “Disjoint Eager Execution. An Optimal Form of Speculative Execution”, in *Proc. of the 28th Int. Symp. on Microarchitecture*, 1995.

[33] S. Vajapeyam and T. Mitra, “Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 1-12, 1997.

[34] S. Wallace, B. Calder and D.M. Tullsen, “Threaded Multiple Path Execution”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 238-249, 1998

[35] K. Wang and M. Franklin, “Highly Accurate Data Value Prediction Using Hybrid Predictors”, in *Proc of the 30th Int. Symp. on Microarchitecture*, pp. 281-190, 1997.