A Chip-Multiprocessor Architecture with Speculative Multithreading

Venkata Krishnan*
Compaq Computer Corporation
334 South Street
Shrewsbury, MA 01545
Venkata.Krishnan@compaq.com
Phone: 508-841-2198
FAX: 508-841-3579

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
torrellas@cs.uiuc.edu
Phone: 217-244-4148
FAX: 217-333-3501

Abstract

Much emphasis is now placed on chip-multiprocessor (CMP) architectures for exploiting thread-level parallelism in an application. In such architectures, speculation may be employed to execute applications that cannot be parallelized statically. In this paper, we present an efficient CMP architecture for speculative execution of sequential binaries without source re-compilation. We present the software support that enables identification of threads from a sequential binary. The hardware includes a memory disambiguation mechanism that enables the detection of inter-thread memory dependence violations during speculative execution. This hardware is different from past proposals in that it does not rely on a snoopy-based cache-coherence protocol. Instead, it uses an approach similar to a directory-based scheme. Furthermore, the architecture includes a simple and efficient hardware mechanism to enable register-level communication between on-chip processors. Evaluation of this software-hardware approach shows that it is quite effective in achieving high performance when running sequential binaries.

Keywords: Chip-multiprocessor, speculative multithreading, data-dependence speculation, control speculation

*Corresponding Author

1 INTRODUCTION

The superscalar approach [12], which allows more than one instruction to be issued in a single cycle, has become the norm for today's high-performance microprocessors. The issue rate of these microprocessors has continued to increase over the past few years, with today's high-performance superscalar processors such as the Compaq Alpha 21264 [4], IBM PowerPC [16], Intel Pentium-Pro [3] or MIPS R10000 [19] able to issue up to four instructions per cycle.

Most of these processors have special hardware that allows them to dynamically identify independent instructions that can be issued in the same cycle. Typically, this involves maintaining a pool of instructions in a large associative window, along with a register renaming mechanism that eliminates any false dependences between instructions [12]. This mechanism permits an instruction to be issued as soon as its operands and functional unit become available, rather than being issued in the order as defined by the static code sequence. The overall result is a potentially high degree of instruction level parallelism (ILP) extracted from the program at run-time.

Unfortunately, it is unclear whether this approach is appropriate for very wide-issue superscalars. This is because it requires centralized hardware structures that lengthen the critical path of the processor pipeline. Such structures include the register renaming logic, the instruction window wake-up and select mechanisms, and the register bypass logic. The latter allows values to be forwarded directly from the output of functional units to their inputs, thereby permitting back-to-back issue of data dependent instructions.

The primary reason for the design of the conventional superscalar to be so centralized is that it exploits ILP in a single thread or execution path in the application. Consequently, a natural approach to developing a decentralized architecture is to divide the application into multiple threads and exploit ILP across them. Thus, rather than implementing one, very centralized high-issue superscalar processor on a chip, many researchers have proposed a decentralized architecture wherein multiple threads run on multiple simpler processing units on a single chip. This is the chip-multiprocessor (CMP) architecture.

The CMP has drawn great attention, with architects proposing various related designs [6, 11, 14, 17,

```
for (i = 0; i < n; i++)
  .... = a[x[i]];
  a[x[i]] = ....;
end
```

Figure 1: Example of a loop that will not be parallelized by a conservative compiler.

25, 26, 27, 28]. Compared to the conventional centralized approach, the CMP has several advantages. First, its design simplicity allows for a faster clock in each of the processing units. The same simplicity eases the time-consuming design validation phase that plagues complex centralized designs. Moreover, the decentralized network implementation permits fast communication in the processing units' localized interconnects. This is in contrast to the long-latency interconnect in the centralized approach.

Finally, the CMP approach also results in better utilization of the silicon space. By avoiding the extra logic devoted to the centralized architecture, a CMP allows the chip to have a higher overall issue bandwidth when compared to a conventional superscalar implemented on the same die area. Indeed, Olukotun et al [10] show how a CMP with eight 2-issue superscalar processing units would occupy the same area as a conventional 12-issue superscalar processor. Overall, from a design perspective, the CMP is a promising approach.

From a software perspective, the CMP is an ideal platform to run a multiprogrammed workload or a multithreaded application. However, if the CMP is to be fully accepted, it must also be able to give good performance when running sequential applications. Since parallelizing compilers [1, 9] are successful only for a restricted class of applications, typically numerical ones, the CMP approach would not be able to handle a large class of general-purpose sequential applications. Even for numerical applications, the compiler is very conservative. For instance, the compiler assumes the existence of inter-thread dependences when it cannot guarantee data independence between threads. This is illustrated in Figure 1. If the values in array x are dependent on inputs to the program, it is impossible for the compiler to prove that each iteration of the loop accesses distinct array locations. As a result, this loop segment is marked serial by the compiler. The situation becomes even worse for non-numerical applications, which often access data through pointers.

To address this problem, speculation may be used. First, speculative threads need to be identified in the application. They may be identified either at compile time [5, 11, 14, 26, 27, 28] or completely at run time with hardware support [17, 23]. Then, the different threads are executed in parallel speculatively. Added software or hardware support enables detection of and recovery from dependence violations. In this mode, the threads that execute on the on-chip processing units do not need to be fully independent; they may have data dependences with each other. When a dependence violation between two threads is detected, the thread violating the dependence along with its successors are squashed and then re-executed.

Indeed, it has been shown that there is considerable performance potential for even integer applications when using this speculative approach [27]. Consequently, CMPs can not only be utilized as a very effective throughput-enhancing platform when running multiprogrammed workloads, but can also be used for achieving good speedups for a wide range of single-application workloads.

In this paper, we present an efficient CMP architecture that speculatively executes sequential binaries without the need for source re-compilation. It uses software support to identify threads from a sequential binary. It includes memory disambiguation hardware to detect inter-thread memory dependence violations. Such hardware is different from past proposals in that it does not rely on a snoopy-based cache-coherence protocol. Instead, it uses an approach similar to a directory-based scheme. Furthermore, the architecture includes a simple and efficient hardware mechanism to enable register-level communication between on-chip processors without complicating the architecture much. In this paper, we also evaluate the architecture and show that it runs sequential binaries with high performance.

The remainder of this paper is organized as follows. Section 2 describes related work on speculation-based CMP architectures; Section 3 describes our software support for identifying threads in a sequential binary; Section 4 describes the hardware support for register communication and for enforcing inter-thread true memory dependences; Section 5 describes our evaluation approach; Section 6 evaluates the system; finally, Section 7 concludes the paper.

2 RELATED WORK

There have been two major approaches for configuring multiple processing units on a chip. In one approach, the architecture is fully geared towards exploiting speculative parallelism. Typical examples are the Trace [24, 25], Multiscalar [26], and Dynamic Speculative Multithreaded processors [17]. These processors can potentially handle sequential binaries without re-compilation of the source. As such, these processors have hardware features that allow communication both at the memory and the register level. For instance, in the Trace processor, additional hardware in the form of a trace cache [23] assists in the identification of threads at run time, while inter-thread register communication is performed with the help of a centralized global register file. In the Multiscalar processor, though threads are identified by recompiling the source program, they can also be identified statically from the binary. At run time, register values are forwarded from one processor to another with the aid of a ring structure. Recovery from mis-speculation is achieved by maintaining two copies of the registers, along with a set of register masks, in each processing unit [2]. The Dynamic Speculative Multithreaded processor [17] follows the Multiscalar approach and also incorporates additional hardware to allow threads to be generated dynamically at run-time. Overall, these processors have sufficient hardware support to tailor the architecture for speculative execution. This enables them to deliver high performance on existing sequential binaries without the need for full re-compilation of the source program. A direct consequence of this, however, is that a large amount of hardware remains un-utilized when running a fully-parallel application or a multiprogrammed workload.

In the second approach, the CMP is generic enough and has only minimal support for speculative execution [11, 27, 28]. Current proposals for these systems restrict the communication between processors to occur only through memory. Such limited hardware may be sufficient when programs are compiled using a compiler that is aware of the speculation hardware [27]. However, the need to re-compile the source is a handicap, especially when the source is unavailable.

