Executing Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support [1]

Venkata Krishnan and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
venkat,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/iacoma/

Abstract

With the conventional superscalar approach of exploiting ILP from a single flow of control giving diminishing returns, integrating multiple processing units on a die seems to be a promising approach. However, in these architectures, the resources are partitioned such that a thread is allocated exclusively to a processor. This risks wasting resources when a thread stalls due to hazards. While simultaneous multithreading (SMT) addresses this problem with complete resource sharing, its centralized structure may impact the clock frequency. An intuitive solution is a hybrid of the two architectures, namely, a clustered SMT architecture, where the chip has several independent processing units, with each unit having the capability to perform simultaneous multithreading.

In this paper, we describe a software-hardware approach that enables speculative execution of a sequential binary on a clustered SMT architecture. The software support includes a compiler that can identify threads from sequential binaries. The hardware includes support for inter-thread register synchronization and memory disambiguation. We evaluate the resulting clustered SMT architecture and show that it is more cost effective than a centralized SMT and architectures where all the resources have a fixed assignment.

Keywords: superscalar, chip multiprocessors, simultaneous multithreading, speculation, binary annotation

1 Introduction

With wire delays dominating the cycle time of future microprocessors, alternatives to the conventional superscalar design are being actively investigated. One promising approach is to integrate multiple processing units on a single die [5]. This allows multiple control flows (also called threads) in the application to be executed concurrently. Unfortunately, by assigning a thread exclusively to a processor [5, 8, 10], we run the risk of wasting resources. Indeed, when a thread cannot execute due to a hazard, the resources of its processor are typically wasted. We term this type of architecture the *fixed assignment* (FA) architecture.

To utilize the resources better, we can design a more centralized architecture where threads can share all the resources. This is the approach taken by the basic Simultaneous Multithreading (SMT) [11]. In this scheme, the processor can support multiple threads such that, in a given cycle, instructions from different threads can be issued. If, in a given cycle, a thread is not using a resource, that resource can typically be utilized by another thread. Evaluation of this architecture running multiprogrammed and parallel workloads [3, 4, 11] has shown significant speedups. However, a drawback of this approach is that it may inherit the complexity of existing superscalars. In fact, with the delays in the register bypass network dictating the cycle time of future high-issue processors [7], a fully centralized SMT processor is likely to have a slower clock.

An intuitive alternative is a hybrid of the FA and the centralized SMT approaches, namely a *clustered SMT architecture.* In this architecture, the chip has several independent processing units, with each unit having the capability to perform simultaneous multithreading. Given that the effort to enhance a conventional dynamic superscalar to perform simultaneous multithreading is small [11], a low-issue SMT processor is feasible. Also, a clustered SMT architecture is a simple design as it involves replicating several low-issue SMT processors on a die. However, it might be argued that this separation of processing units would restrict the sharing of resources. But, as we will show in this paper, this architecture is able to extract most of the benefits that can be achieved with the fully centralized SMT approach at a lower cost.

Clearly, for the FA, centralized SMT, and clustered SMT architectures to deliver good performance on single applications, it is important that the compiler identify parallel threads. Since identifying fully independent threads statically is quite difficult, a good approach is for the compiler to identify threads in a speculative fashion at compile time [1, 10] and for the hardware to assist at run-time to honor dependences [6, 8]. It has been shown that this speculative approach has good potential [9].

A large number of applications are available only in their binary form. As a result, this approach needs to be used at the binary level too. Consequently, in this paper, we describe an automatic annotation process that can identify points of thread creation as well as register dependences across threads from a sequential binary. We use this scheme to generate code for our clustered SMT architecture. We also describe the hardware support that is needed to honor inter-thread register and memory dependences.

Overall, in this paper, we present and evaluate a hardware-software approach that allows the execution of sequential binaries with high performance. The paper is organized as follows: Section 2 describes our architecture and software approach in detail; Section 3 describes the evaluation environment; and Section 4 performs the evaluation.

2 Hardware and Software Support for Speculative Execution

2.1 Software Support: Binary Annotator

This section describes the software support for generating threads from a sequential binary to run on the SMT and FA architectures. We have developed a binary annotator that

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from IBM and Intel.

