3

INTERTHREAD DATA DEPENDENCE MANAGEMENT

In this chapter, interthread data dependences are shown to be one of the most limiting factors in the performance of speculative multithreaded processors. Two approaches for dealing with interthread data dependences are presented. The first one consists in synchronizing the consumer and the producer thread. This synchronization requires to identify which are the dependent values, when are to be produced and hardware some to forward them. The second proposal is based on data speculation and tries to predict the values that flow from the spawner to the spawned thread. Different value predictors are analyzed and a new one, that is called increment predictor, is presented. This predictor uses the control-flow taken by the speculative thread to make its predictions. Finally, the impressive performance of the processor with relatively small-sized value predictors leads to the conclusion that value prediction plays a crucial role on such kind of architectures.

3.1. INTRODUCTION

Speculative multithreading has been presented as a technique to reduce the execution time of sequential code by means of partitioning it into speculative threads that run concurrently. These speculative threads are obtained relaxing the constraints for parallelization, in such a way that those threads are both control and data dependents among them. That is, their execution may depend on the execution of the previous threads.

It will be shown in the next Chapter that the performance of speculative multithreaded processors depends on the effectiveness of the partitioning mechanism. Anyway, it is quite obvious that the best places where speculative threads should be created are those where the resultant speculative threads are both control and data independent on the previous ones. Thus, the concurrent execution of these speculative threads will be completely in parallel. However, as it is expected, such kind of speculative threads are almost impossible to find, specially for irregular or non-numerical applications.

Therefore, as independent threads are rarely found, a speculative multithreaded processor has to provide mechanisms to deal with such dependences, both data and control dependences. The way such dependences are managed will strongly affect the performance of such processors. Thus, very conservative mechanisms will override any possible benefit. Besides, speculative multithreaded processors have to provide roll-back mechanisms to return the processor to a safe state in case of misspeculations.

A speculative thread is control dependent on a previous thread if its execution depends on the control flow taken by the latter. The penalty for executing control-misspeculated threads is that the work done by such threads becomes useless and important resources such as the thread units are wasted. For instance, if the processor spawns speculative threads at loop iterations, these threads are control dependent among them since their execution is control dependent on the outcome of the backward branch that closes the loop. If any of the speculative threads does not take this backward branch, a control misspeculation occurs. Then, all the following speculative threads have to be squashed.

The impact of interthread control misspeculations can be significantly reduced by means of a good partitioning policy. Spawning pairs have to meet some criterias to effectively provide speculative thread-level parallelism and one of them is related to control independence: the probability to reach the control quasi-independent point after visiting the spawning point should be very high. In this way, speculative threads are only spawned at parts of the program that are very likely to be executed and most of the speculative threads spawned are correctly control speculated.

On the other hand, a speculative thread is data dependent on previous threads if it consumes any value produced by at least one of them. Thus, a interthread data misspeculation may occur when a speculative thread uses a value before an older thread produces it. Therefore, when an interthread data dependence is violated, the offending thread executes instructions with incorrect values and the results produced by such thread are useless. In this case, the processor may squash the offending thread and its successors. However, it is not necessary to squash the whole speculative thread and more aggressive mechanisms can be implemented. For instance, the processor can just reexecute those instructions of the offending thread that depend on the misspeculated value.

The penalty paid for misspeculating on interthread data dependences depends on the time required for detecting the misspeculation. If dependent values are only verified when the non-speculative thread commits, misspeculations will be detected very late and obviously, the penalty will be very high. Such penalty can be significantly reduced if misspeculations are checked as soon as values are produced.

As for interthread control dependences, spawning schemes can help to reduce the impact of interthread data dependences (shown in the next Chapter). However, analyzing interthread data dependences is more difficult than interthread control dependence. In applications that present low instruction-level parallelism such as irregular and non-numerical applications, data dependences are very common and they are the most limiting factor on the performance of the processor. Therefore, partitioning the program into data quasi-independent threads is almost impossible in such kind of applications and mechanisms to deal with such dependences have to be provided.

In this Chapter, two different proposals to deal with interthread data dependences are presented. The first one forwards the dependent value from the producer thread to the consumer one as soon as it is produced. This mechanism will be referred to as *synchronization mechanism*. The latter proposal is more aggressive and the potential results are also more promising. This proposal consists in breaking the interthread data dependences by means of predicting the values that flow from the producer threads to the consumer ones. If all the data dependent values are correctly predicted for a given speculative thread, it can be executed as if it was independent. The performance of different value predictors in this kind of microarchitecture are studied in this chapter. Besides, a new value predictor, the *increment value predictor*, is presented. This predictor takes into account the control-flow followed by the speculative threads to perform the value predictions. This scheme results in very huge performance results.

This Chapter is organized as follows. Section 3.2 introduces the concept of live-in value and describes different mechanisms to deal with interthread data dependences. The synchronization mechanisms for both

register and memory values are discussed in section 3.3. Section 3.4 analyzes the use of value predictors and presents a new one that is the increment value predictor. Finally, section 3.5 summarizes the main conclusions of the Chapter.

3.2. INTERTHREAD DATA DEPENDENCES

Speculative threads execute different pieces of the dynamic instruction stream in different hardware contexts. In order to produce any benefit, such speculative threads have to use the input values that will be produced by its predecessor threads. That is, speculative threads will only produce a correct result if they are executed with a concrete state of the architected registers and the memory.

A problem occurs when any of these values, either register or memory, are not available at the moment of the thread spawn. In this thesis, two approaches to deal with interthread data dependences based on speculation techniques are studied. The first one is based on data dependence speculation[13][50]. Data dependence speculation allows the speculative execution of instructions without an exact knowledge of the dependences with previous instructions. Those dependences that are unknown are predicted and the code is speculatively executed obeying the predicted dependences, in addition to the known dependences. When actual dependences become known, a recovery action is initiated in case of any dependence violation.

The second approach is based on data speculation ([37][13][64][8] among others). Note that data speculation is different from data dependence speculation. Data dependence speculation predicts the dependences whereas data speculation predicts the value that flows through the dependences. Thus, data speculation is a more powerful technique since it can eliminate the dependence whereas data dependence speculation does not break the dependence chain, it just avoids the execution of an instruction that may cause a wrong execution.

Predicting the initial values for each logical register or any memory location is unfeasible, especially for all the memory locations. However, it is not necessary to know the initial value for all the architected registers or memory locations. For instance, some of them are not consumed by the speculative thread. In this thesis, we will refer to those values that are consumed by a speculative thread and they are not produced by itself as *live-in input values* or *thread input values*.

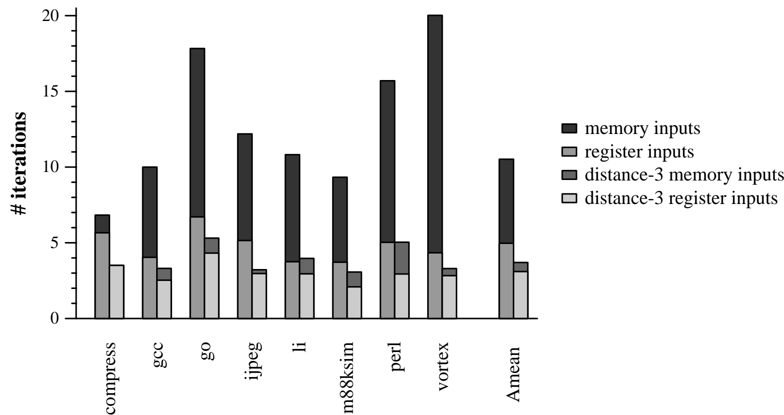


Figure 3.1. Average number of input and live-in input values (through register and memory) for the loop-iteration spawning scheme.

The left bar of figure 3.1 shows the average number of thread input values for each of the programs of the SpecInt95. Statistics have been collected for a 4-thread unit clustered speculative multithreaded processor that spawns speculative threads at each loop iteration. It can be observed that the average number of thread input values is on average greater than 10 and for some benchmarks like `vortex`, these number is considerably high (slightly higher than 20 thread input values).

Nevertheless, there is no need to predict all the thread input values when a speculative thread is created. Only those thread input values that are not available when the speculative thread is to consume it, have to be predicted. In other words, if a given thread input value is already available at the time it is used, the right value will be used regardless of the prediction outcome. That is, the performance of the processor is related to the prediction accuracy of those input values that are produced by the latest previous 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 3 threads).

It can be observed in figure 3.1 (the right-most bar) that the average number of distance-3 values is in average only 3.7, significantly lower than the thread input values. It is also remarkable than the average number of distance-3 memory values has dramatically dropped off to less than 1 and in some benchmarks like `go`, `ijpeg` and `vortex` is almost negligible. The reduction for register values is less impressive and just goes down from 4.8 to 3.1.

This significant reduction in the average number of distance-3 memory values is due to the nature of the spawning scheme. Usually, registers are used to store intermediate computations or values that are being used soon since their access time is much faster than memory. Spawning speculative threads at

innermost loop iterations causes that the instructions executed by the speculative threads are very close in the dynamic instruction stream. So, for this spawning scheme, it is expected that the compiler forwards most of the dependent values through registers rather than for memory. For different partitioning mechanisms where speculative threads are further apart one from the other, the behavior regarding the number of dependences and their distribution between registers and memory locations will change. On the other hand, increasing the number of thread units will also increase the average number of distance-n values per thread.

A significant difference between the meaning of thread input values and distance-3 input values resides in how they can be detected. The distance-3 input values are a subset of the thread input values. Thread input values can be determined statically whereas the distance-3 input values depend on the timing of the dynamic execution.

Regarding memory input values, the identification process is more difficult than for registers. In irregular and non-numerical applications, the compiler is usually unable to statically determine the memory locations accessed. Then, the most common approach is to be conservative and always consider that a dependence exists whenever it can be proved the opposite. This approach may cause that speculative threads are not spawned or have to wait for values they will never consume. The opposite approach is always consider that there are no dependences and, in case of misspeculations, to start a recovery mechanism. Both approaches are very simple and several speculative multithreaded architectures have implemented them, especially the latter one. Some other approaches that uses data dependence speculation, that is, to predict if a memory location is a thread input value that has been produced by a previous thread, or data speculation, to predict the memory value, can be considered.

In the following sections, two different techniques will be investigated both for memory and register dependences. In the next section, a technique based on data dependence speculation is presented for both memory and register dependences and in section 3.4 an approach based for data speculation only for register values is analyzed.

3.3. Synchronization Mechanisms

As it was pointed out in previous subsections, stalling the execution of a speculative thread until the live-in values have been produced and validated by the producer thread may significantly drop off the performance of the processor. Some gain can be obtained if interthread dependent values are forwarded as soon

as they are computed by the producer thread. However, as the forwarded value has not been verified yet, it may be incorrect.

In this thesis, the techniques based on stalling the consumer thread until the value is forwarded from the producer thread are referred to as *synchronization mechanisms*. These mechanisms do not usually require high initialization overhead to be applied. However, the main cost of such mechanisms is the time required to compute the value at the producer thread and to forward it to the consumer thread.