Our proposal for a speculation-based CMP architecture attempts to combine the best of the above two approaches. First, rather than requiring source re-compilation, we operate on sequential binaries. Towards this, we have designed a binary annotation tool that extracts multiple threads from sequential binaries to execute on the CMP. Second, to execute sequential binaries speculatively, we devise some modest hardware support. This hardware allows communication both through memory and registers, without the need for additional register sets. Register-level communication is important in an environment that takes in sequential binaries. Overall, we propose an architecture that adds little hardware to a generic CMP, while being able to handle sequential binaries quite effectively.

3 SOFTWARE SUPPORT

To run an application on a speculative CMP architecture, we first need to identify threads. This may be achieved in software by performing a compilation step [5, 11, 26, 27, 28] or by using hardware support that identifies threads at run time [17, 23]. We use a software approach. However, we perform the compilation step on the sequential executable file. As a result, we do not need to recompile the program and can operate on legacy codes.

We have developed a binary annotator that identifies units of work for each thread and the register-level dependences between these threads. Currently, we limit the threads to loop iterations. In our analysis, we mark the entry and exit points of each loop. During the course of execution, when a loop entry point is reached, multiple threads are spawned to begin execution of successive iterations speculatively. Only one iteration is non-speculative, with all its successors having a speculative status. When a non-speculative thread completes, the immediate speculative successor acquires non-speculative status. To enable interrupt and exception handling, the processor must maintain a sequential state at all times. This is achieved by forcing each speculative thread to maintain its state separately. Only the non-speculative thread is allowed to update the sequential state of the processor.

To simplify the hardware support, we follow sequential semantics for thread termination: we wait for the thread to reach non-speculative status before it can be retired and a new thread initiated on the same processor. If sequential semantics were not adopted, additional hardware would be required to buffer the state of the speculative thread that completed before initiating a new thread on the processor. This is because it may not yet be safe for the speculative thread to update the sequential state of the processor.

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, if a succeeding iteration loaded a memory location before a predecessor stored to the same location, all iterations starting from the one that loaded the wrong value will be squashed and restarted.

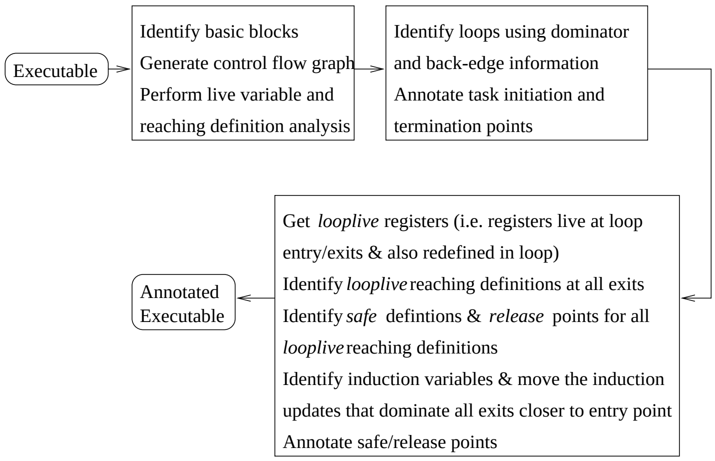

Figure 2: Binary annotation process.

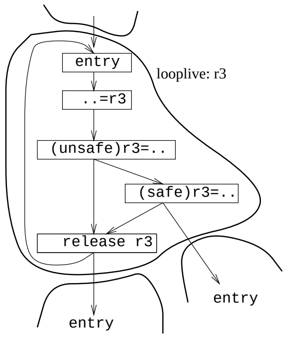

Figure 3: Safe definitions and release points.

The steps involved in the annotation process for binaries are illustrated in Figure 2. The approach that we use is similar to that of Multiscalar [26], except that we operate on the binary instead of on the intermediate code. 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, which are 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 will 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 3 illustrates the safe definition and release points for a looplive register $r3$. These points are identified by first performing a backward reaching definition analysis. This is followed by a depth-first search, starting at the loop entry point, for each and every looplive reaching definition. Finally, induction variables are identified and their updates are percolated closer to the thread entry point, provided that 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.

Note that, unlike register dependences, memory dependences cannot be easily identified from the binary. Therefore, we assign the full responsibility of detecting memory dependences to the hardware. This is detailed later in the paper.

At present, our current approach of analyzing sequential binaries is restricted to loop iterations. Consequently, we can only examine applications which are largely loop-based. However, the approach can be easily expanded to include other sections of code by using heuristics similar to that used for task selection in the Multiscalar processor [30].

In closing, it must be noted that incorporating the annotations in a binary is quite simple and requires only minor extensions to the ISA. Additional instructions are needed only for identifying thread entry, exit, and register value release points. We assume hardware support for thread initiation and termination. All the hardware support is detailed in the next section.

4 HARDWARE SUPPORT

Application threads can communicate with each other through registers or via memory. The former is important in the context of parallelizing sequential binaries. Since compilers perform good register allocation, register-level inter-thread dependences are common. Since these dependences can be found accurately in the binary code, we can enforce them in hardware, thereby preventing unnecessary squashing of threads. To enable flexible inter-thread register communication, we propose augmenting a conventional scoreboard to what we call a *Synchronizing Scoreboard (SS)* (Section 4.1).

In contrast, identifying memory dependences is difficult at the binary level. Therefore, the hardware is fully responsible for identifying and enforcing memory dependences. This hardware should ensure that speculative and non-speculative versions of data are not mixed up and must allow speculative threads to acquire data from the appropriate producer thread. It must also identify dependence violations that may occur when a speculative thread prematurely accesses a memory location. This will result in the squashing of the violating thread along with its successors. The hardware that is required may be a centralized buffer along the lines of the ARB [7]. Alternatively, it may involve a decentralized design, where each processor's primary cache is used to store the speculative data, along with enhancements to the cache coherence protocol to maintain data consistency. We use the latter approach for our hardware. There has been work done concurrently with ours, such as the Speculative Versioning Cache (SVC) [8], which makes use of a snoopy bus to maintain data consistency among different processors. Our work differs from SVC in that we do not rely on snooping; instead, we use an approach similar to a directory-based cache-coherence scheme. We call our approach the *Memory Disambiguation Table (MDT)* (Section 4.2). In the following, we present the SS and the MDT in turn. Our discussion assumes a 4-processor CMP.

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.

RegID	StartSync $F_0F_1F_2F_3$	Sync $S_0S_1S_2S_3$
0		
1	0 1 0 0	0 1 0 0
...		
15	1 0 1 0	1 0 0 0
16	0 0 0 0	0 0 0 0
...		

Table 2: Example of the global fields of a SS.

For our 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 executes the current iteration. Speculative successors 1, 2, and 3 execute the successor iterations, which we call the first, second, and third speculative iteration respectively. As threads complete, the non-speculative *ThreadMask* will move from one thread to its immediate successor and so on.

4.1 REGISTER-LEVEL COMMUNICATION WITH THE SYNCHRONIZING SCOREBOARD

The Synchronizing Scoreboard (SS) is a fully decentralized structure used by threads to synchronize and communicate register values. It is a scoreboard augmented with additional bits. Each processor has its own copy. The SSs in the different processors are connected with a broadcast bus, on which register values are transferred. This bus, which we call the *SS Bus*, has a limited bandwidth of 1 word per cycle. It has one read and one write port for each processor. For a 4-processor CMP, a value written by a processor onto the bus takes anywhere between 1 to 3 cycles to get to the destination processor, depending on the physical distance between the producer and the receiver processors. This latency is based on assuming 0.13 μm chip technology, where a signal would take up to 4 clock cycles to traverse the entire die [18]. In addition, there may be the added delay of contention for the SS bus. The overall hardware setup is shown in Figure 4.

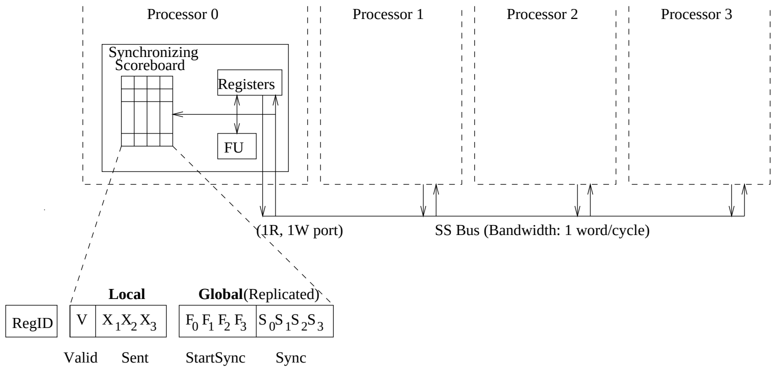

Figure 4: Hardware support for register communication.