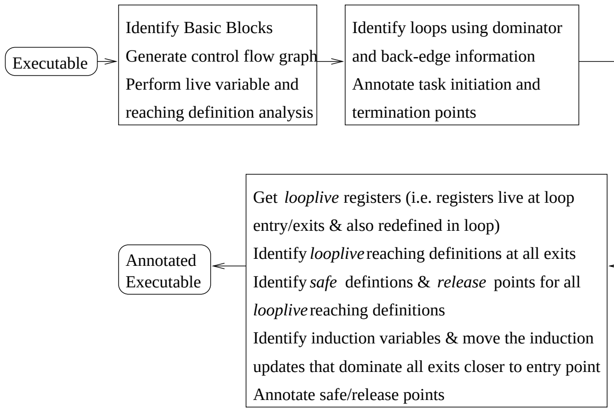


Figure 1: Binary annotation process.

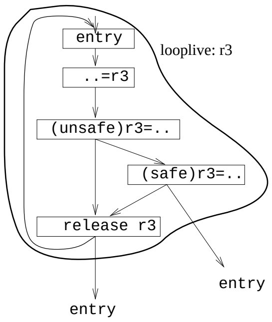


Figure 2: Safe definitions and release points.

identifies units of work for each thread and the register level dependences between these threads. Memory dependences across threads are handled by the additional hardware described in Section 2.2. Currently, we use loop iterations for our threads and mark the loop entry and exit points. During the course of execution, when a loop entry point is reached, multiple threads are spawned to begin execution of successive iterations speculatively. However, we follow sequential semantics for thread termination.

Threads can be squashed. For example, when the last iteration of a loop completes, any iterations that were speculatively spawned after the last one are squashed. Also, threads can be restarted. For example, when a succeeding iteration loads a memory location before a predecessor stores to the same location, all iterations starting from the iteration that loaded the wrong value are squashed and restarted.

The steps involved in the annotation process are illustrated in Figure 1. The approach we use is similar to that of Multiscalar [8] except that we operate on the binary instead of the source program. First, we identify inner loop iterations and annotate their initiation and termination points. Then, we need to identify the register level dependences between these threads. This involves identifying *looplive* registers, those that are live at loop entry/exits and *may* also be redefined in the loop. We then identify the reaching definitions at loop exits of all the *looplive* registers. From these *looplive* reaching definitions, we identify ***safe definitions***, which are definitions that *may* occur but whose value would never be overwritten later in the loop body. Similarly, we identify the *release* points for the remaining definitions whose value may be overwritten by another definition. Figure 2 illustrates the ***safe definition*** and *release* points for a *looplive* register $r3$. These points are identified with a backward reaching definition analysis, followed by a depth-first search, starting at the loop entry point, for every *looplive* reaching definition. Finally, induction variables are identified and their updates are percolated closer to the thread entry point provided the updating instructions dominate the exit points of the loop. This reduces the waiting time for the succeeding iteration before it can use the induction variable.

2.2 Hardware Support For Speculative Execution

Executing threads speculatively requires special support to handle two effects: inter-thread register synchronization and memory disambiguation. These two supports are necessary for both the SMT and the FA architectures that we discuss in Section 3.2. The first support is necessary because, with multiple threads working concurrently, the logical register state of the application is distributed across the register files of all threads. To enable a thread to acquire the correct register value during its execution, we propose a special scoreboard that we call the ***Synchronizing Scoreboard*** (Section 2.2.1). To support memory disambiguation, we use the ***Memory Disambiguation Table***, whose functionality is similar to the ARB [2], together with private write buffers for each thread (Section 2.2.2).

For the hardware to work, each thread maintains its status in the form of a bit-mask (called $ThreadMask$) in a special register. The status of a thread can be any of the four values shown in Table 1. Inside a loop, the non-speculative thread is the one that is executing the current iteration. Speculative successors 1, 2, and 3 are the ones executing the first, second, and third speculative iteration respectively. Clearly, as execution progresses and threads complete, the non-speculative $ThreadMask$ will move from one thread to its immediate successor and so on.

Thread Status	$ThreadMask$
Non-Speculative	0001
Speculative Successor 1	0011
Speculative Successor 2	0111
Speculative Successor 3	1111

Table 1: Status masks maintained by each thread.

2.2.1 Synchronizing Scoreboard

Register	Busy $B_0 B_1 B_2 B_3$	Valid $V_0 V_1 V_2 V_3$	Sync $S_0 S_1 S_2 S_3$
0			
1	0 0 0 0	1 1 1 1	0 0 0 0
...			
15	1 0 0 0	1 1 1 0	0 1 0 0
16	1 0 1 0	0 0 1 1	0 1 0 0
...			