Synchronization mechanisms are based on data dependence speculation. Data dependences are predicted and dependent values are speculatively forwarded from the predicted producer instruction to the consumer thread[20]. Nevertheless, data dependence mispredictions can occur. For instance, if the consumer thread predicts that a instruction is not data dependent on previous threads, it might be executed with wrong input values. Also, the consumer thread may correctly predict the data dependence, but the dependent value is forwarded from a wrong location and the dependent instruction is executed again with incorrect input values. In both cases, data dependence misspeculations requires recovery mechanisms to bring the processor back to a safe state and reexecute the offending instructions with the correct input values. On the other hand, if a speculative thread predicts that an instruction is data dependent and it is really not, a misprediction has occurred, but it is not necessary to recover since the instruction has not been executed. However, the cost of such misprediction is very high too since an independent instruction has been waiting for a value that was not necessary. This instruction is finally issued when the misprediction is detected and this is usually when the thread becomes non-speculative.

Therefore, to implement a synchronization mechanism, three parts are necessary: i) identify the thread inputs, ii) a mechanism to determine which instruction will produce the last value for any thread input value, and iii) hardware support for forwarding the dependent values to the consumer thread.

The mechanisms for identifying the thread input values have been presented in the previous section. Regarding the identification of the instructions that perform the last write for each thread input, -which in this thesis will be referred to as *last-write instructions*,- they can be statically detected by the compiler or dynamically by the hardware[16][84]. For register values, the compiler can easily identify them, even though the different control-flows inside the speculative threads may complicate such detection. For thread input memory values the process is more complex since, in most of the cases, the compiler is unable to statically know which addresses are to be accessed by the speculative threads. Then, memory interthread data dependences as well as the identification of the corresponding last-write instruction can be done through well-known techniques for memory data dependence prediction, such as memory address predic-

tion and pair identification. In this thesis, synchronization mechanisms for memory values are based on memory address prediction[20]. Thus, the addresses that a speculative thread is to access are predicted. Then, such addresses are compared with those that are to be accessed by less speculative threads. If there is a match, a memory dependence is predicted and the last-write instruction is identified.

Finally, hardware support for forwarding the dependent values from the producer thread to the consumer is required. Such hardware is different for register and memory values and depends on the architectural platform of the speculative multithreaded processor. Thus, clustered processors will need mechanisms to communicate the values among the different thread units whereas in a centralized processors values can be shared by the different contexts. In this latter case, it is only necessary to notify to the consumer context that the dependent value is now available and where it can find it.

In the next subsections, different approaches for implementing the synchronization mechanisms are discussed. A case study for a speculative multithreaded processor that speculates on loop iterations is also analyzed.

3.3.1. Identifying the Last Write in a Thread Input Location

Speculative multithreaded processors spawn new speculative threads when a spawning point is reached. The spawned speculative thread starts at the control quasi-independent point associated to that spawning point and then, both threads proceed in parallel. Let denote thread A as the thread that executes the instructions between the spawning and the control quasi-independent point and thread B as the thread that executes the instructions beyond the control quasi-independent point. Also, assume that there are no additional speculative threads running in the processor.

The thread input values of speculative thread B are all the registers and memory locations that are read before they are overwritten. The registers and memory locations that are written before they are read will be referred to as *dead locations*.

Regarding the thread input values of thread B, they can be classified into two groups: those thread input values that are produced by thread A, and those thread input values that are produced before the spawning of the thread B. The thread input values of these latter group can be forwarded directly to thread B at the spawn time whereas the values of the first group have to be forwarded from thread A as soon as they are produced. As only the input values of the first group will affect the performance, the mechanisms presented in this Chapter are only targeted to this group. For short, we refer to it as live-in values and includes any thread input value of a spawned thread that is not available at thread creation.

Thus, the last-write instruction for each of the live-in values have to be identified in thread A. These last-write instructions will be located between the spawning and the control quasi-independent point. As it was previously commented, these last-write instructions can be detected statically by the compiler or dynamically at run-time based on previous executions of the speculative threads.

The compiler can determine which will be the last-write instructions for each live-in register. However, regarding live-in memory values, as it was pointed out, compilers are usually unable to statically determine which addresses are to be accessed and when a dependence will occur. To deal with memory dependences, different static approaches can be considered such as assuming always independence (the most aggressive) or assuming always dependence (the most conservative).

The use of profile information may help the compiler to predict if there are memory dependences. One technique proposed to detect memory dependences consists in keeping track on previous dependences and identify pairs of stores and dependent loads. If a load instruction usually depends on a store performed by a previous thread, the execution of the load should be delayed until the store instruction is performed by the producer thread[50][51]. Then, the value is forwarded from the producer to the consumer thread.

Once the compiler has identified all the last-write instructions for each of the thread input values, it has to mark them. There are different methods to implement that, such as including some special hints to the last-write instructions or introducing new instructions in the instruction set architecture to explicitly forward the values. Such instructions may also be used to forward the rest of the thread input values when the new speculative thread is spawned, that is, the thread input values that are available at the spawn time.

On the other hand, last-write instructions can also be identified with totally dynamic mechanisms. To do that, a table containing information about the instructions executed between the spawning and the control quasi-independent point for each spawning pair is necessary. Thus, each entry of the table contains information for only one spawning pair. Such entries should keep which instructions perform the last write for each of the thread input value. In fact, depending on the spawning policy, it is not necessary that the entries of the table hold information for all the architected registers since at it was shown in figure 3.1 the average number of live-in registers is much lower. For memory values, the entries of the table keep the memory addresses that have been written and the instruction pointers of the store instructions.

For register values, to detect if an instruction in thread A is a last-write instruction for the destination register, its instruction pointer has to be compared with the one stored in the table for the destination register of the instruction.

A similar technique that does not compare the instruction pointer of each instruction can also be considered. In this technique, each entry of the table holds the number of writes for each register that will be performed in thread A. Thus, every time a write is performed on a given register, the counter is decreased and when it becomes 0 and the value is forwarded to the consumer thread.

For memory values, a mechanism based on predicting the addresses that are to be accessed in following executions of the threads is considered. The memory address prediction mechanism is based on keeping track of the effective addresses generated for each static store instruction. Using a simple prediction scheme such as a stride predictor, about 75% of the memory instructions executed by the Spec95 can be correctly predicted.

Nevertheless, the control flow taken to reach the control quasi-independent point after visiting the spawning point may vary among the different executions of the thread and, therefore, the values stored in the table may change from one execution to the other. To avoid that, the processor may allocate different entries for each of the different control-flows. However, this significantly increases the amount of hardware required since the size of the entries is not negligible. Moreover, if the spawning policy has many different spawning pairs, a huge table would be required.

Therefore, it seems that a hardware-only solution is not feasible since it requires a huge amount of hardware to be implemented. Anyway, for spawning schemes that results in a reduced number of spawning pairs that usually follow the same control flow, hardware mechanisms can still be suitable. In the next subsection, a hardware-only mechanism to identify last-write instructions for loop iterations is presented.

3.3.1.1. Case Study: Identifying Last-Write Instructions for Loop Iterations

Spawning speculative threads at loop iterations is a very common scheme to obtain speculative thread-level parallelism. In this case, the processor spawns a new speculative thread every time an iteration starts. Details about the goodness of this spawning policy will be thoroughly analyzed in the next Chapter.

Spawning speculative threads at loop iterations has the main advantage that the processor can exploit some kind of temporal locality. Since speculative threads are spawned for several of the loop iterations, all these speculative threads will use the same spawning pairs. In this case, we will refer to the table that contains information about the spawning pairs as *loop iteration table*[77]. The temporal locality of this loop

iteration table can be observed in Figure 3.2. This figure shows the hit ratio of an 8-entry loop-iteration table in a speculative multithreaded processor with 4 thread units. It can be observed that on average, the hit ratio of the loop iteration table is about 90%.

A speculative thread can be uniquely identified by the spawning pair and the direction of the branches performed within it. In the case of loops, the instruction pointer of the spawning and the control quasi-independent point is the same, so the identifier consists in the instruction pointer of such instruction and the control flow taken. Then, this table is indexed by the thread identifier (some hash function of the instruction pointer of the spawning point and information regarding the control-flow that will be taken by this thread) and each entry contains the following fields:

- LNRW (Last Number of Register Writes): This field is an array of a number of entries equal to the number of architected registers. For each register it indicates the number of times that it was written in the last execution of the iteration. In fact, it is not necessary to take information for all the registers since the number of written registers per loop iteration is about 5.
- CNRW (Current Number of Register Writes): This field is similar to the LNRW but it refers to the current execution of the loop iteration, which has not finished yet.
- LSA (Last Store Addresses): This field is an array with a number of entries equal to the number of store instructions in the last iteration of the loop. The functionality of this field is to detect interthread memory dependences through address prediction. Memory addresses are quite predictable and each entry of the LSA has the instruction pointer of the store instruction, the last address accessed and the stride among consecutive accesses. Then, the number of writes a memory location will have in an iteration can be calculated from this array.

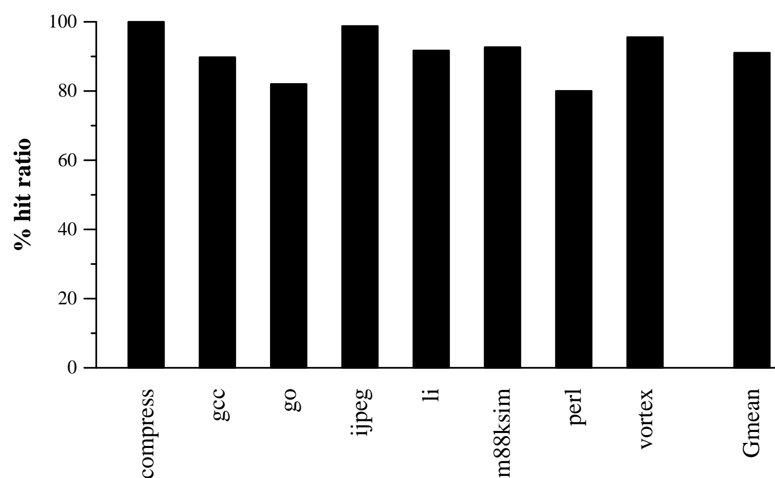


Figure 3.2. Hit ratio of the loop iteration table with 8 entries.

- CSA (Current Store Addresses): This field contains the memory addresses of the store instructions executed so far in the current iteration.
- C (Confidence): This field assigns confidence to the predictions done using the previous fields. In some way, this is similar to assigning confidence to a branch predictor. It can be implemented in different ways (for instance, a 2-bit saturating counter among others). This field avoids to spend resources when the data required by a speculative thread is not highly predictable.

When a thread that is in the table finishes its execution, that is, it has reached the control quasi-independent point, the predictability of the thread has to be determined. Thus, a loop iteration is predictable if:

- LNRW is equal to CNRW,
- LSA and CSA have the same number of elements,
- The addresses accessed in the CSA are equal to the previous addresses accessed that are stored in the LSA plus the stride.