As in a conventional scoreboard, each SS has one entry per register. Figure 4 shows the different fields for one entry. The fields are grouped into *local* and *global*. To avoid centralization, the latter bits are replicated but easily kept coherent across the several SSs in different processors. This is described later in the section. The global fields include the *Sync (S)* and the *StartSync (F)* fields. Each of these fields has one bit for each of the processors on chip. Table 2 shows an example of the global fields of a SS.

For a given register, the S_i bit, if set, implies that the thread running on processor i has not made the register available to successor threads yet. When a thread starts on processor i, it sets the S_i bit for all the looplive registers that it may create. The S_i bit for a register is cleared when the thread executes either a safe definition or the release instruction for that register (Section 3). When this occurs, the thread also writes the register value on the bus, thereby allowing other processors to update their values if needed. At that point, the register is safe to be used by successor threads.

The F bit is set to the value of the corresponding S bit when the thread is initiated. This is done with dedicated hardware that, when a thread starts on processor i, initializes the F_i and the S_i bits for all the registers in the SS of all processors. From then on, the F bit remains unchanged throughout the execution of the thread. Thus, it always remains the same across all the SSs. We will see the use of the F bits later.

Each processor has an additional local *Valid (V)* bit for each register, as in a conventional scoreboard. This per-processor private bit tells whether the processor has a valid copy of the register. When a parallel section of the code is reached, the processors that were idle in the preceding serial section start with their V bits set to zero. The V bit for a register is set when the register value is generated by the local thread or is communicated from another processor.

Within a given parallel section, a processor can reuse registers across threads. When a processor initiates a new thread, namely the latest speculative thread, it sets its V bits as : $V = V - \cup F_{pred}$. This effectively invalidates any registers that are written by any of the three predecessor threads, while allowing other registers to remain valid.

To understand how the SS works, we consider how registers are communicated between processors and the problem of the last copy. Finally, we look at the complexity of the SS.

4.1.1 REGISTER COMMUNICATION BETWEEN PROCESSORS

Register communication between threads can be *producer-initiated* or *consumer-initiated*. The producer-initiated approach has already been outlined. When a thread clears the S bit for a register, it writes the register on the SS bus. At that point, each of the successor threads checks its V bit for the register and also the F bits for the register of the threads between the producer (not inclusive) and itself (inclusive). If all these bits are zero, the successor thread loads the register and sets the V bit corresponding to the register to 1. At the same time, all processors clear the S bit corresponding to the producer thread in their SSs. The F bits, however, remain unchanged.

It is possible that the consumer thread is not running when the producer generates the register. We could allow the values to be stored, by using a buffered communication mechanism, rather than using a simple broadcast bus. The buffer would potentially hold all the live registers after the last speculative thread until a new thread is initiated on the successor. In addition, this would also require further hardware support in the form of duplicate register sets in each processor to enable recovery from squashes [2]. Alternatively, a global register set may be maintained to store these values [24], but at the cost of maintaining a centralized structure.

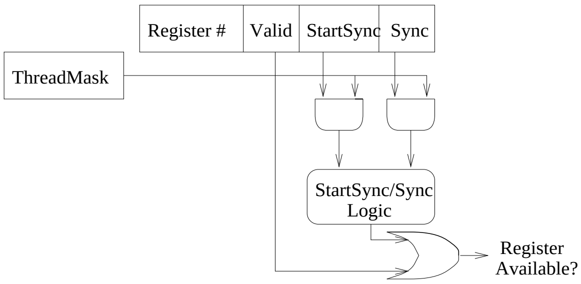

Figure 5: Logic to check register availability.

In our scheme, we add minimal hardware to support a consumer-initiated approach, where communication occurs when the consumer finally runs and needs the register. To support it, the SS has logic that allows a consumer thread to identify the corresponding producer and get the register value. The logic works as follows. The consumer thread first checks its V bit for the register. If it is set, the register is locally available. Otherwise, the F bit of the immediately preceding thread is checked. If it is set, the predecessor thread is the producer. If the predecessor's S bit is set, it means that the register has not been produced yet and the consumer blocks. Otherwise, the consumer gets the register value from the predecessor. However, if the thread immediately preceding the consumer has F equal to zero, that thread cannot be the producer. In that case, the bit checks are repeated on the next previous thread. This process is repeated until the non-speculative thread is reached. For example, assume that thread 0 is the non-speculative thread, that threads 1, 2, and 3 are speculative, and that thread 3 tries to read a register. In that case, the register will be available to thread 3 if:

$$V_3 + \overline{S}_2(F_2 + \overline{S}_1(F_1 + \overline{S}_0)) \qquad (1)$$

Suppose now, instead, that thread 1 is the non-speculative thread, that threads 2, 3, and 0 are speculative threads, and that the access came from thread 0. In that case, the register will be available to thread 0 using a similar equation:

$$V_0 + \overline{S}_3(F_3 + \overline{S}_2(F_2 + \overline{S}_1)) \qquad (2)$$

The accesses to these bits are always masked out with the $ThreadMask$ of Table 1. In examples (1) and (2), the request came from speculative successor 3. Therefore, we have used mask 1111, thereby enabling all bits and computing the whole expression (1) or (2). Consider a scenario like in (2), where thread 1 is non-speculative, except that the access came from thread 3 (speculative successor thread 2). Consequently, we would use $ThreadMask$ 0111 from Table 1. This means that we are examining only 2 predecessors. The resulting function is:

$$V_3 + \overline{S}_2(F_2 + \overline{S}_1)$$

Overall, the complete logic to determine whether a register is available is shown in Figure 5. This extra logic is added for each register in the processor. If the register is available, the reader thread gets the value from the closest predecessor whose S bit is clear and F bit is set. If all the bits are clear, the non-speculative thread provides the value. The transfer of the value is initiated by the consumer thread putting a request on the SS bus to read the register from the appropriate thread. The request and the reply messages can take 1-3 cycles each, depending on the producer to consumer distance, plus the contention for the SS bus. Even though the delay results in the SS entries (specifically, the S bit) to be inconsistent across processors for a short period, it has no effect whatsoever on the overall mechanism. Note that the SS is used primarily by successors to check if the predecessors have produced the value. A delay in the value reaching the consumer just delays the clearing of the corresponding bits in the consumer. Consistency is restored as soon as the value is obtained by the consumer. The bandwidth that is required to obtain optimal performance is evaluated later in the section.

Example:

This example illustrates how the SS entries change for a register $r3$. Let threads t, $t+1$, $t+2$, and $t+3$ execute on processors 0, 1, 2, and 3. Assume that all threads, except t, have $r3$ marked invalid and that thread $t+1$ produces a live-out value. The scoreboard entry appears as follows. Note that each V bit is local to the processor and is denoted by the subscript, while the F and S bits are global and replicated in each thread.

Processor	0	1	2	3
Thread	t	$t+1$	$t+2$	$t+3$
Status	non-spec	spec1	spec2	spec3

V_0	V_1	V_2	V_3	$F_0F_1F_2F_3$	$S_0S_1S_2S_3$
1	0	0	0	0 1 0 0	0 1 0 0

When thread $t+1$ updates $r3$, it clears its S bit and writes the result for its successors to read. This is a producer-driven approach. The scoreboard entry looks as follows:

Processor	0	1	2	3
Thread	t	$t+1$	$t+2$	$t+3$
Status	non-spec	spec1	spec2	spec3

V_0	V_1	V_2	V_3	$F_0F_1F_2F_3$	$S_0S_1S_2S_3$
1	1	1	1	0 1 0 0	0 0 0 0

Note that $r3$ in processor 0 is stale. Now, assume that t completes and a new thread $t+4$ is initiated on processor 0. At this point, V for processor 0 (V_0) is set according to $V_0 = V_0 - \cup F_{pred}$. Since $F_1 = 1$, V_0 is set to 0. The scoreboard entry looks as follows:

Processor	0	1	2	3
Thread	$t+4$	$t+1$	$t+2$	$t+3$
Status	spec3	non-spec	spec1	spec2

V_0	V_1	V_2	V_3	$F_0F_1F_2F_3$	$S_0S_1S_2S_3$
0	1	1	1	0 1 0 0	0 0 0 0

Now, when $t+4$ tries to read $r3$, it uses the register availability logic, $V_0 + \overline{S}_3(F_3 + \overline{S}_2(F_2 + \overline{S}_1))$, which evaluates to TRUE. Thread $t+4$ determines that the value is available in the non-speculative thread $t+1$ and puts a request on the SS bus. This is the consumer-driven approach. Finally, when processor 1 supplies the value to the bus, $r3$ in processor 0 becomes valid.