Table 2: Synchronizing scoreboard.

The synchronizing scoreboard (SS) handles inter-thread register dependences. Table 2 shows an example SS for a processor with four threads. For each register and thread, the SS keeps three bits, namely *Busy (B)*, *Valid (V)* and *Sync (S)*, The sub-indices in the table denote the thread that owns the bit.

The B bit is as in a conventional scoreboard. It is set while the register value is being created by the thread. The S bit means that the register is not yet available to successor threads. When a thread starts, it sets the S bit for all the *looplive* registers that it may create. The S bit for a register is cleared when a safe definition for that register or the release instruction for that register is executed (Section 2.1).

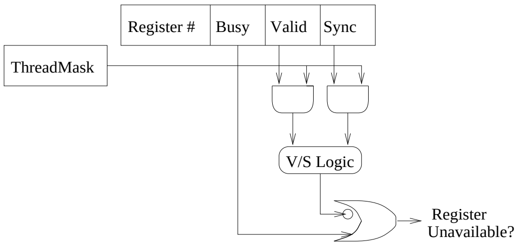


Figure 3: Logic to check register availability. The V/S logic implements the function in equations like (1) and (2).

At that point, the register is safe to use by successor threads. Finally, the V bit tells whether the thread has a valid copy of the register.

The SS works as follows. When a given thread needs a register, the SS is checked to determine if the register is available. This is done by first checking the B bit for the register/thread combination. If the B bit is set, the register is not available. This is identical to conventional scoreboarding. If B is clear, however, we check whether we can get the register from the thread or from predecessor threads. We do it as follows. Suppose that thread 0 is the non-speculative thread, that threads 1, 2, and 3 are busy executing speculative threads, and that the access came from thread 3. In that case, the register will be available to thread 3 if:

$$V_3 + \overline{S}_2(V_2 + \overline{S}_1(V_1 + \overline{S}_0)) \qquad (1)$$

This means that a register for thread 3 is available only if its own copy is valid or the copy in the immediate predecessor is valid with no synchronization required, and so on. This incurs a delay of an AND-OR logic, with the number of inputs to the gate dependent on the number of supported threads.

Suppose, instead, that thread 1 is the non-speculative thread and threads 2, 3, and 0 are busy executing speculative threads, and the access came from thread 0. In that case, the register will be available to thread 0 if:

$$V_0 + \overline{S}_3(V_3 + \overline{S}_2(V_2 + \overline{S}_1)) \qquad (2)$$

To generate all these bit operations for all possible threads without having to perform shifts, we can replicate each of the V and S columns in the SS. We interleave them as follows: $V_1V_2V_3V_0V_1V_2V_3$ for the V bits and in a similar way for the S bits. Now we can perform any bit operation using always four consecutive columns. For instance, for the example (1) above, we use the columns with the lower case letters in $V_1V_2V_3v_0v_1v_2v_3$, while for the example (2) above we use the columns with the lower case letters in $v_1v_2v_3v_0V_1V_2V_3$.

What determines the columns used is the thread's *ThreadMask* of Table 1. In examples (1) and (2), the SS access has been performed by speculative successor 3. Therefore, according to Table 1, we have used mask 1111, thereby enabling all four columns to the left of speculative successor 3 and computing the whole expression (1) or (2). Consider a scenario like in (2), where thread 1 is non-speculative, except that the access is performed by thread 3. In that example, thread 3 is speculative successor thread 2. Consequently, we would use *ThreadMask* 0111 from Table 1. This means that we are examining only 2 predecessors of thread 3. The columns enabled are shown in lower case in $V_1V_2V_3V_0v_1v_2v_3$. The resulting formula will be:

$$V_3 + \overline{S}_2(V_2 + \overline{S}_1)$$

Overall, the complete logic involved in determining whether a register is available or not is shown in Figure 3. If the register is not available, the reader thread stalls. Otherwise, the reader thread gets the value from the closest predecessor thread with the V bit set (itself included). The cost of this register copy depends on whether or not the producer and consumer threads are in the same cluster. If they are not, the transfer takes one cycle more. Overall, the lookup mechanism and port requirements of the synchronizing scoreboard have a hardware complexity that is similar to that of a register re-mapping table in a dynamic superscalar processor.