If all these conditions are true, the C field is increased; otherwise it is decreased. A speculative thread is considered predictable is the most significant bit of C is set. At the end of a loop, CNRW is copied into LNRW and the LRNW is reset, the addresses stored in the CSA are also stored in the LSA and the strides are recomputed.

Thus, when a spawning point is reached, it is checked if the corresponding loop iteration is in the table. If not, the speculative thread is not allowed to be spawned and a new entry is allocated in the table. If information about that loop iteration is in the table, the thread is spawned and the information stored in the LNRW field is used to determine when the last write on a dependent register will occur.

3.3.2. Forwarding Register Values

The implementation of the register forwarding mechanism will depend on the architectural platform of the speculative multithreaded processors and the way the register file is implemented. For instance, having a centralized and huge register file with local register map tables provides lower communication latency but the access time to the registers is increased. On the other hand, a clustered register file reduces the access time but it requires the forwarding of the values from one cluster to the other.

In the following subsections, approaches to forward dependent register values for both of architectural platforms studied in the previous Chapter are presented. Assume in both cases that last-write instructions

are known. Finally, a model to forward register values based on the hardware approach explained for loop iterations is also analyzed.

3.3.2.1. Register Forwarding for Centralized Speculative Multithreaded Processors

In this subsection, the case of a totally shared register file is investigated. The case of partitioning the register file for the different speculative threads will be studied in the next subsection for clustered processors.

In this case, the processor has a huge register file that is shared by all the threads. Each thread has its own register map table with as many entries as number of architected (also called logical) registers that indicates the current physical register allocated to the corresponding logical register. Notice that each map table reflects a different assignment of logical to physical registers, corresponding to a different point of the execution.

For those logical register whose value is not available, their corresponding entry in the register map table contains a special value, NIL, that indicates that this logical register is not currently mapped to any physical register. Also, there are additional bits for each entry in the register map table. These bits denote whether the first operation performed over the register has been a read or a write.

When a speculative thread is spawned, the register map table of the spawned thread is initialized with the same values as the register map table of the parent thread. The entries that correspond to those logical registers that will be written between the spawning and the control quasi-independent point are initialized to NIL. Both read and write bits are cleared for all the entries of the register map table of the spawned thread.

Let us assume again that thread A is the thread that executes instructions between the spawning and the control quasi-independent point and thread B is the one that executes instructions beyond the control quasi-independent point. Thus, when a last-write instruction is found in thread A, the corresponding entry in the register map table has to be copied to thread B. If thread B entry contains the value NIL, the pointer to the physical register is copied and all the instructions that are waiting for this operand can start their execution as soon as they are chosen by the selection logic. If this entry contains a different value from NIL but it has the write bit set, that is, the first operation performed by the speculative thread over that register was a write, no operation is done. However, if the read bit is set, a misspeculation has occurred and a recovery mechanism has to roll back the processor to a safe state and all the dependent instructions have to be reexecuted with the correct value.

When thread A finishes its execution, it is checked whether there is any entry in the register map of thread B that contains the value NIL. If so, the corresponding entry of the register map is copied from the previous thread to the following one.

Finally, the spawning model affects the required hardware for forwarding register values. In sequential thread ordering, speculative threads are created in program order and only the most speculative thread is allowed to spawn a new one. This kind of ordering fits very well with the mechanism described in this subsection and it can be implemented with no modifications.

Nevertheless, sequential thread ordering strongly constrains the ability of speculative multithreaded processors to exploit speculative thread-level parallelism. The unrestricted spawning model allows any speculative thread to create new speculative threads and it potentially can exploit more speculative thread-level parallelism. However, the register forwarding logic becomes more complex for this spawning model than for the sequential thread ordering.

When a thread spawns a new speculative thread and there are some other speculative threads between the spawner and the spawned thread, it is possible that any of the thread input values for the new spawned speculative thread have been already computed by any of the intermediate threads. Then, two solutions can be considered. The former one consists in initializing the register map table in the same manner as for the sequential thread ordering. In this case, thread input registers that have already been forwarded to future threads are to be received by the spawned thread when it becomes the non-speculative. That is, when all the intermediate threads have committed their values it is realized that the new non-speculative thread has an entry in the register map table initialized to NIL.

The second solution consists in looking up if any of the intermediate threads has already written in a thread input register and has attempted to forward it, (i.e., it has performed a last-write instruction.) In this case, the corresponding entry of the register map table is initialized to the entry of the register map table of the speculative thread that has computed it. To do that, it is necessary an additional bit in the register map table that indicates whether a last-write instruction has been performed for each logical register.

Finally, the logic to free a physical register has also to be modified. The rename logic allocates a new physical register to a logical register each time an instruction that is to write on it is decoded. The previous physical register assigned to this logical register is stored in the reorder buffer. When this instruction is committed, the previous physical register is freed for a future use. In our case, different threads may share the same physical register and the processor has to avoid that a physical register that is still in use by other

threads become free. To do that, adding a counter for each physical register that indicates the number of threads that are working with it can be considered. Thus, when an instruction is committed, the counter associated to the physical register is decreased and when it becomes 0, then the physical register is freed.

A different approach consists of checking if a register is in use by any other speculative thread. This happens when the corresponding entry of the register map table is pointing to such physical register. If so, the physical register cannot be freed and its deallocation will be done when such thread is committed. Otherwise, the physical register is deallocated.

3.3.2.2. Register Forwarding for Clustered Speculative Multithreaded Processors

Clustered speculative multithreaded processors are made up by several thread units. Each of them are similar to a superscalar core, in such a way they have their own functional units, their own instruction window and their own register file. The main advantage of having a distributed register file is the reduction in the access time. The major drawback of such kind of architectures is the impossibility to physically share values. Thus, clustered designs require that dependent values have to travel through the interconnection network from the producer thread unit to the consumer one.

Let assume again thread A spawns thread B. Then thread A executes the instructions between the spawning and the control quasi-independent point and thread B the instructions beyond the control quasi-independent point. Thus, when thread B is spawned, it is allocated in an idle thread unit and the context for this thread is initialized. This initialization includes the register file and the register map table of the assigned thread unit.

The simple way to initialize both structures is just copying the contents of both tables from the thread unit of thread A to the thread unit of thread B. In this way, those values that are not to be produced later by thread A are copied to the register file of thread B.

The live-in registers of thread B that are not available allocate a physical register but are marked as not available.

An approach to reserve entries in the register file for live-in register values is to maintain the positions they actually have in use in the parent thread. Besides, each physical register has a reservation bit that indicates whether it contains a valid value. The major drawback for this approach is that when values are produced and forwarded to the consumer thread, the value has to be stored in the physical register allocated. Therefore, when a value is received by a thread unit, the register map table has to be accessed to know at which physical register the value has to be stored.

A second approach is to use a second register file for the incoming register values. This secondary register file is referred to as *live-in register file*. This live-in register file has as many entries as logical registers. However it can also be implemented with a fewer number of entries since the number of dependent values is usually low. When a speculative thread is created, the register file and the register map table are copied from the parent thread excepting those positions of the register map table associated to the live-in register. Those positions are initialized pointing to the corresponding entries in the live-in register file. In this way, when a last-write instruction is found in the thread A, the value of such register is forwarded to thread B and the value is stored in their corresponding entry of the live-in register file.

Write and read bits are also used for detecting register dependence misspeculations. A misspeculation occurs when a thread consumes a value before it is forwarded from a previous thread. Therefore, a misspeculation can be detected if a value is forwarded form the producer thread to the consumer one and thread B has already consumed it. To implement the mechanism to detect misspeculation a new bit is necessary for each entry of the register map table which is the dependent bit. This bit indicates if the corresponding entry has been initialized to an entry of the live-in register file. Therefore, a misspeculation occurs when a thread unit receives a value which logical register has the dependent bit cleared. Besides, a misspeculation occurs if the dependent and the read bit are set and the corresponding entry of the live-in register file has the reservation bit set. This happens when a valid value has been previously stored in the live-in register file and it has been read.

However, the main difference between the centralized and the clustered design resides in the way instructions that try to overwrite a thread input register are treated. It was commented in the previous subsection that in centralized processors, such instructions are not allowed to be renamed since live-in registers do not have any physical register allocated until the producer is renamed. Therefore, dependent instructions do not know from which location they have to read the value when it becomes available. However, in clustered processors live-in registers already have a physical location allocated. Thus, instructions that depends on a live-in register can be hold in the instruction window waiting for the value in the same manner conventional instructions wait for register values that have not been produced yet. Therefore, instructions that overwrite a live-in register can be renamed normally.

Obviously, this technique can also be applied to the centralized register file. To implement it, the spawned thread can allocate as many registers as live-in registers at thread creation. Nonetheless, forwarding values becomes more complex. Now, the value has to be copied to its corresponding physical location instead of just copying the pointer from the register map table of the producer thread to the following one.

Regarding the thread spawning model, the same problem analyzed for the centralized version of the speculative multithreaded processor may occur here. The hardware support needed for allowing unrestricted thread ordering creation in clustered processors is very similar to the one needed for the centralized version but in this case, the value has to be copied from the producer thread unit to the thread unit allocated for the spawned speculative thread.

3.3.2.3. Case Study: Register Forwarding for Loop Iterations

In subsection 3.3.1.1 a hardware mechanism to identify the last-write instructions for thread input values for the loop-iteration spawning scheme was presented. This method consists in counting the number of writes performed at each storage location by a loop iteration. The execution model of this spawning scheme works in a similar way for both a centralized and a clustered design, even though it fits quite well into a clustered processor whose thread units are interconnected by means of an unidirectional ring.

Regardless of the topology and the implementation of the processor, the forwarding mechanism for register values is quite similar. In this subsection a centralized processor is assumed. However, note that the implementation in a clustered processor will be very similar.

Assume a centralized processor with a huge register file shared by all the contexts. Each context has its own register map table. We will refer to these tables as *Rmap*, with different subindices to identify different threads when necessary. A Rmap entry *r* may contain a special value, NIL, that indicates that logical register *r* is not currently mapped to any physical register. In addition, each thread has another table, which is called *Rwrite* (register write table), that contains for each logical register the number of remaining writes to that register.

Let us assume again thread A and thread B; thread A spawns thread B and it executes the instructions between the spawning and the control quasi-independent point whereas thread B executes the instructions beyond the control quasi-independent point.

To explain how these tables work, let $Rmap_A$ and $Rmap_B$ denote the register map table of the thread A and the register map table of the thread B respectively. Similarly, $Rwrite_A$ and $Rwrite_B$ refer to the register write tables.

When a speculative thread is spawned, the $Rwrite_A$ is initialized with the values of the LNRW field of the corresponding entry in the loop iteration table. For each logical register *r*, if $Rwrite_A[r]$ is zero, then no writes are expected to register *r*. In this case, $Rmap_B[r]$ is initialized with $Rmap_A[r]$. In other words, if no writes are expected to a given logical register, all the threads will share the same physical location for that

register. As it was mentioned previously, each physical register has a reservation bit that indicates whether it contains a valid value. When a new free physical register is allocated, its reservation bit is reset and when a result is written into the register, the reservation bit is set. If $Rwrite_A[r]$ is zero, then all instructions of the spawned thread that have register *r* as a source operand are allowed to be issued as soon as the register $Rmap_A[r]$ is written (it could have already been written when the speculative threads are created). If $Rwrite_A[r]$ is not zero, then $Rmap_B[r]$ is initialized to NIL.