4.1.2 THE LAST COPY PROBLEM

When the last speculative thread updates a register, it has no successors to which it can send the value. As a result, any future consumer threads will have to explicitly request the value from it. Also, recall that when a new thread is initiated, it invalidates any local register that a predecessor may produce. Under these conditions, a situation may occur where all the copies of a given register on chip are about to become invalid. We term this scenario as the the last copy problem.

The last copy problem is illustrated in Figure 6. In the example, register $r3$ is live across all threads, with each thread reading the value before writing it. Therefore, any thread will invalidate its local copy of $r3$ on initiation. In Figure 6-(a), the last speculative thread (thread 3) updates $r3$ and no other thread consumes it. Some time later, threads 0, 1, and 2 have finished, threads 4, 5, and 6 have started in their place, thread 3 has acquired non-speculative status, and it is about to finish (Figure 6-(b)). If we now spawn thread 7 on the rightmost processor, we face the last copy problem: $r3$ will be lost. This is because thread 4 has not read $r3$ yet, and thread 7 will clear its valid bit for $r3$ after spawning.

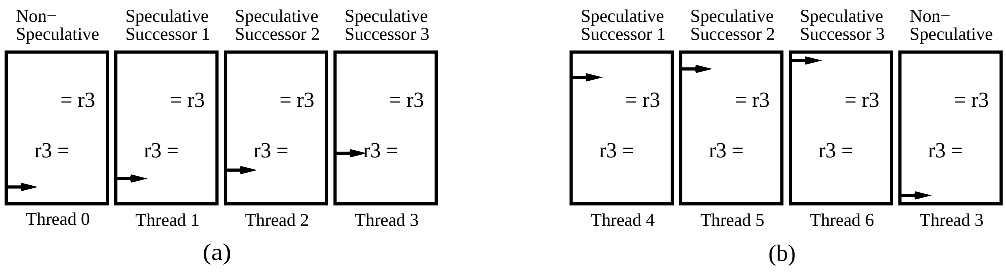

Figure 6: The last copy problem. The arrow points to the instruction currently being executed in each thread.

The last copy problem will not occur if we use a communication mechanism that buffers live-out values to percolate to the new threads, or if we use a centralized global register set that maintains live-out values. For instance, the Multiscalar processor [26] uses the first approach. A ring structure is used to forward register values. Thus, all values move from one thread to another in the ring. They are buffered after the last speculative thread until they can be forwarded further. The forwarding can proceed once the non-speculative thread is completed and a new thread is initiated on it. Since this is a fully producer-driven approach, registers are backed up in case the consumer thread is squashed. Thus, each processor maintains two copies of the register file: one to maintain the past values and the other to store the present values. Forwarded copies from predecessors are held by the *past* register set while the new ones created by the thread are held in the *present* set. In addition, to restore the state, up to 6 different register masks are maintained in each processor [2].

The Trace processor [24] avoids the last-copy problem by keeping a centralized global register set that is visible to all processors. This is in addition to the local register set in each processor. Live-out register values are sent to the global registers, from where any processor can read them.

In both of the above approaches, therefore, the architecture provides significant support for speculation. All these resources remain unutilized when running an explicitly-parallel program.

Our SS design can be enhanced with simple hardware support to overcome the last-copy problem. The idea is for each thread to remember which of the other 3 threads it has sent the register to. Consequently, each processor has 3 extra local bits per register called the *Sent (X)* bits. They are set if the corresponding thread has been forwarded the register in the past, either in a producer- or consumer-initiated transaction. These bits are used as follows. Before we retire a non-speculative thread, we examine the *Sent* bits to ensure that no last copies of registers are going to be lost. For any such last copy, the thread simply writes the register on the SS bus, so that speculative threads can read it.

The logic used by the non-speculative thread to identify last copies is as follows: Assume that processor 0 performs the check. For each of the looplive registers that it produces (those with the F_0 bit set), the register needs to be written on the SS bus if $X_1(F_1 + X_2(F_2 + X_3))$ evaluates to FALSE. The idea is to check if the live value has reached up to the thread that kills the value. If F_i is set, then that thread kills the register. Note that, except for a few negations, this equation is similar to the one used to check register availability in Section 4.1.1. The logic is replicated for each register as in the case of register availability. At thread retire time, each register is checked in parallel to check for last-copy status. Finally, when a new thread is initiated on a processor, the remaining processors clear the corresponding X bit thereby noting that the value is yet to be sent to the new thread.

4.1.3 COMPLEXITY OF THE SYNCHRONIZING SCOREBOARD

To conclude our discussion of the SS mechanism, we examine the area required for its implementation and its potential impact on the processor's cycle time.

To estimate the area, we need to consider first the logic to check for register availability (Section 4.1.1) and to check for last copy status (Section 4.1.2). The AND-OR logic, which is traditionally implemented as a carry-propagate-kill function, and the extra gates to selectively mask out some of the S, F, and X bits require only a few gates. Replicating this logic for each register implies an extra overhead of only a few hundreds of gates even for a processor with a large number of registers.

More importantly, there is the extra space required by the register file in each processor. The number of bits to store the V, X, F, and S bits per register is $3n$, where n is the number of processors on chip (Figure 4). For a 4-processor CMP with 64-bit registers, this works to around 12% storage overhead. Overall, we feel that these are modest hardware requirements when compared to replicating the register sets in each processor [2] or using a centralized global register file [24].

As for the impact on cycle time, if we refer to equation (1) in Section 4.1.1, it may seem that, in the worst-case scenario where all the bits have to be considered, the delay incurred by the SS logic increases quickly with the number of threads supported on the processor. However, by using a binary-tree approach, the logic can be implemented using only $\log_2 n$ levels of gates, where n is the number of processors in the CMP. Consequently, given the small numbers of processors that can be placed on a chip, this circuitry is shallow and unlikely to affect the cycle time.

Furthermore, the SS bus is implemented with staging buffers. By pipelining the bus in this manner, we likely eliminate any impact on the processor cycle time. However, a message from one processor to another one may take several cycles: from 1 to 3 cycles.

Finally, we note that the complexity that we add to the register files is modest: we add only one read and one write port for each register file. As a result, only one value can be written to or read from the bus per processor in a given cycle. However, as we show later in our evaluations, this limitation has little impact on performance.

4.2 HARDWARE SUPPORT FOR HANDLING MEMORY-LEVEL DEPENDENCES

Unlike register dependences, we do not identify memory dependences in the binary annotation phase; we assign the full responsibility for it to the hardware. In this section, we explain the hardware support that is required to identify and possibly recover from memory-level dependences.

When an application runs in speculative mode, it can potentially generate wrong data. Thus, we need to separate the speculative from the non-speculative data. Only the non-speculative thread is allowed to modify the sequential state of the program. Since the sequential state is available at all times during the execution of the program, this restriction greatly eases the recovery from mis-speculations and exception handling. Consequently, in our scheme, each processor in the CMP has a private L1-cache with special support. During speculative execution, speculative threads use the L1-cache in a restricted write-back mode: the L1 is write back but it cannot displace dirty lines. When a dirty line is about to be displaced, the thread stalls.