2.2.2 Memory Disambiguation

We use a fully associative Memory Disambiguation Table (MDT) to identify speculative loads and stores that violate sequential semantics. In addition, each thread also has a private write buffer to hold its stores. Stores performed by speculative threads are held in the buffer until they reach non-speculative status, after which they can be issued to memory. This prevents corruption of memory locations in case of incorrect speculations.

Each entry in the MDT has a word address, as well as *Load (L)* and *Store (S)* bits on a thread basis. The load and store bits are maintained only for speculative threads. Consequently, when a thread acquires non-speculative status, all its L and S bits are cleared. Table 3 shows the MDT for a processor with 4 threads. We now explain how the MDT works in a load and store operation and the handling of MDT overflows.

Valid	Word Address	Load Bits $L_0L_1L_2L_3$	Store Bits $S_0S_1S_2S_3$
1	0x1234	0 0 1 0	0 1 0 0
0	0x0000		
...			
1	0x4321	0 0 1 1	0 1 0 0

Table 3: Memory disambiguation table.

Load Operation

A non-speculative thread need not perform any special action and its loads proceed normally. A speculative thread, however, may perform unsafe loads and has to access the table to find an entry that matches the load's address. If an entry is not found, a new one is created and the L bit corresponding to the thread is set. However, if an entry is already present, it is possible that a previous thread could have speculatively updated the location. Therefore, the store bits are checked to determine the id of the thread that has updated the location. As in the SS scheme, of the full set of columns $S_1S_2S_3S_0S_1S_2S_3$, depending on the ID of the thread issuing the load, a contiguous 4-bit value is selected. In addition, the *ThreadMask* is used to possibly mask out some of these bits depending on the speculative status of the thread issuing the load. For example, if thread 2 is the 2nd speculative successor and it issues the load, then only bits $S_0S_1S_2$ are examined.

The closest predecessor (including the thread itself) whose S bit is set gives the id of the thread whose write buffer has to be read. In the above table, when thread 2 reads location 0x1234, it acquires the value from thread 1's write buffer. If no S bits of any predecessor threads are set, the load proceeds in the conventional manner. This, of course, means checking the write buffer of the non-speculative thread before attempting to load from memory. If, however, a thread loads the value from its own write buffer, we do not set its L bit. This eliminates false dependences by allowing a thread to store to a location and then load from the same location later. Thus, only unsafe load operations are marked in the MDT.

Store Operation

Unlike loads, stores can result in the squashing of threads when a speculative thread prematurely loads a value for an address before a predecessor thread has completed the store. Successor, therefore, rather than predecessor threads are tested on a store operation. As in the load operation, we check for an entry corresponding to the address being updated. If the entry is not found, speculative threads create a new entry with their S bit set. Non-speculative threads, instead, perform a plain store operation with their updates going to the write buffer and eventually to memory.

If an entry is present, however, a check must be made for incorrect loads that may have been performed by one of the thread's successors. Both the L and S bits must be accessed. We again use the $ThreadMask$ bits to select the right bits. However, we now use the *complement* of these bits as a mask. This is because the complement gives us the *successor* threads. For example, assume that thread 0 is the non-speculative thread and is performing a store. The complement of its $ThreadMask$ from Table 1 is 1110. It means that it is checking the bits of its three successors, the lower cases in $l_1 l_2 l_3 L_0 L_1 L_2 L_3$ and $s_1 s_2 s_3 S_0 S_1 S_2 S_3$. The *check-on-store* logic is:

$$L_1 + \overline{S}_1(L_2 + \overline{S}_2 L_3)$$

This logic determines whether any successor has performed an unsafe load without any intervening thread performing a store. Clearly, if an intervening thread has performed a store, there would be a false (output) dependence that must be ignored. For example, the last row of Table 3 shows that thread 1 wrote to location 0x4321 and then threads 2 and 3 read from it. If non-speculative thread 0 now writes to 0x4321, there is no action to be taken because there is only an output dependence on the location between threads 0 and 1. If the check-on-store logic evaluates to true, however, the closest successor whose L bit is set and all its successor threads are squashed and restarted. Finally, if the store is performed by a speculative thread, the S bit is set.

Table overflow

It is possible that the MDT may run out of entries. The thread needing to insert a new entry must necessarily be a speculative thread. If no entries are available, the thread stalls until one becomes available. An entry becomes available when all its L and S bits are zero. Recall that when a thread acquires non-speculative status, it clears its L and S bits, as they are safe from this point onwards. It is at this point that a MDT entry may become available. Nevertheless, the size of the MDT need not be large for optimal performance. This is discussed in Section 4.4.

