# SPECULATIVE MULTITHREADING ARCHITECTURES

BY

VENKATA S. KRISHNAN

B.Tech., Indian Institute of Technology, Madras, 1990
M.S., University of Arizona, Tucson, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

SPECULATIVE MULTITHREADING ARCHITECTURES

Venkata S. Krishnan, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1999
Josep Torrellas, Advisor

With the conventional superscalar approach delivering diminishing returns, alternate designs that make optimal use of the increasing chip densities are actively being explored. Unlike the conventional superscalar that expends all its resources in exploiting instruction-level parallelism (ILP), there has been much emphasis on architectures that can also exploit thread-level parallelism from the application. Chip-multiprocessor architectures (CMP) are one promising approach in this direction that can also better exploit the increasing transistor count on a chip. In these architectures, speculation may be employed to execute applications that are either sequential or cannot be parallelized efficiently. Unfortunately, current approaches either use limited hardware support that permit a restricted communication mechanism between processors, which can occur only through memory, or use a large amount of speculation hardware that would remain unutilized when running a explicitly parallel program or a multiprogrammed workload.

In our thesis, we show that wide-issue dynamic processors that will soon populate CMPs would make fast communication at the register level a requirement for high performance. Consequently, we propose an effective but quite modest hardware that supports communication and synchronization of registers between on-chip processors. Furthermore, we propose hardware support that handles true memory dependence violations when the application is run in a speculative execution mode. We also present the compiler support that enables automatic identification of threads from sequential binaries. We show how the software-hardware approach enables effective speculative execution of a sequential binary on a CMP architecture without source re-compilation. Overall, we augment the CMP with just enough support, while still maintaining the generic CMP architecture to a reasonable degree.

Given that the amount of thread- and instruction-level parallelism of applications vary widely, the traditional CMP approach of statically partitioning the chip resources between threads may lead to wasted resources when one of the threads stalls due to hazards or when the application lacks threads. The Simultaneous Multithreading (SMT) architecture addresses this problem by allowing complete flexibility in resource sharing. Unfortunately, this approach like the conventional superscalar, is so centralized that it may not be a feasible architecture. In our thesis, we explore a hybrid approach, namely the clustered SMT architecture. We show that this restricted level of simultaneous multithreading is able to capture most of the performance benefits of the fully centralized approach while, at the same time, allowing the design to be decentralized.

The final part of our thesis focuses on simulation methodology. Multiprocessor system evaluation has traditionally been based on direct-execution based Execution-Driven Simulations (EDS). In such environments, the processor component of the system is not fully modeled. With wide-issue superscalar processors being the norm in today's multiprocessor nodes, there is an urgent need for modeling the processor accurately. However, using direct-execution to model a superscalar processor has been considered an open problem. In this thesis, we propose a novel direct-execution framework that allows accurate simulation of wide-issue superscalar processors without the need for code interpretation. Overall, this approach enables detailed yet fast EDS of superscalar processors for both a uni- and multi-processor configuration.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Superscalar architectures [31], which allow more than one instruction to be issued in a single cycle, have become the norm for today's high-performance microprocessors. The issue rate has continued to increase over the past few years with today's high-performance superscalar processors such as the Alpha 21164 [19], PowerPC [42], Pentium-Pro [12] or R10000 [50] able to issue up to four instructions per cycle.

Most of these processors have additional hardware that allow them to 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 dependencies between instructions [31]. This mechanism permits an instruction to be issued as soon as its operands and functional unit becomes available, rather than in the strict order as defined by the static code sequence. Consequently, this allows a high degree of instruction level parallelism (ILP) to be exploited from the program at run-time. Superscalar processors that make use of the this approach are also called *dynamic* or *out-of-order* issue processors as opposed to the ones that restrict themselves to issuing instructions in the original program order, which are also called *static* or *in-order* issue processors.

In-order issue processors such as the Alpha 21164 [19] rely completely on the compiler to generate an optimal instruction schedule that would allow multiple independent instructions to be issued every cycle. However, their relative design simplicity facilitates a faster clock. This is in contrast to the dynamic issue processors which, with their complex hardware, can issue higher instructions per cycle (IPC) but can only cycle at a lower frequency. Thus, superscalar processors may differ in the manner in which they achieve high performance by trading off clock frequency for a higher IPC or vice versa.

Indeed, this two seemingly divergent approaches have resulted in a tradition of classifying a superscalar processor as either a "speed demon" or "brainiac", depending on whether it uses a static or dynamic issue approach. However, the distinction is becoming quite blurred. Indeed, Digital Equipment Corporation (DEC), which has conventionally preferred the "speed demon" approach over the "brainiac" for its previous Alpha processors, will be unveiling the Alpha 21264 processor shortly. This processor is not only a dynamic superscalar but can also cycle at very high frequencies of 600Mhz or more[16, 26]. Interestingly, Intel has announced plans for the Merced [27, 28] processor which implements an in-order issue microarchitecture. However, unlike traditional in-order issue processors, this processor will incorporate advanced hardware features such as predication support [47] that would allow the compiler to extract ILP in a more aggressive fashion.