Unlike the speculative threads, the non-speculative thread is allowed to update the shared L2-cache, since its store operations are safe. Consequently, when a speculative thread acquires non-speculative status, dirty lines are allowed to be displaced from the L1-cache. Furthermore, the L1-cache starts working in write-through mode. After the iteration completes and before it can be committed, any remaining dirty lines in the cache are flushed to memory. This allows us to start a new thread with a clean cache. Note that neither write-through caches nor cache flushing would be necessary if the L1-cache kept extra bits to mark committed words that remain dirty in cache after a new, speculative thread starts in the processor. Such bits are used in the Speculative Versioning Cache scheme [8]. However, to keep the system simple, we do not support such bits in our scheme.

Under the caching environment described, we identify memory dependence violations with the help of a Memory Disambiguation Table (MDT). The MDT is somewhat similar to a directory in a conventional shared-memory multiprocessor. Depending on its size, the MDT may be incorporated into the L2 cache or may be located on-chip. The details on the MDT are explained later in this section.

To enable speculative execution, each word cached in the private L1-caches is augmented to have the following bits:

Flush (F)	Invalid (I)	Dirty (D)	SafeWrite (SW)	SafeRead (SR)	Forward (FD)

Table 3: Bits per cached word.

The *Invalid (I)* and *Dirty (D)* bits serve the same purpose as in a conventional cache.

The *SafeWrite (SW)* and *SafeRead (SR)* bits are used by the speculative threads. To understand their meaning, consider the following. In theory, each processor, when performing a load or a store operation, may have to inform the MDT, which tracks memory dependence violations. The *SW* and *SR* bits allow the processor to perform load and store operations without informing the MDT.

Specifically, the *SW* bit, if set, permits a thread to perform writes to a word without informing the MDT. When a thread performs a store for the first time, it sets the *D*, *SW*, and *SR* bits. It also sends a message to the MDT. Subsequent stores to the word can be done without any messages to the MDT, provided the *SW* bit is set. The *SW* bit is cleared when a new thread is initiated on the processor. It is also cleared when a successor thread loads the same word and the MDT forwards the request to the processor with the *SW* bit set. We clear *SW* because, if the thread stores again to the same word, a message will be sent to the MDT. This enables the MDT to flag that a speculative thread has performed a premature load.

The *SR* bit, if set, allows the thread to perform load operations from the word without informing the MDT. This bit is set when the thread loads from or stores to the word for the first time. It is cleared when a new thread is initiated on the processor.

The *Flush (F)* bit is used to invalidate stale words from the L1-cache before a new thread is initiated on the processor. When a thread stores to a word, any words with the same address have to be invalidated from the caches of all its successors. As for the predecessors, information needs to be maintained to denote that the word will become stale in their caches after they complete. The *F* bit identifies those cached words that need to be invalidated across thread initiations, while allowing the reuse of the remaining words. Consequently, when a thread is initiated on a processor, the *I* bit is automatically set to: $I_{new} = I_{old} \vee F$. The *F* bit is cleared after this operation.

Finally, the *Forward (FD)* bit is used to identify forwarded data from other processors. The role of this bit is explained later. Maintaining these bits with a cache-line granularity could result in false dependence detection and, consequently, lead to unnecessary squashing of threads. So, we maintain information at a word level.

4.2.1 MEMORY DISAMBIGUATION TABLE (MDT)

The MDT performs the disambiguation mechanism. It keeps entries on a per memory-line basis and, like the L1-caches, maintains information on a per-word basis. For each word, it keeps a *Load (L)* and a *Store (S)* bit for each processor. When a new thread is initiated on a processor, all its L and S bits are cleared. As the thread executes, the MDT works like a directory that keeps track of which processors shared which words. Table 4 shows a MDT for a 4-processor CMP. Only the state for word 0 of the lines is shown.

The MDT is placed between the per-processor L1-caches and the shared L2-cache, and is distributed in multi-banked fashion. It receives all the requests that are not intercepted by the L1-caches. Later, in our evaluations, we show that the size of the MDT can be quite small. Consequently, it may be configured on chip. The MDT could also be incorporated into the L2-cache. In the following, we first explain how the MDT works when it receives a load or store message from a processor.

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

Table 4: Memory disambiguation table (MDT).

Load Operation

A load from the non-speculative thread does not need to perform any special action, and it can proceed to the L2-cache normally. Similarly, if a load from a speculative thread finds the word in the L1-cache with *SR=1*, the MDT is not accessed. However, when a speculative thread issues a load that misses in the L1-cache or that hits a word with *SR=0*, we have to access the MDT to find an entry that matches the address of the load instruction. If no entry is found, a new one is created and the L bit corresponding to the thread and word requested is set. However, if an entry is already present, it is possible that a previous thread has updated the word. Therefore, the S bits are checked to determine the ID of any predecessor thread that has updated the word. 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 second 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 L1-cache has to be read. Let us follow the example from above and use the state shown in Table 4. Assume threads 0, 1, 2 and 3 are executing on processors 0, 1, 2 and 3 respectively. When thread 2 reads word 0 corresponding to address tag 0x1234, the request is forwarded by the MDT to processor 1 because $S_1 = 1$. Processor 1's L1-cache supplies the word. If processor 2 also missed in its L1-cache, the remainder of the line is supplied from the L2-cache and is merged with the word forwarded from processor 1. Finally, if the S bits of all predecessor threads are 0, the load proceeds to read from the L2-cache.

Store Operation

Unlike loads, stores can result in the squashing of threads if a successor speculative thread prematurely loaded the word. Successor, therefore, rather than predecessor threads are tested on a store operation. All the stores from the non-speculative thread access the MDT, since we use a write-through L1-cache. For the speculative threads, stores that miss in the L1-cache or that hit a word with $SW{=}0$ also access the MDT. When the MDT is accessed, we check for an entry corresponding to the address being updated. If the entry is not found, speculative threads create a new entry and set the corresponding S bit.

If an entry is present, however, a check is made for premature loads performed by one of the thread's successors. Both the L and S bits must be checked. 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. The *check-on-store* logic is:

$$L_1 + \overline{S}_1(L_2 + \overline{S}_2L_3)$$

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 is a false (output) dependence that must be ignored. For example, the last row of Table 4 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 (the *reading* thread) and all its successor threads are squashed and restarted. Note that we also squash all the threads following the *reading* one because it is possible that the *reading* thread has forwarded incorrect data to its own successors. We also invalidate the updated word from the cache of all the squashed threads.

Finally, two more operations are always performed on a store. The MDT sends a message to set the F bit for the word in the L1-caches of the predecessors. Recall that the F bit means that the word remains valid only during the course of execution of their current thread. In addition, an invalidation message is sent to all successors up to, but not including, the one whose S bit is set. This is necessary to prevent all these successor threads from reading an outdated version of the data from their own L1-caches. Contrast this to a conventional SMP, where a write causes the invalidation of the line in the caches of all processors.

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 one. If no entries are available, the thread stalls until one entry becomes available. The non-speculative thread is not affected in any way, and can continue to perform load and store operations with a full MDT.

A MDT entry becomes available when all its L and S bits are zero. These bits are cleared in the following situations. First, when a thread starts, all its L and S bits are cleared. When a thread becomes non-speculative, all its L bits are cleared. We cannot clear its S bits because they indicate that its L1-cache contains the only copy of the non-speculative version of the word. The S bits for the non-speculative thread are cleared progressively: when the dirty word is displaced from the cache, when the dirty word is written-through to memory, and when the dirty word is flushed from the cache right before the commit point.

In practice, lack of free MDT entries is not an important issue: we will see in Section 6.3 that the MDT does not need to be large for the application to deliver good performance.

4.2.2 WHAT HAPPENS WHEN A NEW THREAD IS STARTED OR THREADS ARE SQUASHED?

At the point of thread initiation, all words whose F bits are set are invalidated. The F, SW, and SR bits are cleared. As for the MDT, the corresponding L and S bits are also cleared for this new thread. No special action is taken when threads are squashed, as new threads will be eventually initiated on those processors and all the actions described above will be performed.

When a thread is squashed, we need to invalidate from the L1-cache of its processor some additional words: those that are dirty and those that were forwarded from a squashed predecessor thread. To identify a superset of the latter, we add a *Forward (FD)* bit to each cache word. This bit is set when the MDT informs the processor performing a load that the data is being forwarded from another processor's L1-cache. Thus, when a thread is squashed, we need to invalidate from its L1-cache the words with the D or FD bit set, with $I_{new} = I_{old} + D + FD$. Note that we may be also invalidating words that were forwarded from non-squashed threads. For simplicity, we do not keep any record of where the data came from.