3 Evaluation Environment

In this section, we give details on different architectures evaluated as well as our evaluation environment.

3.1 Conventional Dynamic Superscalar Architecture

We model a very aggressive dynamic superscalar processor for comparing performance with the SMT architecture. This dynamic processor can fetch and retire up to 8 instructions each cycle. It has an instruction window of 128 entries along with 160 int- and 160 fp- registers and can perform multiple branch predictions even where there are pending unresolved branches. All instructions take 1 cycle to complete. The only exceptions are integer and floating-point multiply and divide operations: integer multiply and divide takes 2 and 8 cycles respectively; floating-point multiply takes 2 cycles while divide takes 4 cycles (single precision) and 7 cycles (double precision).

Caches are non-blocking with up to 32 outstanding loads allowed with full load bypassing enabled. We assume the L1 and L2 cache size (latency) to be 64K (1-cycle) and 1M (6-cycles) respectively with misses to the L2 cache taking an additional 26 cycles. We assume a perfect I-cache for all our experiments and model only the D-cache.

3.2 Centralized, Clustered SMT and Fixed Assignment Architectures

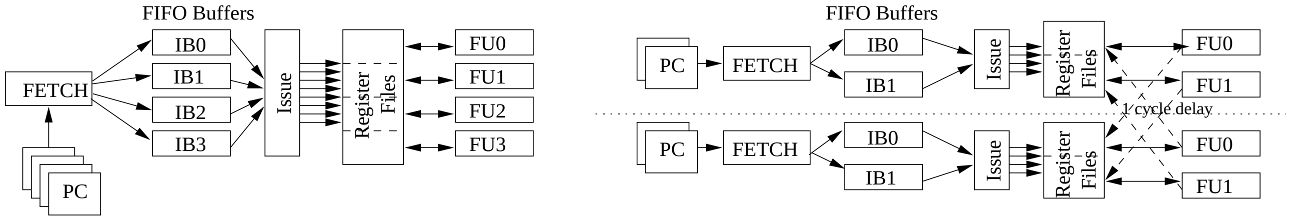
We model both the centralized and the clustered SMT processor architecture. This is shown in figures 4-(a) and (b). The centralized architecture can issue up to 8 instructions per cycle to a set of 8 integer, 4 floating-point and 4 load/store units. It has, for each thread, a 16-deep FIFO instruction buffer and 32 int/fp registers. The fetch unit fetches 8 instructions from a different thread every cycle in a round-robin fashion. We can issue up to 8 ready instructions from the different instruction buffers of the threads. Only the first 8 instructions at the head of each FIFO buffer are considered for issue. If any instruction is not ready, the ones behind it are ignored. Thus, we strictly issue instructions in-order for each thread. Note that for 4 threads, up to 32 instructions may have to considered. So the complexity of this phase is less than that of the *wakeup and selection* delay for a 32-instruction associative window in a dynamic superscalar processor. Thus, this doesn't affect the cycle time of the centralized architecture as much as the bypass delay [7].

To alleviate the bypass problem, we partition the SMT processor into 2 clusters of 4-issue SMT processing units. Each processing unit supports 2 threads and has its own fetch unit that can fetch 4 instructions/cycle in a round robin fashion. Each thread has a 8-entry FIFO buffer from which up to 4 instructions may be issued to a set of 4 integer, 2 floating-point and 2 load/store units. We assume an additional delay of 1-cycle for bypassing across clusters. Unlike the centralized scheme, no resource sharing is done across clusters. So, if there is only one thread in the program, this processor can achieve at most 4 IPC.

Finally, for the fixed assignment (FA) architectures, we consider 2 variations: FA_2 and FA_4. FA_2 is a chip with 2 4-issue processors, while FA_4 is one with 4 2-issue processors. The description for all the architectures is given in Table 4.

Note that we have modeled the SMT and the FA architectures to be in-order issue rather than dynamic issue. This is to show that, in spite of this restriction, the SMT and FA architectures can still give high performance. Thus, the resulting area savings could be used for a larger cache. However, for simplicity of the analysis, we keep the caches constant across architectures.

