Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor[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

Abstract

Chip-multiprocessors (CMP) are a promising approach for exploiting the increasing transistor count on a chip. To allow sequential applications to be executed on this architecture, current proposals incorporate hardware support to exploit speculative parallelism. However, these proposals either require re-compilation of the source program or use substantial hardware that tailors the architecture for speculative execution, thereby resulting in wasted resources when running parallel applications.

In this paper, we present a CMP architecture that has hardware and software support for speculative execution of sequential binaries without the need for source re-compilation. The software support enables identification of threads from sequential binaries, while a modest amount of hardware allows register-level communication and enforces true inter-thread memory dependences. We evaluate this architecture and show that it is able to deliver high performance.

1 Introduction

Current technology permits the integration of multiple processing units on a single chip. Unlike a conventional superscalar design, this approach avoids resource centralization and long interconnects, thereby enabling a faster processor clock. Moreover, it permits multiple control flows or threads in the application to be executed concurrently. Though this architecture would be an ideal platform for the new generation of multithreaded applications, it may not be good for applications that are either sequential or cannot be parallelized effectively. To alleviate this problem, additional hardware may be used that would enable speculative parallelism to be exploited from such sequential applications. In this mode of execution, the threads do not need to be fully independent; they may have data dependences with each other.

There have been two approaches for configuring multiple processing units on a chip. In one approach, the architecture is fully geared towards exploiting speculative parallelism from sequential applications. Typical examples are the Trace [9, 10] and Multiscalar [11] processors. Indeed, these processors can 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 [8] 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, threads are identified statically by a compiler. Register values are forwarded from one processor to another with the aid of a ring structure, while recovery from mis-speculation is achieved by maintaining two copies of the registers, along with a set of register masks, in each processing unit [1]. Overall, both of these processors have sufficient hardware support that tailors the architecture for speculative execution, thereby enabling them to achieve high performance on existing sequential binaries without the need for re-compilation. A direct consequence of this, however, is that a large amount of hardware may remain un-utilized when running a parallel application or a multiprogrammed workload.

The chip-multiprocessor (CMP) [6, 12, 13], on the other hand, is generic enough and has minimal support for speculative execution. Current proposals support a restricted communication mechanism between processors, which can occur only through memory. Such limited hardware may be sufficient when programs are compiled using a compiler that is aware of the speculation hardware [12]. However, the need to re-compile the source is a handicap, especially when the source is unavailable.

In this paper, we present a CMP architecture that is able to leverage the large number of transistors that can be integrated on chip to speed up the execution of sequential binaries in a very cost-effective manner. We have designed a binary annotation tool that extracts multiple threads from sequential binaries to execute on the CMP. We also use modest hardware support to execute speculatively the code generated by the binary annotator. This hardware allows communication both through memory and registers, without the need for additional register sets. Overall, we propose an architecture that adds little hardware to a generic CMP, while it is able to handle sequential binaries quite effectively.

The paper is organized as follows: Section 2 describes our software support for identifying threads in a sequential binary. Section 3 discusses the hardware support required for speculative execution; Section 4 describes the evaluation environment; Section 5 performs the evaluation; and finally, Section 6 concludes the paper.

2 Software Support: Binary Annotator

We have developed a binary annotator that 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 3.

[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.

Currently, we limit the threads to inner loop iterations, where the loop body may have recursive function calls. This will be extended soon to cover other code sections.

First, we mark the entry and exit points of these loops. 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.

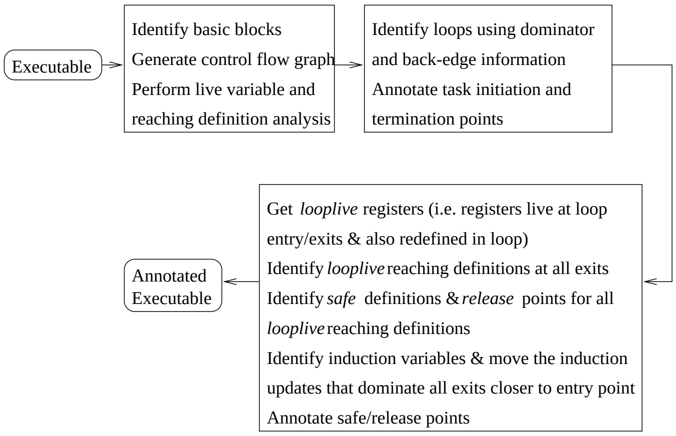

Figure 1: Binary annotation process.