When a thread decodes an instruction *with* destination logical register *r*, a new free physical register is selected and its identifier is stored into Rmap[r] (if there are no free registers, the thread is stalled). However, if the Rmap[r] is equal to NIL, this instruction is not allowed to be renamed and it has to stall.

On the other hand, when an instructions with destination register *r* is committed, the previous physical register to which register *r* was mapped is freed and Rwrite[r] is decreased. Assume that thread commits an instruction whose destination register is r. Depending on the value of $Rwrite_A[r]$ the following actions are taken:

- If $Rwrite_A[r]$ is equal to 0, then the current instruction is expected to be the last one that writes to logical register *r*. If $Rmap_B[r]$ is equal to NIL, then $Rmap_A[r]$ is copied into $Rmap_B[r]$. In other words, the free physical register assigned to *r* can be shared by threads *A* and *B*. Note that the reservation bit of such physical register is reset. When the instruction of thread *A* writes into it, it will be set and instructions of thread *B* that have that physical register as a source operand will be allowed to issue. In this way, the synchronization between two threads required to obey a data dependence through a register is implemented using the conventional mechanisms already present in most superscalar processors. Note also that if $Rmap_B[r]$ is not equal to NIL and the write bit is set, thread *B* must have found a write to register *r* before a read. In this case, it is obvious that the new register allocated by thread *A* must not be shared with thread *B*. If the read bit is set and $Rmap_B[r]$ is not equals to NIL, a misspeculation has occurred.
- If $Rwrite_A[r]$ is lower than 0, then the current thread is going to perform a non expected write to register *r* and a misspeculation has occurred.

When thread *A* finishes, if some entry *r* of the Rwrite table is greater than zero, then the thread has performed less writes to register *r* than predicted. If there is a subsequent thread executing the next iteration and $Rmap_B[r]$ is equal to NIL, then $Rmap_A[r]$ is copied into $Rmap_B[r]$. That is, the free physical register to which logical register *r* is mapped at the end of thread *i* becomes visible to the next thread since it contains the last value written by thread *i* onto that logical register.

When a speculative thread *B* is squashed, all physical registers allocated to it that are not shared with the previous thread are released. That is, for each logical register *r*, if $Rmap_B[r]$ is different of $Rmap_A[r]$, then $Rmap_B[r]$ is freed. In addition, for each instruction currently in the local reorder buffer with a register destination, the previous mapping for such register is also freed.

It can be observed that this model of forwarding register values only works for the sequential spawning model. It can also be adapted for the unrestricted spawning model even though it becomes much complex.

3.3.3. Forwarding Memory Values

Thread memory input values are more difficult to detect than registers. In many cases, compilers are unable to statically determine the memory addresses that will be accessed by the program at execution time. Mechanisms proposed to detect the last-write instruction on which a load depends are speculative and thus, hardware mechanisms to detect interthread memory dependence violations have to be implemented. A violation occurs when a thread performs a store on a memory location that has been read by a more speculative thread.

In addition to the problems to detect dependent memory thread inputs, memory instructions have an special behavior on speculative multithreaded processors. Only the non-speculative thread is allowed to write in main memory whereas speculative threads must keep all their stored values locally. Then, hardware has not only to provide support for forwarding dependent memory values from one thread to the other but it also has to provide support for detecting misspeculations and keeping the speculative memory state for the speculative threads. If a thread does not have space to store its speculative memory state, this thread either has to stall its execution until it becomes the non-speculative thread or has to be squashed.

In fact, these requirements are the same as for register forwarding. However, register misspeculations are not as frequent since compiler can almost totally determine the interthread data dependences through registers.

Different solutions to provide support for keeping different versions of the same memory locations for each speculative thread can be considered. The simplest one is based on a local memory for each of the speculative threads. This solution fits very well into the clustered implementation of speculative multithreaded processors. However, the values stored in these local caches cannot be written into main memory until the thread becomes non-speculative. For a centralized processor, the same solution can be applied. Thus, different caches can be used for each of the contexts to store the local memory values. The partition into local caches can be physical, that is, the processor has as many local caches as contexts it may support

or logical, that is, threads are only allowed to access to a subset of a big cache. In this latter case, the cache can be partitioned statically (each speculative thread has a fixed part of the cache) or dynamically (cache space is assigned on demand).

A problem that appears with this solution of maintaining different caches is the peak traffic produced when a thread is committed. When the non-speculative thread commits, it has to free its context. However, before freeing the local cache, all the dirty lines of the local cache have to be copied into the main memory. As it was commented in the previous Chapter, a different solution can be taken. Instead of flushing the local cache into main memory when a thread is committed, the peak traffic can be avoided if the dirty lines are marked with a special tag. Then, when the new thread requires to replace a line, that is marked as committed, it updates the main memory. In this way, the peak traffic is reduced even though it is more complex to detect memory misspeculations.

In addition to mechanisms for storing the speculative values, speculative multithreaded processors have to provide support for forwarding dependent memory values from the producer to the consumer thread. Fortunately, similar problem is present on multiprocessors for exploiting non-speculative thread level parallelism and lots of works have been proposed to solve this problem.

Memory consistency protocols are mechanisms that have been proposed to deal with memory dependences in multiprocessors with distributed caches. In order to correctly execute parallel processes in a multiprocessor, memory operations have to be performed in some order that guarantee that the final state of the memory is the same that if the process has been executed sequentially. That is referred to as sequential consistency[35]. Sequential consistency may be either strong or weak. Strong consistency denotes that memory operations have to be performed in the same order than for the sequential execution whereas weak consistency means that the order may differ from the sequential one, but it does not damage the sequential operation. That is, the memory operations of the different concurrent threads can be executed in any order among them if they do not affect to the behavior of the sequential execution.

Hundreds of works have been proposed to maintain the sequential memory ordering. Most common is based on sending to the memory bus information about the addresses accessed by each thread. Processors snoop the bus to know if such information may affect their execution. In our case, if a thread needs to load a value from a memory location and such value is not present in the local cache, it sends a request to the threads that are executing previous code. If any of them has the value, it is sent to the requester thread. If not, the value is read from main memory. On the other hand, when a thread stores a value on a memory

location, it notifies to the following threads. If any of the successor threads have loaded the value, then a dependence violation has occurred.

3.3.3.1. Case Study: Memory Forwarding for Loop Iterations. The MultiValue Cache

Spawning threads at innermost loop iterations is implemented by means of a loop iteration table. In this case, the information related to memory addresses accessed by each iteration is held in the CSA and LSA fields. In the LSA field, in addition to store the last address accessed by the last iteration, the delta between of the two last memory addresses is stored.

Then, the number of times a memory position will be accessed by a speculative thread can be easily computed. In this way, the processor can try to make use of such information to reduce the amount of traffic of the classical snoop consistency protocol. Load instructions that are to access to memory positions that are to be written by less speculative threads are not issued until the value is produced.

This can be achieved with a special first level data cache. This cache has some particularities and it will be referred to as the *Multi-Value (MV)* cache. The distinguishing feature of a MV cache (see figure 3.3) is that its data words are replicated for each context (maximum number of threads). In this way, each thread may have a different view of the contents of memory. Another important feature is that it stores non committed values and does not allow to modify the next memory level (here called the L2 cache) until they are committed. This is implemented by means of a write-back policy together with a particular replacement scheme (explained below).

For each replicated word, the MV cache contains two additional fields: the number of writes that the corresponding thread is expected to perform (NW) and a bit indicating whether it contains a valid value for the corresponding thread (V).

When a thread creates a speculative thread, the LSA field of the loop iteration table allows to predict the addresses into which the spawner threads are going to write. For each predicted write address, the cor-

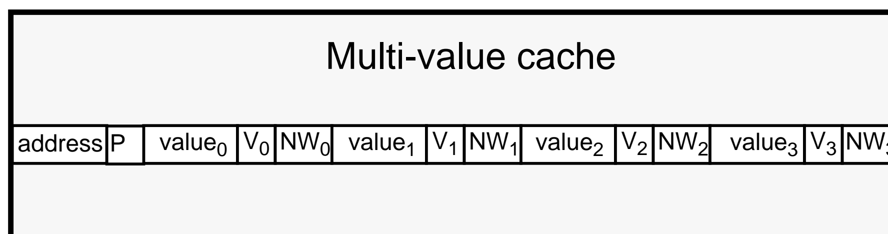


Figure 3.3. :The multi-value cache for a SM processor with four thread units.

responding line is allocated in the MV cache if not present. If some thread has not enough entries in the MV cache, it cannot be speculated. When a new cache line is allocated due to a predicted write of a thread, its contents is initialized with the current data (obtained from L2) for all the multiple copies. The NW field of the corresponding thread is set to 1 and the V fields of the thread and the preceding ones are set whereas the V fields of succeeding threads are reset. If the line is already in the MV cache, the NW field of the thread is increased and its V bits of the succeeding threads are reset.

Then, each thread executes memory instructions out-of-order using a total disambiguation scheme. That is, memory instructions compute their effective address as soon as their operands are available and then are sent to a load/store buffer. Stores write to memory when all the previous instructions of the same thread have completed whereas loads read from memory when the addresses of all previous stores are known. If the load matches a previous store address the store data is forwarded to the load destination register; otherwise the read is performed from memory.

When a thread performs a read from memory, the MV cache is checked first. If the corresponding data line is in the MV cache, it will contain a different copy for each thread. If the data corresponding to that thread has its V bit set, then this value is read. Otherwise, the load is cancelled and stored in a *load wait* queue. Loads from this queue are tried again in idle cycles of the MV cache. If the corresponding cache line is not in the MV cache, the data is read from L2. Optionally, if there is a candidate for replacement, a new line can be allocated into the MV cache, with all its valid bits set and the NW fields equal to zero (this will speed up further accesses to the same line).

When a thread performs a write and the corresponding line is not in the MV cache, a misspeculation may have occurred because this write was not predicted. A recovery action is initiated which may consist of a partial or a full squash. A more powerful solution would be to keep track of memory reads operations to exactly determine if a misspeculation has really occurred. The new line is brought into the MV cache with all the V bits set and all the NW fields equal to zero.

If the line corresponding to the written data is in the MV cache, its NW field is decreased. If it becomes zero, the data is copied to succeeding threads, from the next one to the first that has either NW or V different from zero (excluded). In addition, if the following thread has the V bit reset, it is copied also into that thread. The V bits of these threads are set. If it becomes negative, a misspeculation may have occurred.

When a thread finishes (or is squashed), if the corresponding NW field of any line of the MV cache is greater than zero it is reset to zero, and the value is propagated to succeeding threads as in the case when

the counter becomes zero. This occurs when some predicted write did not actually occur. In this case, all dependences have been obeyed but there may be loads of succeeding threads waiting for a non existent write.