Finally, we consider the clock cycle time of the different architectures. The large bypass network determines the cycle time for the centralized SMT architecture, while the synchronizing scoreboard determines the cycle time for the clustered SMT and FA architectures. In terms of delay, the synchronizing scoreboard can be thought of as a renaming table of 32 entries. Using the model provided by [7] for an 8-issue processor with 0.18μ technology, the estimated delay for the synchronizing scoreboard would be around $500ps$, while the estimated delay for the bypass network is over $1000ps$. These values indicate that the cycle time of the centralized SMT will be much larger than the cycle time of the clustered SMT and FA architectures.



(a) (b)

Figure 4: SMT architectures: (a) centralized and (b) clustered.

Processor Type	Number of Processors	Threads per Processor [Chip]	Max. IPC per Processor [Chip]	Number of FUs per Processor [Chip] (int/ld-st/fp)
Conventional Superscalar	1	1 [1]	8 [8]	8/4/4 [8/4/4]
Centralized SMT	1	4 [4]	8 [8]	8/4/4 [8/4/4]
Clustered SMT	2	2 [4]	4 [8]	4/2/2 [8/4/4]
FA_2	2	1 [2]	4 [8]	4/2/2 [8/4/4]
FA_4	4	1 [4]	2 [8]	2/1/1 [8/4/4]

Table 4: Description of the different types of architectures evaluated.

3.3 Simulation Approach

Our simulation environment is built upon the MINT [12] execution-driven simulation environment. We use binaries generated on the SGI PowerChallenge for 3 integer benchmarks (*compress*, *ijpeg* and *eqntott*) and 4 floating point benchmarks (*swim*, *tomcatv*, *hydro2d* and *su2cor*). All are from the SPEC95 suite except *eqntott*, which is from SPEC92. We use the train set as input for the SPEC95 applications and the ref input for *eqntott*. Note that for the 8-issue conventional dynamic superscalar processor, the native SGI compiler cannot generate an optimally compiled binary. However, the 128-entry instruction window in the simulated processor should offset this to a large extent. In addition, we also use a rescheduler on the execution trace before feeding it to the simulator. This gathers several loop iterations thereby enabling aggressive loop-unrolling and performs a resource constrained scheduling of instructions. It is this optimized window of instructions that is finally passed to the processor simulator.

4 Evaluation

4.1 Dynamic Superscalar vs Centralized SMT Processor

Both the aggressive dynamic superscalar (Section 3.1) and the centralized SMT processor (Section 3.2) are 8-issue processors. Therefore, the complexity of their bypass networks is likely to be similar for a given implementation. Since we follow [7] in assuming that the delay in the bypass network is the most dominant one, both processors should have approximately the same clock cycle time.

Figure 5 shows that, for the integer applications, *compress*, *ijpeg* and *eqntott*, both processors have similar performance, while for the remaining, floating-point applications, the SMT processor is better than the dynamic superscalar.

From Figure 6, we see that in each of the four floating point applications, the four threads of the SMT processor are used in a balanced manner. Approximately, one quarter of all instructions are issued by each thread. This is because loops with independent iterations dominate the execution of the applications. When a thread suffers a hazard, issue slots are not necessarily wasted; they may be used by other threads. The integer applications, however, behave differently. According

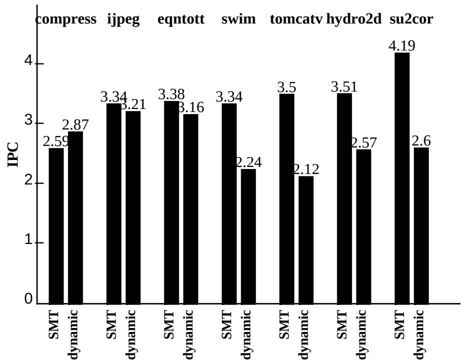


Figure 5: IPC for the applications under the centralized SMT and dynamic superscalar processors.

to Figure 6, the use of the threads is less balanced. In fact, in *compress*, most of the time only one thread is active. The reason is that these applications are less loop intensive than the others. In this environment, the ability of the dynamic superscalar to exploit ILP in the serial sections allow it to perform as well as the centralized SMT. Overall, Figure 5 shows us that the SMT processor has a more stable performance and is an average of 31% faster than the dynamic superscalar processor.

4.2 SMT Processor: Centralized vs Clustered

An obvious problem with the centralized SMT and the dynamic issue processor is that the wire delay in their bypass networks will severely limit how much we can reduce the clock cycle time. As discussed in section 3.2, the centralized SMT architecture has a big disadvantage in cycle time when compared to the clustered SMT and FA architectures.