The steps involved in the annotation process are illustrated in Figure 1. The approach that we use is similar to that of Multiscalar [11] except that we operate on the binary instead of 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 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*

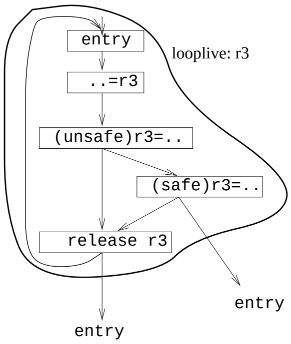

Figure 2: Safe definitions and release points.

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 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 incorporating these 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.

3 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 3.1).

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 separate speculative updates from non-speculative ones 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 would 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 [2]. 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) [3], which makes use of a snoopy-bus to maintain data consistency among different processors. Our work differs from SVC in that we use a decentralized approach similar to a directory-based protocol. We call our approach the *Memory Disambiguation Table (MDT)* (Section 3.2). In the following, we present the SS and the MDT in turn. All our discussion assumes a 4-processor CMP.

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 is the one 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. Threads commit in order.

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.

3.1 Synchronizing Scoreboard

The synchronizing scoreboard is used by threads to synchronize and communicate register values. It is a per-processor conventional scoreboard augmented with additional bits. In addition, to allow register values to be communicated, we use a simple broadcast bus. This bus, which we call the *SS Bus*, has a limited bandwidth of 1 word/cycle, a low latency of up to 3 cycles, and one read and one write port for each processor. The overall hardware setup is shown in Figure 3.

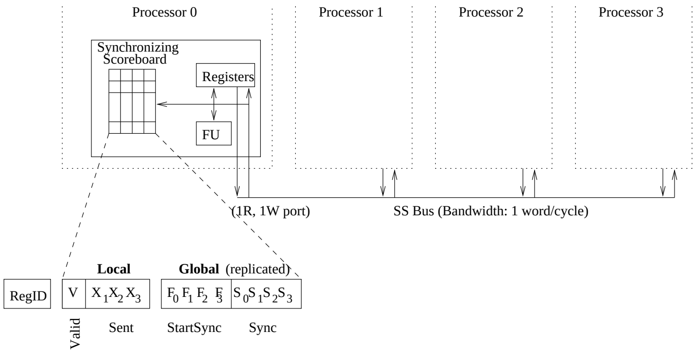

Figure 3: Hardware support for register communication.

Each SS has one entry per register. Figure 3 shows the different fields for one entry. The fields are grouped into *local* and *global*. The latter are replicated and kept coherent across the several SS. 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.

RegID	StartSync $F_0 F_1 F_2 F_3$	Sync $S_0 S_1 S_2 S_3$
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 a given register, the S_i bit means 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 performs a safe definition for that register or executes the release instruction for that register (Section 2). When this occurs, the thread also writes the register value on the SS bus, thereby allowing other processors to update their values if needed. At that point, the register is safe to use by successor threads.

The F bit is set to the initial 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 (to 0 or 1) in the SS of all processors. From then on, the F bit remains unchanged throughout the execution of the thread.

Each processor has an additional *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 speculatively parallel section of the code is reached, the processors that were idle in the preceding serial section start with their V bits equal 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 the same parallel section, a processor can reuse registers across threads. When a processor initiates a new thread, namely speculative successor 3, 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. We now look at how registers are communicated between processors.

3.1.1 Register Communication

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 own V bit and the F bits of the threads between the producer and itself. If all these bits are zero, the successor thread loads the register and sets its V bit to 1.

It is possible, however, that the consumer thread is not running when the producer thread generates the register. We could allow the values to be stored, by using a buffered communication mechanism, rather than using a simple broadcast bus. However, this would require further hardware support in the form of duplicate register sets in each processor to enable recovery from squashes [1]. Alternatively, a global register set may be maintained to store the values [9], but at the cost of maintaining a centralized structure. In our approach, we add minimal hardware to support the consumer-initiated form of communication. Specifically, the SS has simple logic that allows a consumer thread to identify the correct producer and get the register value. The logic works as follows. The consumer thread first checks its V bit. 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 threads, and that thread 3 tries to read a register. In that case, the register will be available to thread 3 if:

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

This check incurs a delay of an AND-OR logic, with the number of inputs to the gate dependent on the number of supported threads. 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 SS access has been performed by speculative successor 3. Therefore, according to Table 1, 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 is performed by thread 3. In that example, thread 3 is speculative successor thread 2. Consequently, it uses *ThreadMask* 0111 from Table 1. This means that it is examining only 2 predecessors of thread 3. The resulting formula will be:

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

Overall, the complete logic involved in determining whether a register is available or not is shown in Figure 4. If the register is available, the reader thread can get the value from the closest predecessor thread whose S bit is clear but F bit is set. If all the bits are clear, the non-speculative thread can provide the value. The transfer of 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 reply messages can take 1-3 cycles each, depending on the producer-consumer distance, plus the contention for the SS bus. The latency and bandwidth requirements of the bus to achieve optimal performance is one topic of our evaluations in Section 5.

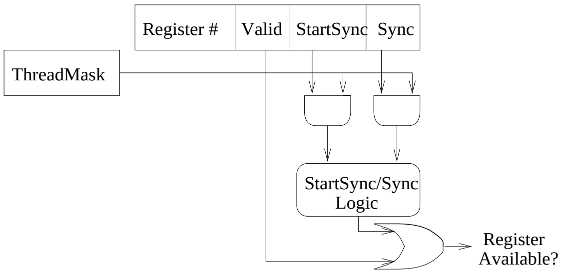

Figure 4: Logic to check register availability. This implements the function in equations (1) and (2).

Finally, the private X bit is used to avoid the undesirable state where all copies of the register become invalid. This could have occurred when there is a delay in the consumer thread requesting the value. More details may be found in [4]. Overall, the extra overhead per register is $3n$ bits, where n is the number of processors on chip. This is indeed a modest amount of storage overhead when compared to replicating the register sets in each processor [11] or using a centralized global register file [9].

3.2 Memory Disambiguation

Unlike register dependences, we do not identify memory dependences in the binary annotation phase and, therefore, assign the full responsibility to hardware. Each processor in the CMP has a private L1 cache. The L2 cache is shared. During speculative execution, speculative threads are in a restricted write-back mode in that they cannot update the L2 cache. When a dirty line needs replacement, the thread stalls. Unlike the speculative threads, the non-speculative thread is allowed to update the L2 cache, since its store operations are safe. When a speculative thread acquires non-speculative status, dirty lines are allowed to be displaced from the L1 cache. Note that the non-speculative thread cannot be committed and a new thread initiated on the processor, if there are any dirty lines present in the processor's L1-cache. Thus, allowing the non-speculative thread to work in a write-through mode reduces the number of dirty lines that may have to be flushed towards the end of the thread's execution.

Identifying memory dependence violations is achieved with the help of a memory disambiguation table (MDT). The MDT is analogous 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 MDT is examined later in this section.

To enable speculative execution, each private L1-cache line is augmented to have the following bits:

Flush	Invalid	Dirty	SafeWrite	SafeRead

Table 3: L1-cache line bits.

The *Invalid (I)* and *Dirty (D)* bit are used as in a conventional cache. Each processor, when performing a load or store operation, may have to inform the MDT, which tracks memory dependence violations. The *SafeWrite (SW)* and *SafeRead (SR)* bits allow the processor to perform most of the load and store operations without informing the MDT, while the *Flush (F)* bit allows cache lines in a processor to remain valid across multiple thread initiations.

The *SafeWrite (SW)* bit, if set, permits a thread to perform writes to cached dirty data without informing the MDT. When a thread performs a store to a line for the first time, it sets both the D as well as the SW bit, and also sends a message to the MDT. Subsequent stores to a dirty data 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 that processor. It is also cleared when a successor thread reads that same location. The read by a successor is forwarded to the processor by the MDT. Thus, if the thread stores again to the same dirty location but the SW bit is not set, it signifies that a successor thread has loaded this location prematurely. The processor sets the SW bit and also sends a message to the MDT. This enables the MDT to determine that a speculative thread has performed a load prematurely.

The *SafeRead (SR)* bit, if set, allows the thread to perform load operations from the cache without informing the MDT. This bit is set when the thread reads the location for the first time. In addition, the MDT is notified about the load. Future loads from the same location can be performed without informing the MDT as long as the SR bit is set. The SR bit is cleared when a new thread is initiated on that processor. This bit is used only when the thread is speculative, since loads performed by non-speculative threads are safe.