A line of the MV cache can be considered for replacement only if all its NW fields are equal to zero (this will be always the case when all the speculative threads have finished). If the line is dirty, it is considered for replacement if in addition there are not speculative threads. This ensures that the L2 cache is only updated with committed values. Deadlock is guaranteed not to happen since new lines are only necessary to be allocated at speculative thread creation. During speculative thread execution, bringing additional lines into the MV cache is an optional feature that will increase performance.

To reduce the pressure on the L2 cache when speculative threads are created, the MV cache could just initialize the V bits and add a single bit per memory address, which is called *presence bit* (P), that indicates whether the data has been brought from L2. Initially this bit is reset and when a read operation finds the corresponding V bit set but the P bit reset, the data is read from L2 and propagated to succeeding threads in the same way as when the NW field becomes zero after a write.

The MV cache can also be implemented in a distributed way. This implementation requires that when a speculative thread is spawned, all the memory locations with the NW greater than 0 of previous threads allocate a new position in the local MV cache and set their V field to 0. This is done in order not to read from this memory location until the value has been produced by the corresponding thread.

3.3.4. Related Work

Some speculative multithreaded architectures have used synchronization mechanisms to deal with interthread data dependences, especially for memory values. Regarding interthread register data dependences, the Multiscalar uses a distributed register file[3] to store the different speculative states. Such register files are divided into three parts, the current register file that holds the current register values for that task, the previous register file that holds the values produced by the previous tasks and the subsequent register file to store the values of the succeeding tasks for future allocated tasks. Dependent register values are bypassed from the producer thread to their successors when a task has performed the last-write operation on it.

Similarly, the Synchronizing Scoreboard[34] proposed by Krishnan and Torrellas considers a clustered version of the register file but the consumer thread is allowed to initiate the forward operation if the value has been produced before the speculative thread is spawned. To do that, for each logical register the local scoreboard has additional bits to know if the value is already available for the successor threads. Thus, the

consumer thread knows if the requested register has been produced. This information is replicated for all the scoreboards in the thread units.

Regarding interthread memory dependences, the use of data dependence speculation has been deeply studied to speculate on dependences through memory since dependences through registers are easily identified by both the compiler and the hardware. Memory references whose effective address are unknown are usually called ambiguous references. When memory instructions are executed out-of-order, a memory reference may be performed before the effective address of all previous references are known, that is, it is performed without completely disambiguating the reference. This scheme is used by the address resolution buffer of the Multiscalar[17] and the address reorder buffer of the HP PA8000[28] among others. Both approaches use a very simple speculation heuristic: they assume no dependence between an instruction and any previous instruction whose effective address is unknown.

The Address Resolution Buffer may be implemented centralized or clustered and is used to detect misspeculations. A misspeculation occurs when a task writes to an address that has been read by a previous task.

Another approach is the Speculative Versioning Cache by Gopal *et al*[23]. This proposal uses distributed caches for the different thread units and uses the snoopy bus-based coherence protocol to detect when a previous task produces a value on a location that has been already read by a more speculative thread. In that work, the problem of having to completely flush the local caches when a thread commits is also studied. To avoid the burst traffic when a thread is committed, the values are not written into memory but all the lines are marked as dirty committed state. Those cache lines are written to memory when the local thread requires that line. To avoid that speculative threads that have written to the same memory location store the value in different order, a new bit per cache line is necessary (the stale bit).

Another mechanism proposed to detect memory misspeculations is based on the MDT (Memory Disambiguation Table) and was initially proposed by Krishnan and Torrellas for an on-chip multiprocessor [34]. This table may be incorporated to the L2 shared cache and it would be similar to a directory even though due to its small size, it can be considered separately. This table stores for each accessed address which operations have been performed by all the concurrent threads in the processor. For bigger systems, a clustered version, the GMDT, has also been proposed by the authors in [5].

Unlike previous models that are hardware-based schemes for detecting memory dependence violations, Rundberg and Stenström proposed a software approach to deal with interthread memory depen-

dences[62]. To do that, the compiler associates to each shared variable a data structure to allow multiple threads to access to it. When memory dependences cannot be disambiguated, speculative loads and a speculative stores use this structure.

3.3.5. Performance figures

Next figures correspond to the execution of a clustered speculative multithreaded processor and with perfect synchronization mechanisms, that is, there is a perfect last-write instruction prediction and a perfect dependence prediction for memory values.

The spawning policy is the loop iteration spawning scheme, that is, speculative threads are spawned for each loop iteration.

Performance statistics are obtained through trace-driven simulation of the whole SpecInt95 benchmark suite. 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[70]. For the statistics, we simulated 300 million of instructions after skipping initializations. The programs are executed with the *ref* input data.

The clustered speculative multithreaded processor has 16 thread units and each one has the following features:

- Fetch: up to 4 instructions per cycle or up to the first taken branch, whichever is shorter.
- Issue bandwidth: 4 instructions per cycle.
- Functional Units (latency in brackets): 2 simple integer (1), 2 memory address computation (1), 1 integer multiplication (4), 2 simple FP (4), 1 FP multiplication (6), and 1 FP division (17).
- Reorder buffer: 64 entries.
- Local branch predictor: 14-bit gshare[40]. Local branch prediction tables are assumed to be copied from the parent thread to the new spawned thread at spawning time.
- 32 KB non-blocking, 2-way set-associative local, L1 data cache with a 32-byte block size and up to 4 outstanding misses. The L1 latencies are 3 cycles for a hit and 8 cycles for a miss.

In the next figures, the cost of spawning threads is assumed to be zero. Memory dependence violations are detected by means of a MultiVersion Cache based on the Speculative Versioning Cache[23].

The delay of forwarding a value from the producer thread unit to the consumer is assumed to be 3 cycles for memory values and 1 cycle for registers. In the ring topology, the penalty considered for bypass-

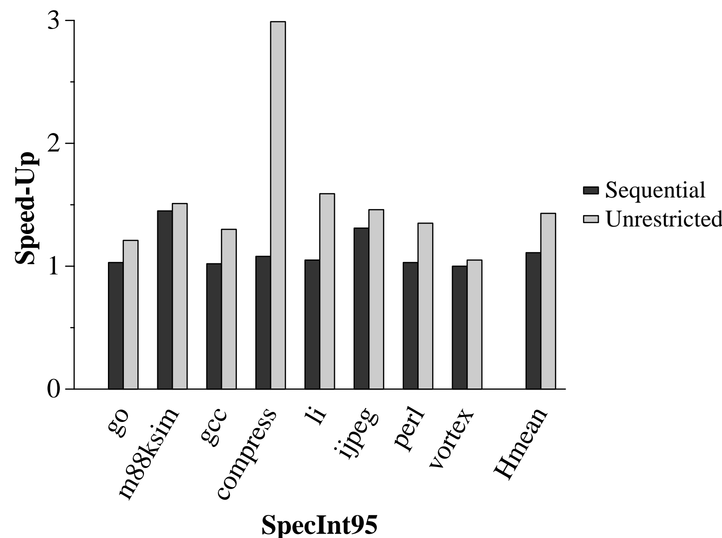


Figure 3.4. Speed-up over single-threaded execution of a clustered speculative multithreaded processor with 16 thread units and perfect synchronization mechanism.

ing the value is 1 cycle per hop. Larger delays for bypassing register values can be considered. However, the major cost of this mechanism is not the penalty cost of forwarding the value from one thread unit to the other but waiting for their computation in the producer thread

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

Figure 3.4 shows the speed-up achieved by the synchronization mechanism. As it is expected, the highest results are achieved by the unrestricted thread ordering since it can potentially exploit more speculative thread-level parallelism. However, the differences among them are not so significant. On average, the unrestricted thread ordering outperforms the sequential thread ordering just by a 30% and the most of this improvement is due to `compress`.

Nevertheless, the most important conclusion that can be extracted from that figure is that a clustered speculative multithreaded processor with 16 times more resources than the superscalar processor only achieves for the most aggressive spawning scheme a speed-up of 43% and only 11% for the sequential thread ordering. Besides, some benchmarks like `vortex` only have a 5% improvement for the unrestricted thread ordering and perfect synchronization mechanisms.

As it will be shown in next Chapter, interthread data dependences can be reduced by selecting different spawning pairs. However, it seems that for applications like SpecInt95 which present low degrees of instruction level parallelism, the results obtained with other spawning policies do not significantly vary from those presented for the loop-iteration scheme.

Thus, it seems that refining the synchronization mechanisms to reduce the possible misspeculations is not worthy since its potential is very low. The main conclusion of this subsection is that interthread data dependences strongly affect the performance of speculative multithreaded processors and more aggressive mechanisms to deal with them have to be proposed in order to make use of the resources of the processor.

3.4. VALUE PREDICTION

SpecInt95 benchmarks are a set of applications that usually provides low degree of both thread and instruction level parallelism. It has been shown on several works that data dependences strongly affect their performance.

Synchronization mechanisms are good to ensure the correctness of the speculative execution of the applications. Instructions of speculative threads do not start their execution until their operands have been produced. Thus, the performance that speculative multithreaded processors can achieve with this methods is limited by the critical path of the data dependence chain. To achieve performances beyond the upper bound due to data dependences, a mechanism that breaks the serialization imposed by data dependences is required.

Data value speculation is a technique that has been proposed to relieve the cost of this serialization and to boost up the performance of superscalar processors. This technique is based on the observation that values tend to repeat or follow a known pattern over a large fraction of time. With appropriate mechanisms such values can be correctly predicted. Predicting the input/output operands of instructions before they are available allows the processor to start the execution of those instructions and their dependent ones speculatively. If the prediction is not correct, the predicted instruction and its dependent ones have to be reexecuted with the corresponding correct value. However, if the value is correctly predicted, the work related to such instructions has been anticipated and an improvement on the performance is produced. Also, breaking the data dependence graph increases the number of eligible independent instructions to be selected for the issue logic and produces a better usage of the resources of the processor.

Some studies on value prediction have shown that the performance potential of this technique in superscalar processors approaches to a linear function of the hit ratio of the predictors. Therefore, the cost of improving the value predictor does not justify the performance improvement achieved. On the other hand, such study concludes that the performance potential of this technique is greater in other platforms such as multithreaded processors[22].

In the previous section it has been shown that the performance achieved by a speculative multithreaded processor with a perfect synchronization mechanism hardly achieves a 40% improvement with 16 times more hardware than a superscalar processor. Such poor results are due to the fact that speculative threads have to wait for the computation of the dependent value. Value prediction may help to break such dependences and if the values that flow from one thread to the other are correctly predicted, then both threads can proceed in parallel as if they were independent.

Figures 3.5 shows the performance potential of speculative multithreaded processors with perfect value prediction with 4 and 16 thread units. In both cases, perfect value prediction is applied for the sequential and the unrestricted thread ordering and for register and register and memory values. However, the hardware for synchronization is required for early detect mispredictions. Statistics are obtained for the loop iteration spawning scheme.