Figure 7 shows the IPC of the two processors. For most of the applications, the IPC difference between the two architectures is within 5%. This means that the issue limitations of the clustered architecture do not impair its average issue rate significantly under these applications. The only application where such issue limitations are obvious is *compress*, which has large serial sections. In such sections, the only thread in the application can issue up to 8 instructions per cycle in the

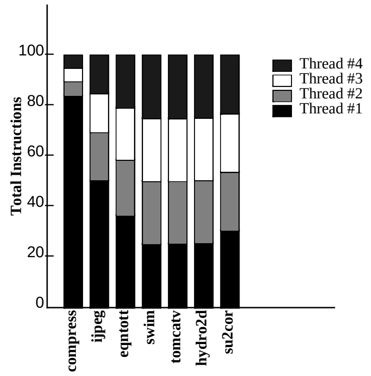


Figure 6: Percentage of instructions issued by each of the four threads in the SMT processor.

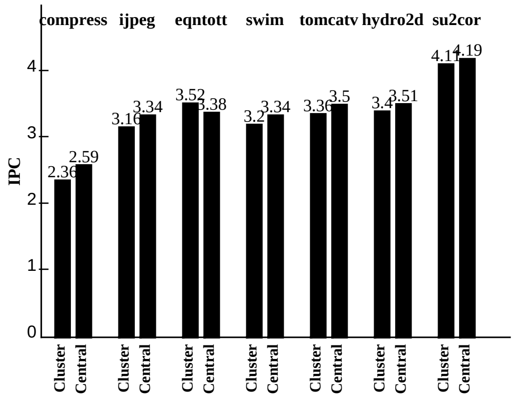


Figure 7: Comparing a clustered and centralized SMT processor with the same cycle time.

centralized SMT processor, while it can only issue up to 4 instructions in the clustered SMT processor. Overall, however, given the small differences in IPC and the large difference in the clock cycle time between these two architectures, we conclude that the clustered SMT delivers much more performance than the centralized SMT.

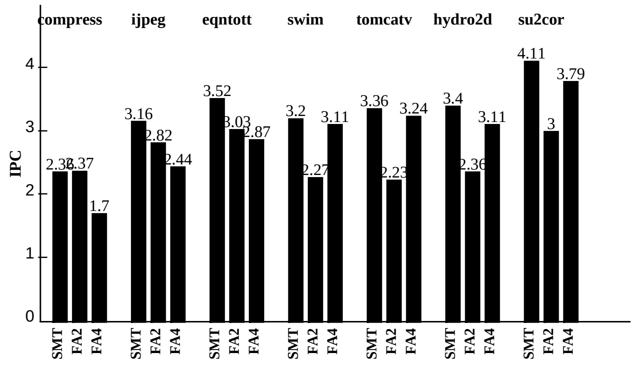


Figure 8: Comparing a clustered SMT processor with FA architectures FA2 and FA4.

4.3 Clustered SMT vs FA Architectures

Recall that, in the FA architectures, each thread has a fixed number of issue slots assigned that it does not share with any other thread. FA_2 has two threads with 4 issue slots each, while FA_4 has four threads with 2 issue slots each. The clock cycle time in these two architectures and in the clustered SMT is comparable. The reason is that the bypass networks in the three architectures are too small to determine the cycle time [7]. Instead, the cycle time is determined by the synchronizing scoreboard, which is the same for the three architectures.

Figure 8 compares the IPC of the applications on the three architectures. In the floating point applications, performance is decided by the thread parallelism. As a result, the architectures with 4 threads, namely the clustered SMT and FA_4 perform much better than FA_2, which only has two threads. FA_2 is about 40% slower than SMT. In the integer applications, which do not have much thread parallelism, it is crucial for the hardware to speed up the serial sections. In both the clustered SMT and FA_2 architectures, the thread executing the serial section has 4 issue slots available, while, in the FA_4 architecture, it has only 2 slots. As a result, the serial section takes longer in FA_4. Overall, FA_4 is about 31% slower than SMT.

This clearly illustrates the main advantage of the clustered SMT architecture over architectures where the issue slots are statically assigned to threads. The SMT has flexibility to handle the varied behavior of the parallel threads extracted from the sequential binaries.