The *Flush (F)* bit is used to flush stale locations, before a new thread is initiated in the processor. When a thread stores to a location, any lines with the same address have to be invalidated in all its successors. As for the predecessors, information needs to be maintained to denote that the location will become stale after the current thread completes. The F bit identifies those private L1-cache lines that need to be invalidated after the current thread completes. When a thread is initiated on a processor, the new value of the *Invalid* bit will be: $I_{new} = I_{old} + F$. The F bit is cleared after this operation. As explained later, this is done with the aid of the MDT that keeps track of the loads and stores issued by each thread for the cached locations.

Maintaining these bits on a cache-line level sometimes results in false sharing and, consequently, leads to unwanted squashing of threads. So, we maintain information at a word level, except the *Dirty* bit, which is maintained at the line level. More details of the protocol may be found in [4].

3.2.1 Memory Disambiguation Table (MDT)

The disambiguating mechanism is performed with the help of the MDT. The MDT maintains entries on a cache line basis. As with the cache lines, we maintain the information on a word basis. For each word, there is a *Load (L)* and a *Store (S)* bit for each processor. 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. The bits are also cleared after a thread is squashed, but before a new thread is initiated on the processor. Table 4 shows a MDT for a 4-processor CMP.

The MDT may be incorporated into the L2 cache. Thus, the MDT would be similar to a directory that keeps track of sharers in a conventional shared-memory multiprocessor. Later, in our evaluations, we show that the size of the MDT that is required is quite small. Consequently, it may be even configured on-chip. But first, we explain how the MDT works when it receives a load or store message from a processor.

Valid Bit	Address Tag	Load Bits (Word 0) $L_0 L_1 L_2 L_3$	Store Bits (Word 0) $S_0 S_1 S_2 S_3$
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 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 predecessor thread has speculatively updated the location. Therefore, the store bits are checked to determine the id of any thread that has updated the location. 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 $S_0 S_1 S_2$ 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. In the above table, when thread 2 reads the first word corresponding to the address tag 0x1234, the request is forwarded by the MDT to processor 1 so that its L1-cache can supply data for the word. The remainder of the line is supplied from the L2-cache, which is merged with the forwarded data by the receiving processor. The forwarded request to thread 1 also clears the SW bit in its cache line. As explained in the next section, if thread 1 performs a store to the same line, it sends a message to the MDT informing it about the store operation, which would then squash thread 2. Finally, if no S bits of any predecessor threads are set, the load proceeds in the conventional manner to the L2-cache, and finally to memory.

For a given thread, only the first unsafe load request to the address reaches the MDT. Subsequent loads to the same address or a load to the address to which the thread has already performed a store do not reach the MDT.

Store Operation

Unlike loads, stores can result in the squashing of threads when a successor speculative thread prematurely loaded a value for an address. Successor, therefore, rather than predecessor threads are tested on a store operation. As in the load operation, we check the MDT for an entry corresponding to the address being updated. This is done by both non-speculative and speculative threads. If the entry is not found, speculative threads create a new entry with their S bit set.

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. 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 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 and all its successor threads are squashed and restarted. In all cases, a message to set the flush (F) bit for the address is sent to all predecessors that have their S or the L bit set. This would result in their corresponding cache lines to remain valid only during the course of execution of their current task. Also, an invalidation message is sent to all successors up to (but not including) one whose S bit is set. This is necessary as a successor thread may use a old value from a valid cache line when the thread was initiated. Contrast this to a conventional SMP, where cache lines in all processors are invalidated on a write.

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 free entries are available, the thread stalls until one becomes available. Since the non-speculative thread can continue to perform load and store operations, its execution will not be affected. 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.

3.2.2 What Happens When a New Thread Is Started or Threads Are Squashed?

At the point of thread initiation, all the 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 initiated on those processors and all the actions described above will be performed. The only issue is that there may be cache lines that may remain valid and whose value was forwarded from another thread. If that thread was squashed, we need to invalidate the lines that were forwarded. For simplicity, we do not record what thread is the source of what forwarded line. We simply identify all the forwarded data with a *Forward* (FD) bit. This bit is set when the MDT informs the processor performing a load that the data being forwarded from another processor's L1-cache. Thus, for squashed threads, not only the F bit but also the FD bit is used for deciding the invalidity of the cache line: $I_{new} = I_{old} + F + FD$.