Overall, our scheme induces a 6-bit overhead per word cached in the L1-cache (including the D and I bits). The coherence protocol is similar to that of a conventional directory-based scheme. Like a conventional directory, the MDT can also be distributed. For example, it can be distributed in several banks according to the physical address of the memory line. Finally, the MDT is a separate dependence-tracking device that can enable further improvements. For example, it could be extended on the lines of [20] for enforcing memory dependences, thereby disallowing premature loads from occurring.

4.2.3 COMPARISON WITH OTHER SCHEMES

There have been other schemes that have been proposed earlier or concurrently with our proposal, and that permit speculation at the memory level. The Address Resolution Buffer (ARB) is one such scheme [7]. The ARB is a specialized module placed between the multiple processing units of a Multiscalar and its shared L1-cache. It provides storage for speculative data and also identifies dependence violations. In such a scheme, the L1-cache stores the sequential state of the program. All memory accesses from all processing units are streamed through the ARB before they can reach the L1-cache. It is a very centralized scheme and would hurt performance due to the added latency of L1-cache accesses.

There are three other schemes besides ours that allow each processor in the CMP to have a private L1-cache or buffer. Oplinger et al [21] have proposed a scheme that uses extra buffers, rather than the L1-cache, to hold speculative data. Although this simplifies the protocol, the small buffers may become full and stall the speculative threads. Steffan and Mowry [27] outline a scheme that allows the L1-cache to buffer the speculative state. However, their description does not mention a precise cache-coherence scheme for handling data dependences. The Speculative Versioning Cache (SVC) [8], developed concurrently with our work, is a detailed architectural proposal for handling dependences. Unlike our MDT scheme, which uses a directory-like approach, the SVC scheme uses snooping caches for detecting memory dependence violations. Finally, the Multi-Value cache [17] maintains multiple states of the same memory location and uses a broadcast mechanism for conveying information between threads.

4.3 OVERALL SUPPORT FOR SPECULATION

In this section, we summarize the overall support that is required to run applications speculatively on our CMP. First, we need to predict the starting address of the next thread. Since, in our current work, we are restricting the threads to loop iterations only, the starting point of the next thread is always the same as the current one, namely the beginning of the loop iteration. As a result, predicting the starting address of the next thread is trivial.

Since the grain size of the tasks used in our system is small, we need hardware for thread initiation and termination. The hardware for thread initiation includes support for initializing the extra bits in the SS, L1-cache, and MDT entries. The hardware for thread termination includes support to perform the termination operations on these same bits and table entries. In addition, L1-caches need to work as write-back or write-through depending on the state of the thread and, for speculative threads, cause a thread stall on a dirty line displacement.

Register-level communication requires a broadcast bus along with one additional read and one additional write port for each register file. In addition, the scoreboard needs to be augmented with $3n$ extra bits per register, where n is the number of processors on chip. Finally, we need a some additional logic replicated for each register to perform various checks as specified in previous sections.

Handling memory dependences requires a MDT along with the related logic that checks for memory dependence violations. We will see later that the MDT can be small.

Overall, enhancing a CMP to execute applications speculatively requires a modest amount of hardware.

5 EVALUATION ENVIRONMENT

The CMP architecture that we propose is modeled as a chip with four 4-issue dynamic superscalar processors and hardware support for speculation (`4x4-issue`). This CMP is compared to conventional 4- and 12-issue dynamic superscalar processors. We assume an aggressive dynamic superscalar core for each of the processors in the CMP and the 4- and 12-issue processors. The core is modeled on the lines of the MIPS R10000 [19]. It can fetch and retire up to n instructions each cycle. It has a large fully-associative instruction window along with integer and floating-point registers for renaming. Some of its characteristics are shown in Table 5. A 2K-entry direct-mapped two-level branch prediction table allows multiple branch predictions to be performed even when there are pending unresolved branches. All instructions take 1 cycle to complete except the following ones: integer multiply and divide take 2 and 8 cycles respectively, floating-point multiply takes 2 cycles, and divide takes 4 and 7 cycles for single and double precision respectively. The 4- and 12-issue processors can have up to 16 and 24 outstanding memory accesses respectively, of which half can be loads. The baseline parameters for the MDT and SS bus are shown in Table 6. We will vary them later.

Issue Width	Number of Functional Units (int/ld-st/fp)	Entries in Instruction Window	Number of Renaming Registers (int/fp)
4	4/2/2	64	64/64
12	12/6/6	200	200/200

Table 5: Characteristics of the dynamic processor core.

Parameter	Value
MDT entries	16K
MDT associativity	8
L1 to MDT latency (Cycles)	2
MDT banks	3
SS bus bandwidth (Words/Cycle)	1
SS bus latency per stage (Cycles)	1

Table 6: Baseline characteristics of the MDT and SS bus. Latencies refer to round trips without contention.

We model the memory sub-system in great detail. Caches are non-blocking and have full load-bypassing enabled. We assume a perfect instruction cache for all our experiments and model only the data caches. Each processor in the CMP has a small private L1-cache and shares a L2-cache with the other processors. This is in contrast to the conventional superscalars, which have a larger L1-cache. Supporting a high-issue processor with a large L1-cache requires several banks and a complex interconnect between the processor and the banks. Consequently, we assume that the round-trip latency to access the L1-cache in the 12-issue superscalar is 2 cycles. For the CMP and the 4-issue superscalar, it takes 1 cycle. Several characteristics of the memory hierarchies are shown in Table 7. In the table, latencies correspond to round trips from the processor without contention. In the simulations, we use the release memory consistency model.

5.1 SIMULATION APPROACH

We use a MINT-based execution-driven simulation environment [13]. MINT [29] generates events by instrumenting binaries and captures both application and library code execution. We have modified

Parameter	CMP	4-Issue Superscalar	12-Issue Superscalar
[L1 , L2] size (Kbytes)	[4x16 , 1024]	[64 , 1024]	[64 , 1024]
[L1 , L2] line size (Bytes)	[32 , 64]	[32 , 64]	[32 , 64]
[L1 , L2] associativity	[2 , 4]	[2 , 4]	[2 , 4]
L1 banks	3	7	7
L1 latency (Cycles)	1	1	2
L2 latency (Cycles)	6	6	6
Memory latency (Cycles)	26	26	26

Table 7: Characteristics of the memory hierarchy. Latencies refer to round trips from the processor without contention.

the MINT front-end to handle MIPS2 binaries as well as instrument basic block boundaries. The binary annotation process starts as soon as the MINT front-end is done with its instrumentation. Later, the application is executed through MINT, which generates basic block and memory events. This information is used by our back-end simulator. The back-end simulator performs a cycle-accurate simulation of very detailed models of the architectures described.

We use MIPS binaries of 4 integer and 4 floating-point applications. The integer applications are *compress*, *ijpeg*, *mpeg* and *eqntott*, while the floating-point ones are *swim*, *tomcatv*, *hydro2d* and *su2cor*. They all belong to the SPEC95 suite except *eqntott*, which is from SPEC92 and *mpeg*, which is a multimedia application from the MediaBench suite [15] that performs image decoding. We use the train set as input for the SPEC95 applications, and the ref set as input for *eqntott*.

The integer applications have more cross-iteration dependences than the floating-point ones. Note, however, that the optimized sequential code generated by the MIPS compiler introduces some dependences by assigning to registers those memory locations that are repeatedly loaded in successive iterations. This eliminates redundant loads to be issued to the L1-cache, but it forces artificial cross-iteration dependences to appear. This applies to both integer and floating-point applications.

Note that, for the high-issue dynamic superscalars simulated, the native compiler cannot generate an optimally-scheduled binary. However, the large instruction window in the simulated superscalars should offset this to a large extent. Furthermore, we also reschedule the execution trace before feeding it to the simulator. The rescheduler performs a resource-constrained scheduling of instructions. Since it gathers several loop iterations, it enables aggressive loop-unrolling. It is this optimized window of instructions that is finally passed to the processor simulator [13].

6 EVALUATION

To evaluate the proposed architecture, we compare the performance of our `4x4-issue` CMP to that of an aggressive superscalar (Section 6.1). Later, we assess the requirements imposed by our speculation hardware (Sections 6.2 and 6.3).