4.4 Memory Dependence Table Size

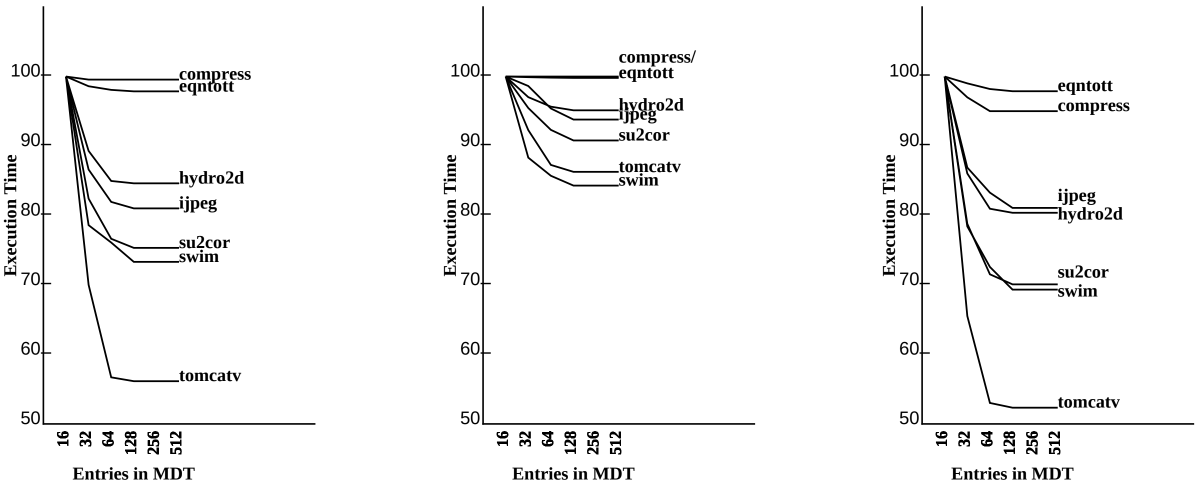
Finally, we look at the effect of the size of the MDT on the performance of the processors. Clearly, if the MDT is too small, the performance suffers. This is because, as discussed in Section 2.2.2, when no MDT entries are free, speculative threads stall. However, a very large MDT is not practical because it may affect the cycle time. In all our previous simulations, we have used a 128-entry MDT. To determine the impact of the MDT size on performance, we now vary the MDT size from 16 to 512 entries.

Figure 9 shows the execution time of the applications under the clustered SMT, FA_2 and FA_4 architectures for different MDT sizes. In the charts, for each application, the execution time is normalized to the one with the 16-entry MDT. Figure 9 shows that there is little change in performance beyond a MDT size of 64 entries. Then, with 128 entries, the performance saturates. Therefore, we recommend having 128 entries. All our previous results would change very little if we had used a 64-entry MDT instead of a 128-entry one. Since 128 entries are reasonable, the results indicate that effective speculation does not demand a large MDT.

5 Conclusions & Future Work

Although basic simultaneous multithreading (SMT) permits resource sharing, its structure is so centralized that it is likely to slow down the clock frequency. The solution that we have explored in this paper is distributed or clustered SMT architectures. We have discussed how we extract parallel threads out of sequential binaries and presented speculative hardware support for inter-thread register synchronization and for memory disambiguation. Our evaluation indicates that the performance of such an architecture is better than an aggressive dynamic superscalar processor, basic SMT and architectures where all the resources have a fixed assignment.

There are some drawbacks to our hardware scheme. We have assumed a centralized synchronizing scoreboard to handle register dependences. Since this may be a bottleneck for higher issue processors, we are currently working on decentralizing this table such that each processor can maintain its own



(a) Clustered SMT (b) FA2 (c) FA4

Figure 9: Impact of varying the number of entries in the MDT for the (a) clustered SMT, (b) FA_2, and (c) FA_4 architectures.

copy of the table. Also, we are improving the memory disambiguation mechanism to allow the first-level caches, rather than separate write-buffers, to hold speculative data.

Acknowledgments

We thank the referees and members of the I-ACOMA group for their valuable feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, June 1995.

[2] M. Franklin and G. Sohi. ARB: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[3] V. Krishnan and J. Torrellas. Efficient use of processing transistors for larger on-chip storage: Multithreading. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[4] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

[5] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[6] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Computer Systems Laboratory, Stanford University, February 1997.

[7] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[8] G. Sohi, S. Breach, and T. Vijayakumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[9] J. Steffan and T. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical Report CSRI-TR-350, Computer Systems Research Institute, University of Toronto, February 1997.

[10] J. Tsai and P. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, October 1996.

[11] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[12] J. Veenstra and R. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.