As it is expected, the most aggressive thread ordering benefits more from value prediction than the sequential thread ordering. In fact, the sequential thread ordering scheme achieves very poor speed-ups anyway, lower than 2 when all live-in values are correctly predicted for the 16-thread unit configuration. For the unrestricted thread ordering, only predicting register values achieves speed-ups over single-threaded execution higher than 2 for the 4-thread-unit configuration and close to 4 with 16 threads. Predicting both register and memory values can boost the performance up to 2.7 with 4 thread units and higher than 6x for 16 thread units on average.

Therefore, these impressive averages demonstrate that value prediction may help speculative multithreaded processors to exploit high degrees of speculative thread-level parallelism. In this section, the

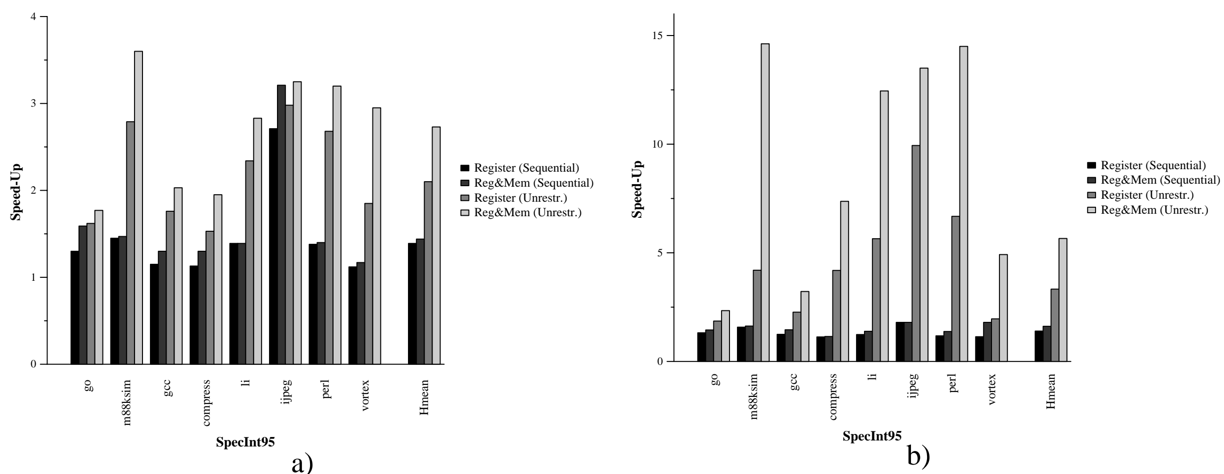


Figure 3.5. Performance potential of a speculative multithreaded processor with perfect value prediction for register and memory values and both thread ordering schemes for a) 4 thread units and b) 16 thread units.

impact of realistic value predictors only for register values are considered. Moreover, a new value predictor specially designed for these architectures, the increment value predictor is presented.

3.4.1. Related Work

Value prediction is a technique that has been largely studied in the past, especially to improve the performance of superscalar processors. This technique consists in predicting the input or output operands of an instruction and it is based on the observation that such input/output operands tend to repeat or follow a pattern. Mechanisms for value prediction try to capture that pattern based on tables that store information reflecting the history that has been observed in the recent past.

First proposal on value prediction was the last value predictor, presented by Lipasti, Wilkerson and Shen[37][38]. This simple predictor assumes that the next value an instruction will produce is the same value as in its previous execution.

Afterwards, more complex value predictors have been proposed such as the stride predictor[18][64] and the finite context machine (FCM) context-based[65] value predictor among others, and some combinations of them to obtain hybrid value predictors[87]. The stride predictor speculates that the new value seen by an instruction is the sum of the last value and a stride, which is the difference between two consecutive values. The predicted stride is replaced when a new stride has been seen twice in a row. On the other hand, the FCM predictor considers that values follow a repetitive sequence, and thus, estimates the new value based on the sequence of previous values of the same instruction operand.

A similar approach for FCM context-based value predictor was proposed by Goeman et *al.* in the Differential FCM context-based[19] value predictor. In this predictor, instead of storing the sequence of values produced by an instruction, it is stored the stride among them in order to detect sequences of strides.

Some other value predictors make their prediction using control-flow information by correlating the values with the direction of branches[51].

3.4.2. Value Prediction for Speculative Multithreaded Processors

This subsection is devoted to analyze the behavior of value prediction in the context of a speculative multithreaded processors. As it was shown, the performance of this type of architecture strongly depends on the ability to predict the input or output values of speculative threads.

First of all, it is necessary to determine which values are needed to be predicted[4]. In this way, a thread input value is defined as a value (in a register or memory location) that is consumed by the specula-

tive thread (the thread uses it without having been computed by the same thread). In the same manner, a thread output value is defined as a value (in a register or memory location) that is computed by the thread. Thus, the analysis of value prediction focuses on thread input or output values, since these are the values that flow through inter-thread dependences.

Depending on how value predictors perform the prediction of the thread input or output values, they can be classified into those that exploit correlation with past values of the same instruction operand and those that exploit correlation with values of the same trace. The former are called *instruction-based value predictors* while the latter are called *trace-based value predictors*.

Instruction-based value predictors have in common that their history tables store information about the values seen by individual instruction operands. Well-known predictors are the last value (LV), stride (STR), context-based (FCM) and hybrid schemes such as the stride-context (HYB-S) predictor. These value predictors have been thoroughly studied in the context of superscalar and VLIW processors.

The performance of instruction-based value predictors can be improved if information about the thread to which the instruction operand to be predicted belongs is also included in the history tables. This gives way to the so-called trace-based value predictors. Threads being considered in this paper are loop iterations that the thread speculation unit of the speculative multithreaded processor delimits for speculation. Obviously, this kind of predictors can be used for any other type of speculative threads.

Let us see a common case where instruction-based value predictors fail. The instruction-based stride predictor behaves badly when consecutive loop traces follow different paths of the same loop. Figure 3.6 shows an example where threads T_A, T_A, T_B and T_A are consecutively executed and Ri is the destination operand produced by instructions at PC_A and PC_B. Consider also that these instructions compute an output thread value. The stride predictor would speculatively compute the value of Ri in the last thread (Ri_4) as the last value produced by the same instruction (Ri_2) plus the stride computed for that instruction operand. This may give an incorrect value since thread T_B modifies the value of Ri.

For instruction-based predictors, the history tables are indexed through the instruction address of the operand to predict. If a source operand is to be predicted, a bit is appended to the instruction address in order to identify each source operand. A destination operand is directly identified by its instruction address. We refer to this indexing scheme as *PC-based indexing*.

Trace-based value predictors access the history tables through a thread identifier and a operand identifier (e.g. register identifier). For threads, we have considered a pseudo-identifier that consists of the instruc-

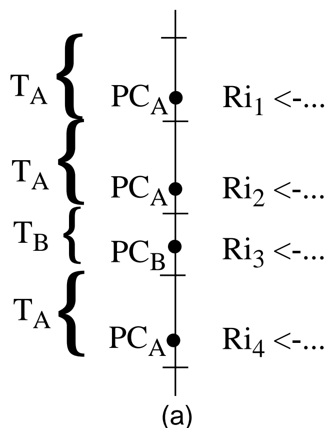
tion pointer of the spawning point along with a bit vector with the result of all conditional branches in the thread. It is not a unique identifier because a thread can have indirect unconditional branches, and their branch target addresses are not considered. We refer to this indexing scheme as *trace-based indexing*.

Note that trace-based predictors could alternatively index the history tables based on the instruction address of the producer (resp. first consumer) of the output (resp. input) value. Both types of indexing, PC- and trace-based indexing, are considered for trace-based predictors.

Finally, note that the instruction-based predictors presented in the previous section can be extended to correlate their predictions with previous instances of the same instruction operand in the same trace. This extension to convert them into trace-based predictors only requires minor modifications in the implementation, namely, the indexing function in the history table should consider both the instruction address and the trace identifier.

3.4.3. The Increment Predictor

A stride predictor computes a difference between two consecutive values of an operand at the same instruction address. Writes to the same storage location produced between these two instructions affect the accuracy of the predictor. Instead, it may be 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 loop iteration.



$Ri_4 = Ri_2 + STRIDE(PC_A)$
(b)

$Ri_4 = Ri_3 + INCREMENT(T_A, Ri)$
(c)

Figure 3.6. Stride and increment predictors: a) 4 consecutive traces (T_A, T_A, T_B and T_A) which reference register Ri; b) predicted value of Ri_4 using a stride predictor; c) predicted value of Ri_4 using the increment predictor.

The increment predictor predicts every thread output value as the value of that storage location at the beginning of the thread plus an increment. This increment is computed as the value at the end of the thread minus the value at the beginning of the thread in previous executions of the same trace. The predicted increment is updated when a new increment has been seen twice in a row.

Regarding figure 3.6, note that the value of Ri_3, which is an output value of thread T_B is also the value of register Ri at the beginning of the fourth thread (T_A). In this way, the value Ri_4 is predicted as Ri_3 plus the increment observed for this register in thread T_A in the past. Ri_3 may in turn contain a predicted value, which was computed as Ri_2 plus the increment observed for this register in thread T_B in the past. This scheme may be more accurate than an instruction-based predictor, since different traces are considered to update operands in a different way.

Besides, a hybrid scheme composed of the proposed 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.

3.4.4. Prediction Accuracy

This subsection analyzes the accuracy of the different value predictors for the two indexing functions and different table capacities. The differences in predictability of inputs and outputs are also investigated. The objective is to devise the configurations with most potential, whose impact on IPC will be later analyzed in subsection 3.4.6.

In this section we use a trace-driven simulation of the SecInt95 benchmark suite. The programs were compiled with the Compaq/Alpha compiler for an AlphaStation 600 5/266 with full optimization (-O4), and instrumented by means of the Atom tool. Programs used the reference input data during 300 millions of instructions after skipping the initializations.

Regarding the speculative threads that are being considered in this analysis, they have an average size of 36 instructions. Moreover, instructions belonging to innermost loop iterations represent almost the 62% of total instructions. Individual data for every program is depicted in Table 4.1.

3.4.4.1. Predicting Register Values through PC-Based Indexing

A proper selection of the values to be predicted may have an important impact on the performance of data value speculation techniques. We first compare the difference in predictability between trace input and out-

	instr in loop iterations	instr / loop iteration
go	44.30 %	40.57
m88ksim	83.00 %	54.26
gcc	54.36 %	32.09
compress	74.99 %	16.19
li	38.82 %	25.87
ijpeg	81.93 %	43.38
perl	52.19 %	50.90
vortex	75.75 %	247.33
AVERAGE	**61.73 %**	**36.13**

Table 4.1. Loop iteration statistics.

put values. Note that predicting the outputs of previous loop traces is another way to obtain the input values of a loop trace. Value predictors being analyzed here use a PC-based indexing mechanism.

In all figures presented for FCM predictors (included the hybrid version) we have considered that the VHT contains the last 3 values and these values are 0-bit, 2-bit and 4-bit shifted, respectively, before xoring them in order to obtain the index to the VPT. We also assume that the number of entries in each table is the same.

Figure 3.7.a shows the prediction accuracy for trace input register values whereas trace output register values are analyzed in Figure 3.7.b. The impact of the capacity of the history tables on the prediction accuracy is depicted along the X-axis. The INCR and HYB-I predictors are not depicted for input values since they only predict trace output values.