6.1 CMP VERSUS SUPERSCALAR

To put the comparison in the proper perspective, we consider two important issues in any processor design, namely area and timing. We first consider the area. Typically, the instruction window to enable the dynamic issue of instructions requires a large die area. For example, the PA-8000, which is a 4-issue superscalar, devotes 20% of the die area solely to the instruction window. In general, the area requirements increase quadratically with issue width. In addition, increasing the issue width typically requires an increase in the number of ports in the register file. Alternatively, it may involve replicating the register file as in the Alpha 21264. Finally, the number of data-bypass paths between the functional units and the register file increases quadratically with issue width. Overall, support for dynamic issue leads to a near-quadratic increase in die space with issue width and, typically, consumes up to 30-40% of the total die area.

From this discussion, it may be shown that the area required by a `4x4-issue` CMP is only slightly larger than that of a 12-issue superscalar. This is why we consider this superscalar in our comparison to CMP. Note that the CMP requires extra hardware for speculation support. However, the overhead for register communication is quite modest. Only the 16K-entry MDT is likely to occupy a substantial chip area. However, we will show in Section 6.3 that the MDT needs to have relatively few entries and, therefore, can be placed on chip without a significant impact on the die area.

We now consider the timing. Palacharla et al [22] have argued that the register bypass network may be an important factor in determining the cycle time in future high-issue processors. If we use their assumptions, the `4x4-issue` CMP would have a tremendous cycle time advantage over the 12-issue dynamic superscalar when implemented in technologies that are denser than 0.18 μm.

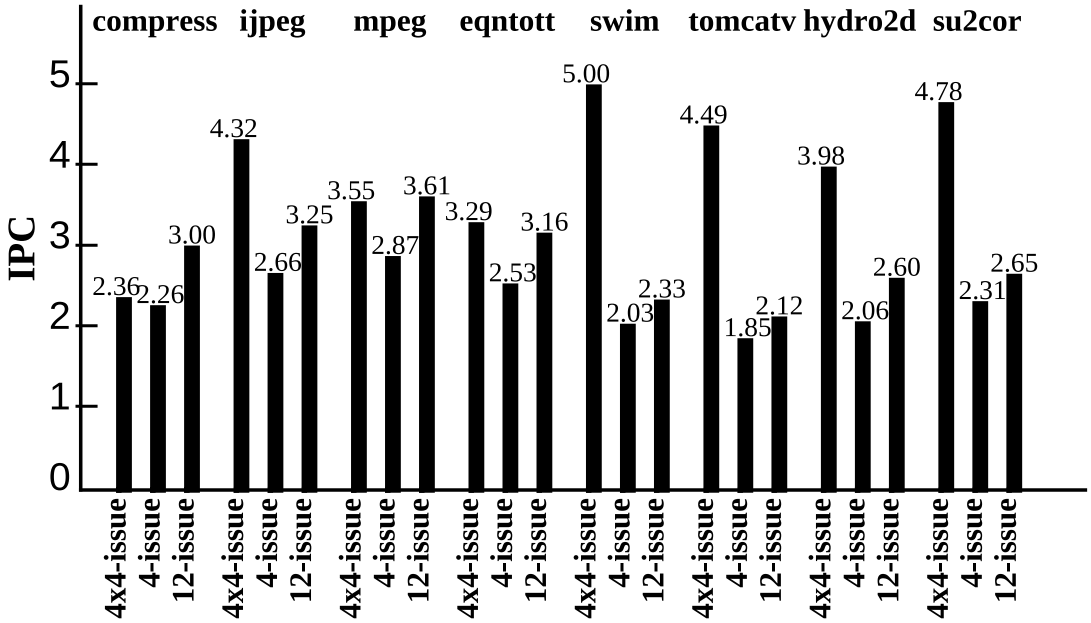

Figure 7: Comparing the IPC for the applications under the 4x4-issue CMP and 4- and 12-issue conventional superscalars. No cycle time differences are considered.

Finally, we consider performance independently of area or cycle time considerations. Figure 7 shows the IPC for the eight applications on the `4x4-issue` CMP and the 12-issue superscalar. For reference purposes, we also show a 4-issue superscalar. Note that, because we look at IPC, this data does not take into consideration any disparity in cycle time among the different architectures. For the CMP, Figure 8 breaks down the number of instructions issued based on which of the 4 processors in the CMP issued them. Note that a very uneven distribution of work among the different processors is an indication that we are unable to extract a significant amount of parallelism from the application when restricting ourselves to inner loops.

To understand the two charts, we examine the floating-point applications first. From Figure 8, we see that their execution results in the four processors of the CMP being busy most of the time. These

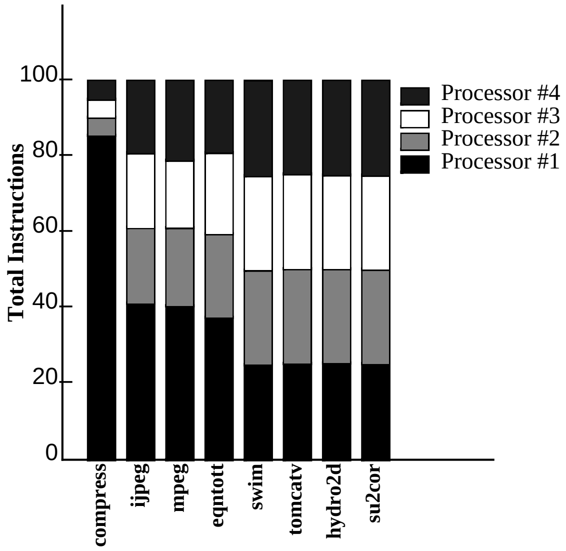

Figure 8: Fraction of instructions issued by each of the four processors in the CMP.

applications are fully loop-based, and most of the loops have few or no loop-carried dependences. The CMP is able to exploit the parallelism in these applications to a greater degree than the 12-issue superscalar. Each processor in the CMP executes an iteration and, most of the time, can independently issue instructions without being affected by dependences with other threads. In the 12-issue superscalar, instead, the centralized instruction window is often clogged by instructions that are either data dependent on long-latency floating-point operations or are waiting on cache misses. Overall, as shown in Figure 7, the IPC of the `4x4-issue` CMP is on average nearly twice that of the 12-issue superscalar. The 4-issue superscalar has a lower IPC than the 12-issue one.

The integer applications behave differently. According to Figure 8, one single thread issues a larger fraction of all the instructions in the application. In fact, in *compress*, most of the time there is only one active processor. *Compress* spends most of the time inside a big loop with a small inner-loop and an otherwise complicated control flow. Since we exploit inner loop parallelism only, there is little parallelism to be exploited. In this environment, the ability of the 12-issue superscalar to exploit ILP in the serial sections gives it an advantage. Overall, for the combined integer applications, the difference in IPC between the `4x4-issue` CMP and the 12-issue superscalar is small. Figure 7 shows that, on average, the `4x4-issue` CMP has about the same IPC as the 12-issue superscalar. The 4-issue superscalar has a lower IPC.

Our approach of exploiting parallelism in binaries without recompilation limits the performance of the CMP in integer applications. Indeed, it has been shown that there is significant scope for improvement even for such inherently-sequential applications, when the source code is recompiled [27]. Despite this limitation, for *ijpeg*, *mpeg* and *eqntott*, the CMP has a comparable or even higher IPC than the 12-issue superscalar.

We stress that this comparison is solely based on IPC values and does not take cycle time into consideration. If we conservatively assume a modest 25% shorter cycle time for the `4x4-issue` CMP over the 12-issue superscalar [22], the `4x4-issue` CMP performs much better. Overall, therefore, we conclude that a CMP with speculation support delivers high performance even without the need for source recompilation.

6.2 EVALUATING THE SS BUS

We now examine the bandwidth requirements and the latency effects of the SS bus. Figure 9 shows the execution time of the applications when the bus bandwidth is increased from 1 to 3 words per cycle. The execution time is normalized to the baseline system of Section 5, which has a bandwidth of 1 word per cycle. From the figure, we can see that there is little performance to be gained with higher bandwidth.