Overall, the additional overhead added to each cache line is modest. The coherence protocol is similar to a conventional directory-based scheme with sub-blocks. Finally, having a separate dependence-tracking module (MDT) leaves room for further improvements, such as memory dependence prediction [5], thereby disallowing premature loads from occurring.

4 Evaluation Environment

For our representative CMP, we consider a chip with four 4-issue dynamic superscalar processors (`4x4-issue`). This CMP, with hardware support for speculation, is compared to conventional 12-issue and 16-issue dynamic superscalar processors. We assume an aggressive dynamic superscalar core for the CMP and the 12-issue/16-issue processors. The core is modeled on the lines of the R10K. 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. The actual numbers are shown in Table 5. A 2K-entry direct-mapped 2-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: integer multiply and divide take 2 and 8 cycles respectively; floating-point multiply takes 2 cycles, while divide takes 4 (single precision) and 7 (double precision) cycles. The 4-, 12- and 16-issue processors can have up to 16, 24 and 32 outstanding memory accesses, of which half can be loads. The default 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
16	16/8/8	256	256/256

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)	1
SS bus latency (Cycles)	1

Table 6: Characteristics of the MDT and SS bus.

Finally, we model the memory sub-system in great detail. Caches are non-blocking with full load-bypassing enabled. We assume a perfect I-cache for all our experiments and model only the D-cache. Typically, each processor in the CMP would have a small private L1-cache and a shared L2-cache. This is in contrast to the conventional superscalar that has a larger L1-cache. Since supporting a higher-issue processor with a large L1 cache requires more banks and a complex interconnect between the processor and the banks, we assume that the latency of access to the L1-cache to be 2 cycles for the 12/16-issue superscalars as opposed to 1 cycle for the CMP. The characteristics of the memory hierarchies are shown in Table 7.

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

Table 7: Characteristics of the memory hierarchy. Latencies refer to round trips.

Simulation Approach

Our simulation environment is built on a modified MINT [14] execution-driven simulation environment. MINT captures both application and library code execution and generates events by instrumenting binaries. Our back-end simulator is extremely detailed and performs a cycle-accurate simulation of the architectures and hardware support for speculation described.

Finally, we use MIPS binaries for 5 integer (*compress*, *li*, *ijpeg*, *mpeg* and *eqntott*) and 5 floating-point (*swim*, *tomcatv*, *hydro2d*, *su2cor* and *mgrid*) applications. All are from the SPEC95 suite except *eqntott*, which is from SPEC92 and *mpeg*, which is a multimedia application used for image decoding. We use the train set as input for the SPEC95 applications and the ref input for *eqntott*.

5 Evaluation

We compare our `4x4-issue` CMP to the aggressive superscalars in Section 5.1, and then evaluate the requirements imposed by the speculation hardware in Sections 5.2 and 5.3.

5.1 CMP vs Superscalars

To put the comparison in the proper perspective, we consider two issues whose complexity dictates any processor design, namely *area* and *timing*. We first consider the area complexity. Typically, the instruction window to enable dynamic issue requires a large die area that increases quadratically with issue width. In fact, the PA-8000, which is a 4-issue superscalar, devotes 20% of the die area solely for the instruction window. In addition, increasing the issue width requires an increase in the number of ports on the register file. This may be achieved by 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 and, typically, consumes up to 30-40% of the total die area. As a result, the area required by a `4x4-issue` CMP would only be slightly larger than that of a 12-issue superscalar, while being much less than that of a 16-issue superscalar. This is why we consider the above two superscalars in our comparisons 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. While the MDT can be moved off-chip at the cost of an increased access latency, we will show in Section 5.3 that the space requirements imposed by the MDT are in fact small enough that it can be placed on-chip without a significant impact on the die area.

We now consider the timing complexity. Palacharla and Jouppi [7] have argued that the register bypass network will dominate the cycle time of future high-issue processors. In fact, they have shown that an 8-issue processor would have almost twice the cycle time of a 4-issue processor with 0.18μ technology. Based on their observations, the `4x4-issue` CMP would have a tremendous cycle time advantage over the 12-issue and 16-issue dynamic superscalars.