As observed for superscalar processors, a FCM can achieve a high prediction accuracy but it requires very large history tables. LV and STR predictors can achieve a better accuracy for small-sized history tables. For large tables, the LV predictor is the least accurate. A remarkable result is that input values are more predictable than output values (70% of inputs for a 64-KB table using a STR predictor and 60% of outputs using a HYB-I predictor with the same capacity). Another important result is that a STR predictor outperforms an INCR predictor by around a 10%. The difference in performance of the respective hybrid predictors is not so high; in fact, HYB-I has a slightly advantage over HYB-S which suggests that the type of patterns predicted by the STR and the FCM have more overlap than those predicted by the INCR and the FCM.

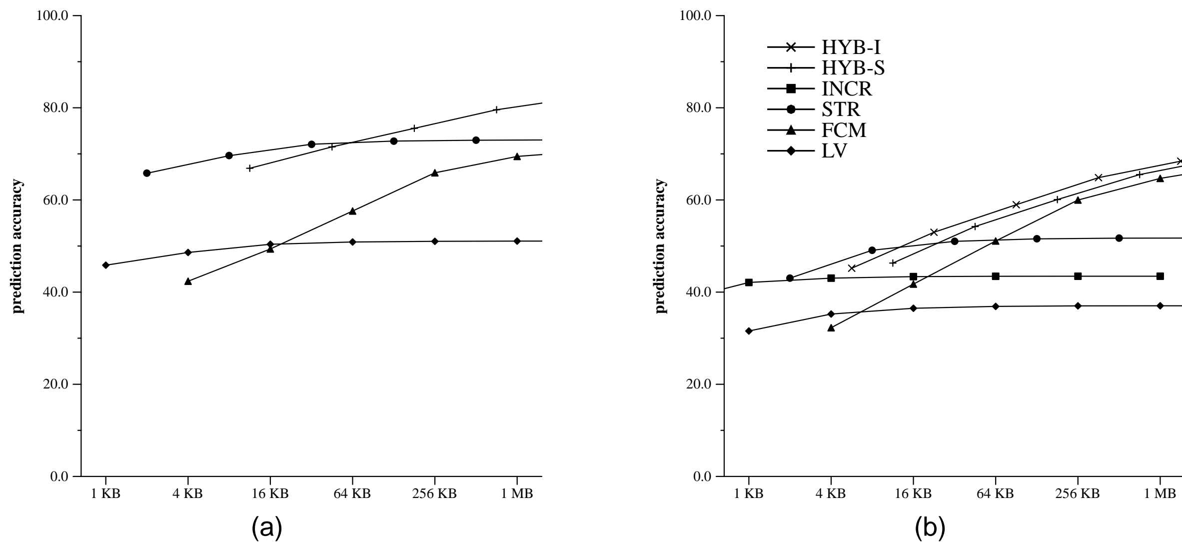


Figure 3.7. Predicting register values of loop traces using PC-indexed predictors: a) trace input values; b) trace output values.

Nonetheless, among all the inputs or outputs of a trace, 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. To estimate this effect, we compute the prediction accuracy for those input values produced by any of the previous 3 threads that run concurrently with it and those output values consumed by any of the following 3 threads. We refer to these values as distance-3 inputs and outputs respectively.

For distance-3 inputs (see Figure 3.8.a), the prediction hit rate diminishes when compared with that of predicting all values (it goes from 70% for a 64-KB HYB-S predictor as it can be seen in figure 3.7.a, to 60%). However, for distance-3 output values (see Figure 3.8.b), this trend is reversed. A 64-KB HYB-I predictor increases the prediction accuracy by 20% when compared with its accuracy for all outputs.

This is due in part to the fact that threads have in average more register outputs than inputs (8.2 versus 5.0), as shown in Figure 3.9. However, the average number of distance-3 register outputs is lower than distance-3 register inputs. The figure also includes statistics for memory values, showing that the average number of distance-3 memory inputs and outputs is rather low. The difference between the number of input and outputs resides in that an output can be used by more than one speculative thread. That is, if the thread input values are individually predicted, it is possible that a single value is to be predicted more than once,

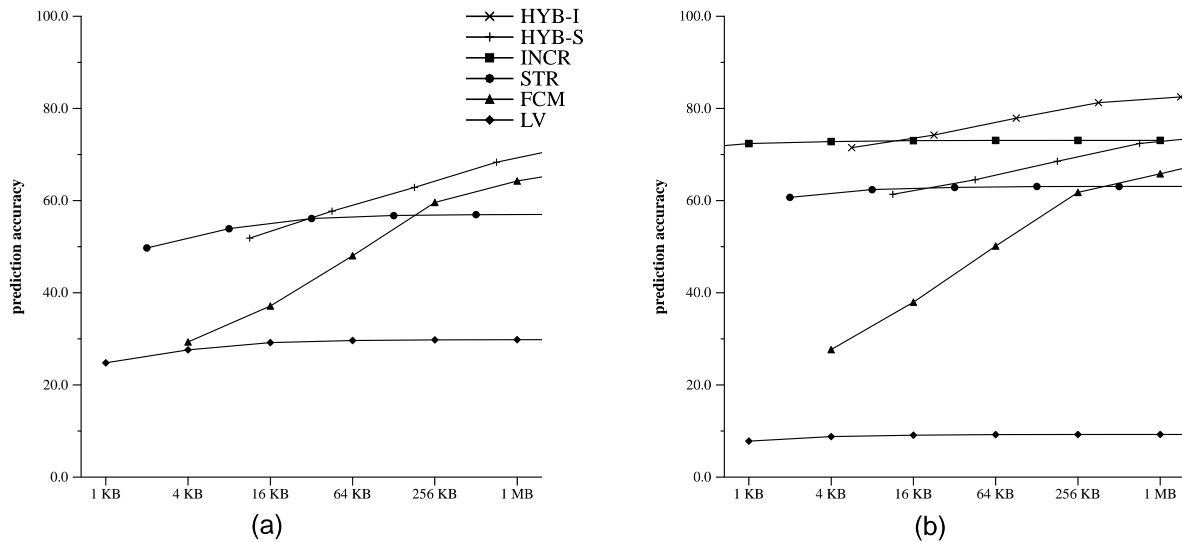


Figure 3.8. Predicting distance-3 values of loop traces using PC-indexed predictors: a) input values; b) output values.

one for each speculative thread that uses the same values. On the other hand, predicting the outputs only requires to predict the value once.

Another remarkable fact is that the INCR predictor outperforms the STR predictor by about 10% 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, as discussed in section 3.4.2, whereas these interferences are avoided by a trace-based predictor such as the INCR, even if the indexing function uses only the instruction address.

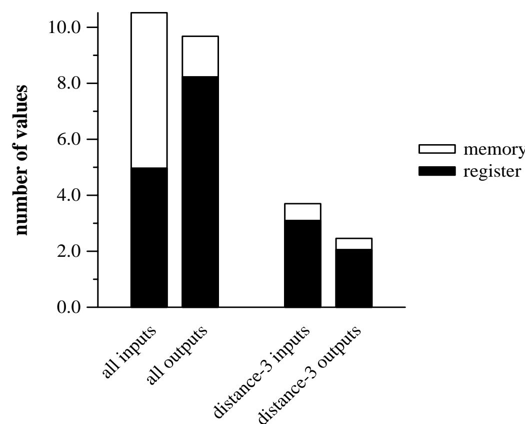


Figure 3.9. Average number of inputs/outputs and distance-3 inputs/outputs per trace.

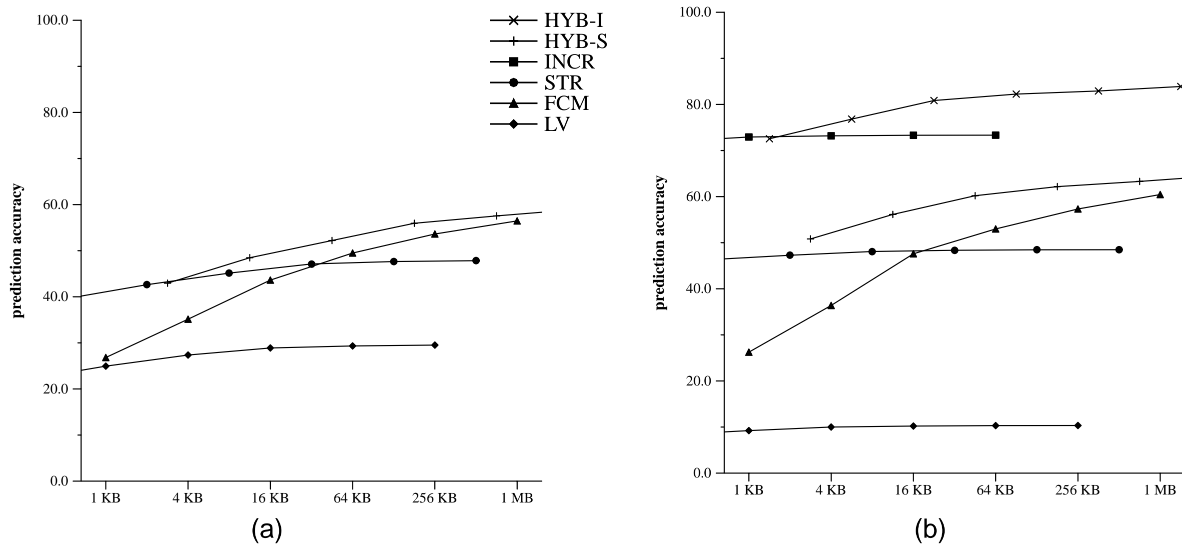


Figure 3.10. Predicting distance-3 values of loop iterations using trace-based indexing: a) input values; b) output values.

As conclusions up to this point, for a speculative multithreaded architecture based on loop traces, the most predictable values are trace outputs. Moreover, the INCR 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% for a 1 KB table).

3.4.4.2. Predicting Register Values through Trace-Based Indexing

For trace-based predictors, the thread identifier can be included in the indexing function. Figure 3.10 shows the prediction accuracy for distance-3 input/output vales. It can be observed that the hit rate for input values decreases when compared with PC-based indexing (60% for a 64-KB PC-indexed HYB-S versus 50% for the trace-indexed version of the same predictor). However, the performance for output values increases. Although the INCR predictor obtains a similar performance (73% hit rate for the whole range of table capacity), the HYB-I predictor can achieve a 80% hit ratio with relatively small tables (16 KB in total). This is mainly caused by the significant performance boost of trace-based indexing for the FCM predictor, which is due to the benefits of using different value sequences for different traces.

As conclusions of this analysis on prediction accuracy, the four selected predictors for a further analysis are those with the highest prediction accuracy for a moderate-sized history table (16 KB): HYB-I, INCR, HYB-S and FCM, with a trace-based indexing function.

Figure 3.11 shows the percentage of speculative threads whose all distance-3 outputs are correctly predicted. This gives an estimation of the percentage of traces 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% for 1-KB INCR predictor). With large history tables, this percentage can be as much as 70% when a HYB-I predictor is used.