Nevertheless, it is not clear if any of the conventional approaches would be appropriate for very wide issue superscalars. This is because their centralized design structures have a direct impact on the critical path of the processor pipeline. These include not only the dynamic superscalar specific features such as register renaming and instruction window wake-up and select mechanisms, but also the register bypass logic common to both the in-order and dynamic issue architectures. Indeed, the bypass logic will prove to be one of the most critical structures for future wide-issue superscalar processors that are based on technology with features under $0.18\mu$ [59]. With newer generation process technologies, a billion transistor chip is likely in the the next decade [9]. Hence, there is a need for an architecture that can utilize these transistors very effectively.

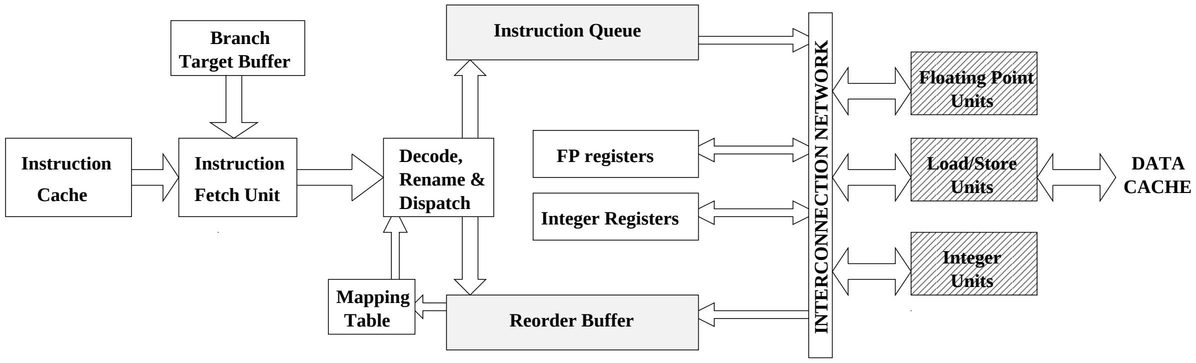



**Figure 1.1** Architecture of a typical dynamic superscalar processor.

## 1.2 Scalability of Conventional Superscalars

In this section, we discuss why a conventional superscalar architecture may not scale up efficiently. For the sake our discussions, we consider a dynamic superscalar as a conventional processor. A typical example, which is on the lines of the R10000 [50] is shown in Figure 1.1.

Let us consider each stage of the pipeline starting at the fetch stage. For a very wide-issue superscalar processor, it is imperative to have the ability to fetch across multiple basic blocks, even before an earlier branch is resolved. This is due to the fact that an average basic block in an integer application is typically only a few instructions [89, 38]. This calls for a very accurate branch prediction mechanism that would allow several successive branches to be predicted even before the first one is resolved. In addition, we also require instructions that may be present in different cache lines to be fetched and aligned properly for the decoding stage. Instruction alignment may be achieved with advanced schemes such as the collapsing buffer [13] or the Trace cache [68] approach. Also, proposed branch prediction techniques [10, 93] have shown that a highly accurate prediction is feasible, provided a substantial amount of hardware is invested towards this purpose. However, all these schemes would still fall short of ideal prediction and any misprediction will have a much bigger impact on very high-issue superscalar processors with deep pipelines compared to low-issue superscalar processors. For instance, a pipeline stall of $x$ cycles on a $y$ wide-issue processor implies a wastage of $xy$ slots for each stage of

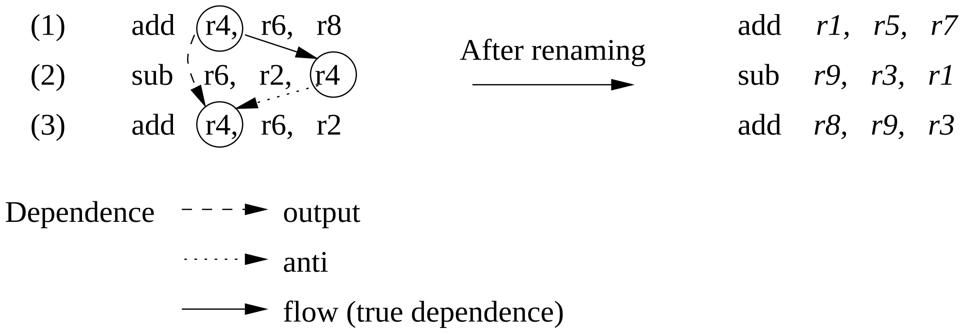



**Figure 1.2** Removing false dependencies by renaming registers.

the pipeline until the issue stage. Furthermore, the instruction queue has to be flushed of speculative instructions and the architectural state before the misprediction needs to be restored. Overall, aggressive speculation not only complicates the hardware but also increases the power requirements of the processor [48].