Figure 5 shows the IPC for the ten applications on the CMP and superscalars. Note that, because we look at IPC, this data does not take into consideration any disparity in cycle time among the different architectures. Figure 6 breaks down the number of instructions issued based on which of the 4 processors in the CMP issued them. Note that an even distribution of work among the different processors implies that our binary annotator is able to identify a large amount of speculative parallelism in the application.

To understand the two charts, we examine the floating-point applications first. From Figure 6, we see that they have much speculative parallelism. This is because these applications are fully loop-based, with most of them having few or no loop-carried dependences. Consequently, the CMP is able to exploit the parallelism in these applications to a much greater degree than the conventional superscalars. Each processor in the CMP executes an iteration and, most of the

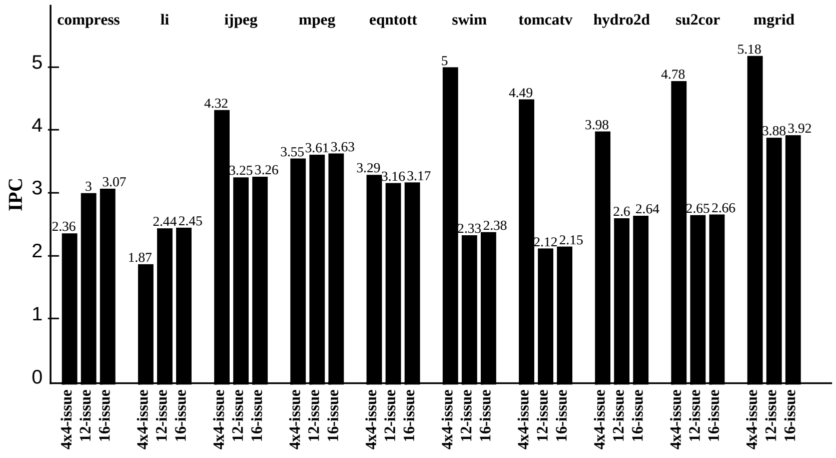

Figure 5: Comparing the IPC for the applications under the 4x4-issue CMP and a 12-issue and 16-issue conventional superscalar. No cycle time differences are considered.

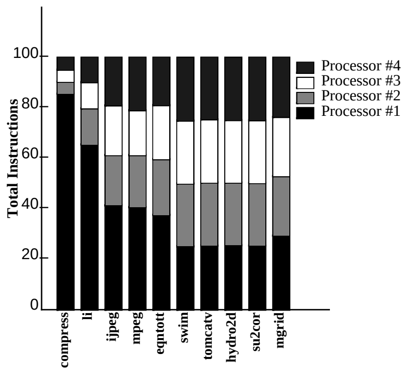

Figure 6: Fraction of instructions issued by each of the four processors in the CMP.

time, can independently issue instructions without being affected by dependences with other threads. This is not the case in the conventional superscalars, where the centralized instruction window is often clogged by instructions that are either data dependent on long-latency floating-point operations or are waiting on a cache miss. Overall, as shown in Figure 5, the IPC of the `4x4-issue` CMP is 1.8 times that of the superscalars.

The integer applications behave differently. According to Figure 6, the amount of speculative parallelism extracted by the binary annotator is much lower. In fact, in *compress* and *li*, most of the time there is only one active processor. *Compress*, for example, 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 to be exploited. In this environment, the ability of the dynamic superscalars to exploit ILP in the serial sections gives them an advantage. As a result, Figure 5 shows that they have around 25% higher IPCs for these two applications.

Our approach of exploiting parallelism in binaries without re-compilation limits the performance of the CMP in these applications. Indeed, it has been shown that there is a large scope for improvement even for such inherently sequential applications, when the source code is recompiled with appropriate speculation support [12]. For the remaining three applications (*ijpeg*, *mpeg* and *eqntott*) which have relatively smaller sequential sections, the CMP has a comparable or a higher IPC than the conventional superscalars.

We stress that we have made our comparisons solely based on the IPC values, without taking cycle time into consideration. Even assuming a modest 25% shorter cycle time for the CMP, it would perform much better than the superscalars. Overall, therefore, we see that the CMP with speculation support delivers high performance even without the need for source re-compilation.