3.4.5. Microarchitectural Issues on Value Prediction

In previous subsection, the ability to predict the thread input/output values of different value predictors have been shown. In this section how to implement this value predictors in a speculative multithreaded processor is described.

This implementation is related to the time when the predictions are generated and speculative values are verified. Regarding the prediction, when a new speculative thread is created, the value predictor has to supply the predicted values for the thread input register/memory locations to the new spawned speculative thread. This may cause a peak of predictions when the speculative thread is created.

Finally, regarding the verification of the prediction, this process has to be performed as soon as possible. Thus, recovery mechanisms will early proceed in case of misspeculations and also, the propagation of the mispredicted value to other speculative threads is avoided.

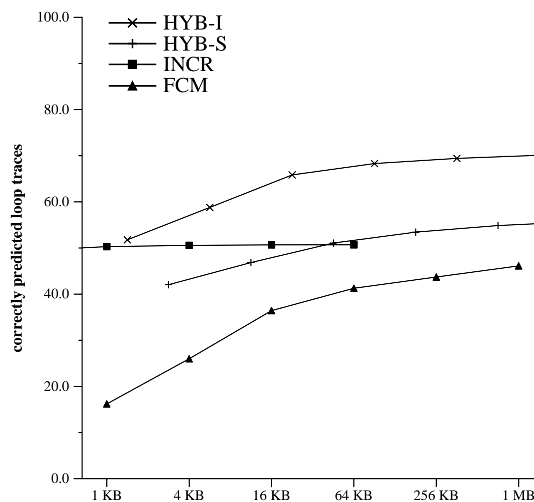


Figure 3.11. Percentage of traces that have all their distance-3 output values correctly predicted.

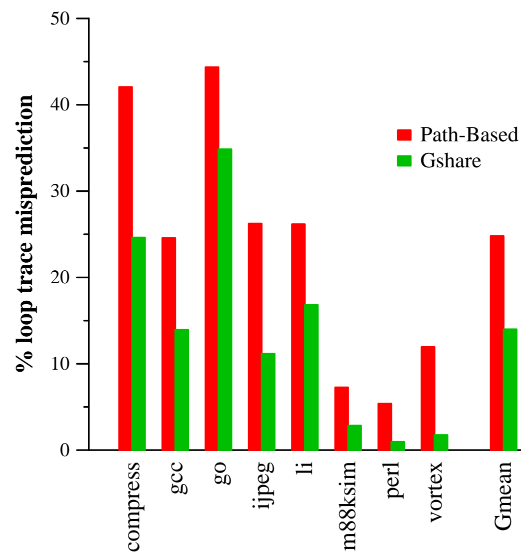


Figure 3.12. Control-flow misprediction for the Path-based and the ideal gshare for loop iterations.

3.4.5.1. Initializing Speculative Threads.

Speculative multithreaded processors behave like a superscalar processor until a spawning point is reached. At this point, a new speculative thread can be spawned. First of all, it is necessary to find an idle thread unit to allocate the new spawned thread. Then, input values are predicted.

It has been shown that there are some value predictors, such as the increment value predictor, which require to know the control flow taken by the speculative thread in order to do the prediction. Such prediction implies to anticipate the outcome of all the conditional branches between the spawning and the control quasi-independent point. To do that, a control-flow predictor based on the Path-based Next-Trace[29] predictor can be used. This predictor works similarly to a FCM context-based value predictor and it is has made up of two tables: a correlating table which contains the last *n* control-flows taken after such thread and a secondary table which holds the predicted next control flow.

Figure 3.12 shows the prediction miss ratio of a Path-Based predictor to predict the whole control-flow for loop iterations. The sizes of the tables considered are 1K-entry secondary table and 8K-entry correlating table which stores the last 3 paths. For comparison, an ideal gshare which is able to predict as many branches as a loop trace is also depicted in the graph. Such thread predictor is idealized in the sense that it is assumed to predict multiple branches for the whole control-flow of a thread. It can be observed that the control-flow misprediction for the Path-Based predictor is quite elevate. It is closer to 24% and quite ele-

vate for some benchmarks like `go` and `gcc` which reaches a 40% misprediction. Besides, its performance is quite far from the ideal gshare which only mispredicts 14% of threads.

Fortunately, most of the live-in values use the same stride or follow the same pattern independently of the control-flow taken by the thread. In this way, a low hit rate does not necessarily imply a significant drop in the value prediction accuracy. Experiments have shown that on average, the misprediction rate is just increased by 10% compared with a perfect next-thread predictor.

When a thread reaches a spawning point and there is an idle thread unit, the live-in values of the spawned thread have to be correctly initialized. If the value predictor requires to know the control flow associated to that thread, a prediction regarding its control flow is done. Such predictor is indexed by a hash function of the instruction pointers of the spawning and the control quasi-independents point and predicts the sequence of the conditional branch outcomes. With this prediction and the instruction pointer of the spawning and the control quasi-independent point, the value predictor is indexed. In the case study of loop iterations, the value prediction information can be stored in the loop iteration table. This table contains the history for each trace, which is used to predict output register values and memory dependences, among other things. If the predicted thread is not currently in the loop iteration table, the spawning of speculative threads is aborted. Otherwise, the corresponding entry of the loop iteration provides a bit vector that identifies the output registers of the trace. In other cases, the information regarding which register are input/output may be encoded in the spawn instruction.

In parallel to these actions, the register map table is copied from the previous thread. In clustered speculative multithreaded processors, the whole register file has also to be copied. Then, the values for those registers that are expected to be written by any previous thread (according to the predictions done for it) are predicted. That is, only those registers whose values are not available at thread creation are predicted.

In order to make those predictions, a series of instructions are inserted into the instruction window of the speculative thread. These instructions are responsible for computing the predicted data values. These instructions can be arithmetic operations such as adding the corresponding increment or stride to a register. If the processor uses a context-based or a last value predictor, such instructions can be "move" a given value to a given register. These instructions will be executed as normal instructions and they do not have more priority than the others. If the previous threads have already computed all their outputs, or they have been predicted, these instructions will immediately execute since their operands are ready.

3.4.5.2. Verifying the Predicted Values.

When a thread becomes the non-speculative one, all the predictions done have to be verified in order to know if the program is being executed correctly. These verifications consist of checking that the value of the predicted registers at the end of the committed thread match the predicted values. To do that, it is necessary that speculative threads store the predicted values in order to be compared with the real ones.

For this purpose, the inserted instructions for predicting the values also store the produced value on an especial table which is referred to as *Register Prediction Table*. Thus, to perform the verification, the value of the registers at the end of the non-speculative thread are compared to the values stored in this table.

The verification process can be done at the moment the speculative thread becomes non-speculative. This late verification might cause that the cost of mispredictions be very elevate since threads can take a long time to become non-speculative. Thus, such high misspeculation cost requires very accurate value predictors in order to obtain any benefit.

Another approach consists of performing the verification as soon as the values are produced. To know when a value is computed by the producer thread, the processor can use the last-write instruction as for synchronization mechanisms. In this way, when a last-write instruction is committed, the value is forwarded to the consumer thread and it is compared with that stored in the Register Prediction Table. If values are not the same, a misspeculation has occurred and a recovery mechanism has to start. This approach allows the processor to early detect misspeculations and reduce the cost of them. However, it may incur in false misspeculations, that is, a speculative thread may forward speculative values that are incorrect whereas in the first approach, the forwarded values are always correct. Different solutions for this phenomena can be considered such as allowing only the non-speculative thread to forward values for early validation but in fact, false misspeculations are quite uncommon.

Finally, regarding the recovery mechanisms, they can be a simple squash of all the following speculative threads or a selective reissuing of the dependent instructions.

3.4.6. Performance Figures

In this subsection, different register value predictors for a clustered speculative multithreaded processor are evaluated. To evaluate the performance of the processor, the configuration detailed in subsection 3.3.5 is used.

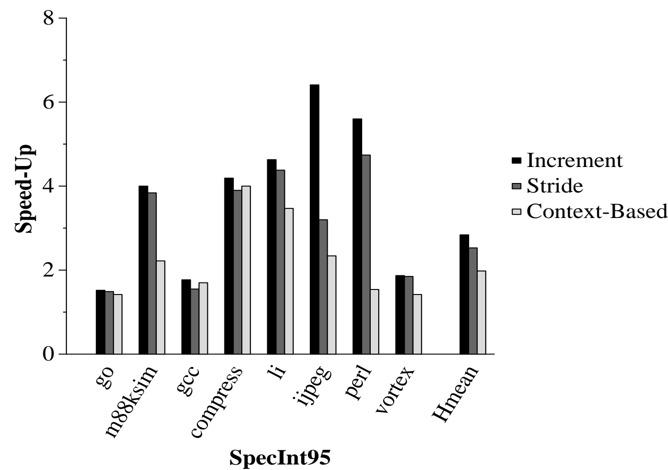


Figure 3.13. Speed-up for the different value predictors and for the loop-iteration spawning policy.

The size of the value predictor is fixed to 8KB and the misprediction penalty considered is the elapsed time until the correct value is available plus an extra cycle to forward the correct value plus 1 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, i.e. only dependent instructions of the mispredicted value have to be re-executed. Memory values are not predicted and dependent values are forwarded from the producer to the consumer with a delay of 3 cycles by means of a Speculative Versioning Cache.

Figure 3.13 shows the speed-up achieved by the clustered speculative multithreaded processor over a single-threaded execution for different register value predictors and the unrestricted spawning model. On average, the losses due to a realistic value predictor in comparison with perfect register value prediction are 16% 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.

3.5. CONCLUSIONS

In this chapter different mechanisms to deal with interthread data dependences are proposed and analyzed. The former approach consists of stalling the execution of dependent instructions of a speculative thread until the producer thread has computed the corresponding values and has forwarded them to the consumer. Different mechanisms based on hardware and software techniques are studied even though the results reported by a speculative multithreaded processor with perfect synchronization are quite poor.

Value prediction is presented as a mechanism to deal with interthread data dependences. The main feature of this mechanism is its ability to break the dependence chain between the speculative threads so that

both can be executed as if they were independent. The performance of different well-known value predictors is studied for speculative multithreaded architectures and a new value predictor targeted to this type of architectures, which is referred to as increment predictor, is presented. The increment predictor computes a new value for a thread output as its value at the beginning of the thread plus an increment that depends on the control-flow that thread will follow.

Experimental results have shown the importance of choosing the correct values to be predicted. Output trace values are more predictable than inputs. Moreover, trace-based indexing outperforms PC-based indexing. We have also shown that the increment predictor obtains the highest prediction accuracy with small-sized history tables. This accuracy is increased for larger history tables by means of a hybrid predictor that combines an increment predictor and a context-based predictor. Average hit ratio for SpecInt95 ranges from 73% to 84% depending on the capacity of the history table.

Overall, the main conclusion of this chapter is that value prediction plays an important role on this kind of architectures since it helps to break interthread data dependences. Different spawning schemes that take into account the predictability of the interthread data dependent values will be analyzed in the next Chapter.