Figure 10 looks at the effect of decreasing the latency between end-points of the SS bus. The execution time is normalized to the baseline system of Section 5, which has a latency of 3 cycles between end-points. As shown in the figure, reducing the latency has only a very modest effect on performance. The maximum change is only 3% when the latency decreases from 3 to 1 cycle. From the above two results, we conclude that adding more resources to our currently-inexpensive register communication mechanism would not improve the performance much.

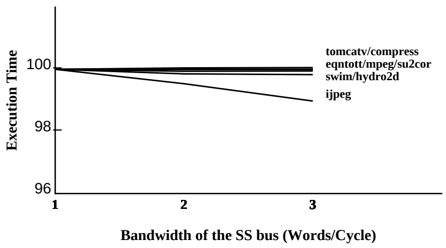

Figure 9: Effect of the SS bus bandwidth.

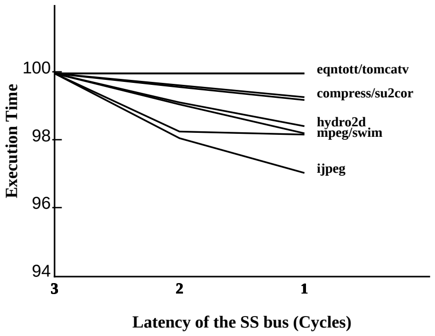

Figure 10: Effect of the latency between end-points of the SS bus.

6.3 EVALUATING THE MDT

Since a 16K-entry MDT occupies a substantial die area, it may have to located alongside the L2-cache. This, in fact, would have little impact on performance. We reach this conclusion by simulating scenarios of increased L1 to MDT latency with correspondingly-reduced MDT to L2 latency. Figure 11 shows how the execution time is affected when the round-trip latency from L1 to MDT changes from 1 to 5 cycles. In all data points of the chart, the round-trip latency from L1 to L2 remains fixed at 5 cycles. In the figure, the execution time is normalized to a system with a L1 to MDT latency of 5 cycles, which means that the MDT is located alongside the L2-cache. The baseline system of Section 5 corresponds to the point labeled *2* in the X-axis of Figure 11. Overall, Figure 11 shows that there is little change in performance for all the applications as the MDT is moved to different positions in between the L1- and L2-caches.

Unfortunately, configuring the 16K-entry MDT off-chip would increase the pin requirements of the chip. Consequently, we would like to explore a reduction in the MDT size so that it can be easily configured on chip. Figure 12 studies the variation of the performance when the number of MDT entries varies from just 8 to the baseline 16K. The figure is normalized to the system with 8 entries. The figure shows that bigger MDTs enhance performance. However, the performance does not keep improving for MDTs bigger than 64 entries in most applications. The only exception is *swim*, where the working set

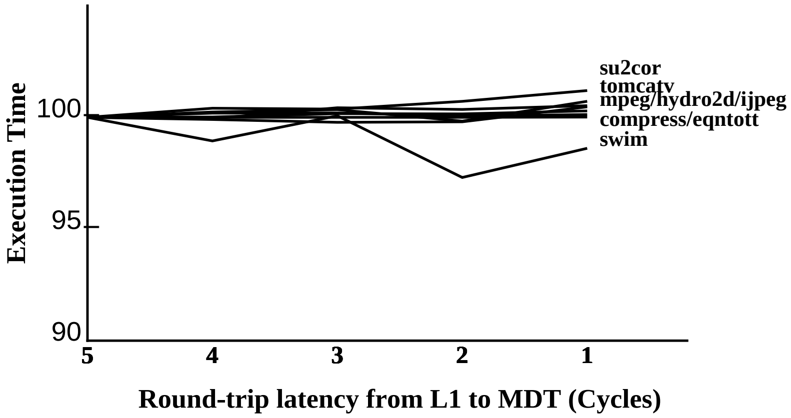

Figure 11: Effect of the round-trip latency from L1 to MDT. The round-trip latency from L1 to L2 remains fixed at 5 cycles.

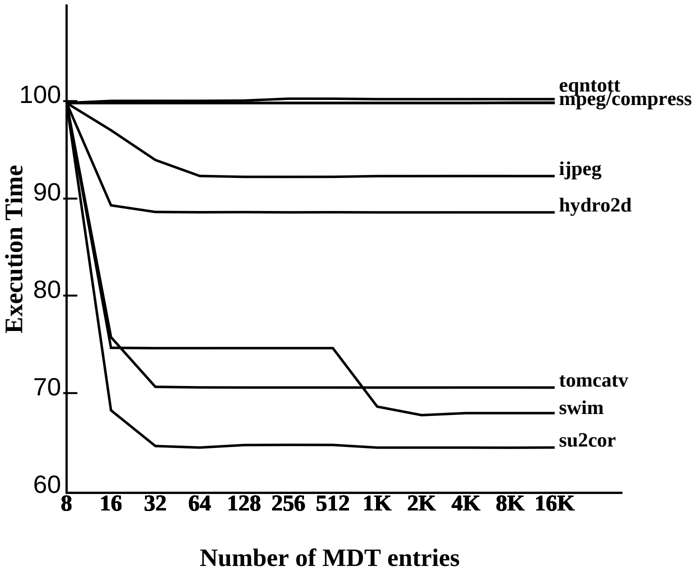

Figure 12: Effect of the MDT size.

is large enough to require a 1K-entry MDT. Overall, with such minimal space requirements for most applications, the MDT disambiguation hardware can be easily configured on chip.

7 CONCLUSIONS

The chip-multiprocessor (CMP) approach is a promising design for exploiting the ever-increasing on-chip transistor count. It is an ideal platform to run the new generation of multithreaded applications as well as multiprogrammed workloads. Moreover, it can also handle sequential workloads when hardware support for speculation is provided. However, current proposals either required recompilation of the sequential application or devoted a large amount of hardware to speculative execution, thereby wasting resources when running fully-parallel applications. In this paper, we have presented a CMP architecture that is generic enough and has modest hardware support to execute sequential binaries in a most cost-effective manner. We have discussed how we extract threads out of sequential binaries and presented hardware support that enables communication through both registers and memory. Overall, our evaluation shows that this architecture delivers high performance for sequential binaries with modest hardware requirements.

Acknowledgments

We thank the referees and the members of the I-ACOMA group for their valuable feedback. 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.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[2] S. Breach, T. N. Vijaykumar, and G. Sohi. The Anatomy of the Register File in a Multiscalar Processor. In *27th International Symposium on Microarchitecture (MICRO-27)*, pages 181–190, December 1994.

[3] R. Colwell and R. Steck. A 0.6μm BiCMOS Processor with Dynamic Execution. In *ISSCC Proceedings*, 1995.

[4] The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Microprocessor Forum, October 1996.

[5] 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.

[6] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine Multicomputer. In *28th International Symposium on Computer Microarchitecture (MICRO-28)*, pages 146–156, November 1995.

[7] M. Franklin and G. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[8] S. Gopal, T. N. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[9] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[10] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[12] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1990.

[13] V. Krishnan and J. Torrellas. A Direct Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *PACT '98*, pages 286–293, October 1998.

[14] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *12th International Conference on Supercomputing (ICS)*, July 1998.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems . In *30th International Symposium on Microarchitecture (MICRO-30)*, pages 330–335, December 1997.

[16] D. Levitan, T. Thomas, and P. Tu. The PowerPC 620 Microprocessor. In *Spring CompCon Proceedings*, 1995.

[17] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *12th International Conference on Supercomputing (ICS)*, July 1998.

[18] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, September 1997.

[19] MIPS Technologies, Inc. *R10000 Microprocessor Chipset, Product Overview*, 1994.

[20] A. Moshovos, S. Breach, T. N. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[21] 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.

[22] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[23] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *29th International Symposium on Microarchitecture (MICRO-29)*, pages 24–34, December 1996.

[24] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *30th International Symposium on Microarchitecture (MICRO-30)*, pages 138–148, December 1997.

[25] J. Smith and S. Vajapeyam. Trace Processors: Moving to Fourth Generation Microarchitectures. *IEEE Computer*, 30(9):68–74, September 1997.

[26] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[27] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[28] J. Tsai and P. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *PACT '96*, pages 35–46, October 1996.

[29] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

[30] T. N. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *31st International Symposium on Microarchitecture (MICRO-31)*, December 1998.