5.2 Evaluating the SS Bus

In this section, we look at the bandwidth requirements as well as the latency effects of the SS bus. Figure 7 shows the execution time of the applications when the bus bandwidth is increased from 1 to 3 registers/cycle. The execution time is normalized to that for a bandwidth of 1 register/cycle. From the figure, we can see that there is little performance gain with higher bandwidth. Figure 8 looks at the effect of higher bus latencies. The execution time is normalized to that for a latency is 3 cycles. As shown in the figure, changing the latency has only a very modest effect on performance, with a maximum change of only 4% when the latency varies from 3 to 1 cycle. From the above two results we can, therefore, conclude that our register communication mechanism does not place a great demand on hardware resources.

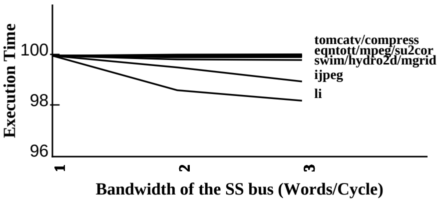

Figure 7: Effect of the SS bus bandwidth.

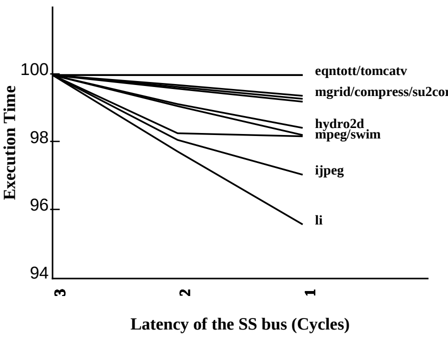

Figure 8: Effect of the SS bus latency.

5.3 Evaluating the MDT

Since a 16K-entry MDT would occupy a substantial amount of die area, it would have to be moved off-chip and located alongside the L2 cache. This, in fact, would have little impact on performance. To see why, we simulate this scenario by increasing the L1 to MDT latency while correspondingly reducing the MDT to L2 latency. Figure 9 shows how the execution time is affected when the L1 to MDT latency changes from 1 to 5 cycles. Here, the execution time is normalized to that for a latency of 5 cycles. We can see that there is little change in performance for all the applications.

Unfortunately, configuring the 16K-entry MDT off-chip would increase the pin-requirements of the chip. Consequently, we would like to see a reduction in the MDT size so that it can be configured on chip. Figure 10 studies the effect on performance when the number of MDT entries varies from just 8 to 16K. Surprisingly, the performance saturates with a small 64-entry MDT for most applications. *swim* is the only exception, where the working set is large enough to require a 1K-entry MDT. Overall, with such minimal space requirement for most applications, the MDT disambiguation hardware can be easily configured on chip.

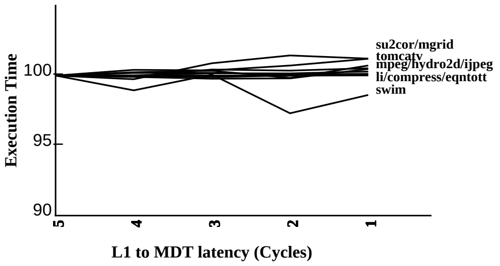

Figure 9: Effect of the L1 to MDT latency.

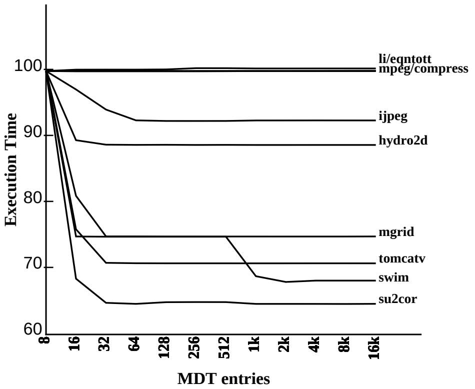

Figure 10: Effect of the MDT size.

6 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 require re-compilation of the sequential application or devote a large amount of hardware to speculative execution, thereby wasting resources when running 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 speculative hardware support that enables communication both through registers and memory. Our evaluation shows that this architecture delivers high performance for sequential binaries with very modest hardware requirements.

Acknowledgments

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

References

[1] 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.

[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] 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.

[4] V. Krishnan. *A Multithreaded Architecture for Enhancing the Performance of Sequential and Parallel Applications.* PhD thesis, University of Illinois at Urbana-Champaign, May 1998.

[5] 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.

[6] 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.

[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] 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.

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

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

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

[12] 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.

[13] 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.

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