Dynamic superscalars are able to issue instructions out of order by removing any false dependencies (output- and anti-dependencies [2, 90]) among instructions. This is achieved by using a renaming logic that maps logical registers into physical registers at run time [92]. Typically, a table is used to store the mapping between the logical and physical registers. When an instruction writes to a logical register, a new physical register is alloted to that instruction. Without this renaming mechanism, anti- and output- dependencies may prevent instructions from being issued. This is illustrated in Figure 1.2. Here, register $r4$ causes a true dependence between the instruction pair (1, 2) while it results in an anti-dependence and output dependence between instruction pairs (2,3) and (1,3) respectively. Renaming removes the false dependencies leaving only the true dependence in the code sequence on the right.

However, increasing the issue-width of a superscalar processor increases the port requirements of the renaming table. Given that an instruction may have two read and one write operand, the number of ports that is required increases linearly with issue width. Thus, a $n$ wide-issue processor may require up to $2n$ read and $n$ write ports. Furthermore, additional logic is also required to check for true dependencies within the instructions being considered for issue.

Moving on to the issue window logic, the instruction queue (or window) holds the fetched instructions that are waiting for the operands or a functional unit to be ready before they can be dispatched to the appropriate unit. The instruction queue may be a single queue or could be partitioned to reduce the hardware complexity [73]. Additional logic is also required that allows the broadcast of the results to all the entries in the queue. Unfortunately, the complexity of this logic increases linearly with the size of the window and the issue width [59]. As for the instruction operand values, it is likely that an instruction may be data dependent on an instruction that was issued in a earlier cycle but yet to update the register file. Consequently, the ability to bypass the data from the register file with that of a functional unit is a necessity for high performance. However, the size of the bypass network grows quadratically while the bypass delay itself increases linearly with increasing issue width [59]. Indeed, the window logic and the bypass logic will increasingly dictate the cycle time of future high issue processors. The primary reason is that on-chip interconnects, hitherto not a significant factor, cannot scale well and are likely to become a bottleneck in future processors, where a signal is expected to take multiple clock cycles to traverse the entire chip [6, 49]. Unlike transistors, whose performance is unaffected with reductions in dimensions when moving to newer-generation technologies, thinning the metal for the interconnects decreases the speed at which the signal can propagate in the wires. This is because the speed is proportional to the wire resistance which increases with thinner wires. Unfortunately, increasing the wire thickness to offset this problem would decrease the density, leading to wastage of silicon space. This problem can be ameliorated somewhat by increasing the number of metal layers. However, the number of layers have to increase tremendously to satisfy the density requirement, which is not a feasible long-term solution [5]. Overall, the design challenge posed by the interconnects can never be underestimated. Indeed, with current $0.25\mu$ technology, Intel spends over 50% of its total process cost in forming the interconnects and the situation is only bound to deteriorate with finer implementations [5].

To conclude, the conventional superscalar approach is so centralized that it may not scale well when higher issue bandwidth is required. A limited effect can be seen even in today's Alpha 21264 [26] processor, where the integer units are clustered into two sets. This was done primarily to avoid complicating the data path as it might have resulted in a larger cycle time.

## 1.3 Alternative Approaches

As mentioned in the previous section, a conventional superscalar would not be able to scale well. However, advances in VLSI technology now allow over 100 million transistors to be configured on a single die [41] and a billion-transistor chip is in the offing [9]. Hence, there is a real need for alternative architectures that can exploit the increasing number of transistors in an effective manner. For alternative architectures, we choose to focus on those that are oriented towards improving the CPU or the computational part of the processor. There have been proposals that take a more memory-centric view and argue for integration of memory and logic on the same chip [34, 70]. However, we consider this approach completely orthogonal to the CPU-centric approach. Though it is not clear at this point, it is possible that a future architecture may accommodate both the approaches.

A fundamental limitation of the conventional superscalar approach is that it exploits ILP in a single thread or execution path in the application. This is also the primary reason for the design to be so centralized. A natural solution that would pave the way for a decentralized architecture would be to divide the application into multiple threads or tasks and exploit ILP across them. Thus, rather than implementing just one, but a very centralized high-issue superscalar processor on a chip, many researchers have proposed a de-centralized architecture wherein, multiple simpler processing units are configured on a single chip. This is also known as the chip-multiprocessor(CMP) approach. Alternatively, some researchers have proposed upgrading a conventional superscalar to support multiple threads. We consider these two alternative architectures in the following sections.

```
for (i = 0; i < n; i++)

   .... = a[x[i]];
   a[x[i]] = ....;

end
```

**Figure 1.3** Example of a loop that will not be parallelized by a conservative compiler.

### 1.3.1 Chip Multiprocessor

The chip-multiprocessor(CMP) has drawn great attention, with architects proposing various related designs [21, 30, 69, 76, 78, 82]. The design simplicity of the CMP approach not only allows for a much faster clock in each of the processing unit, but also allows efficient use of the resources. The interconnect problem discussed in the previous section would also be ameliorated since such a decentralized implementation would permit fast communication in their localized interconnects. In addition, the CMP approach eases the time-consuming design validation phase that plagues complex centralized designs.

The CMP approach also results in better utilization of the silicon space. By avoiding the extra logic devoted for the centralized architecture, a CMP allows the chip to have a higher issue bandwidth when compared to a conventional superscalar implemented on the same die area [30]. Olukotun et al [30] 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 very promising approach for exploiting the increasing on-chip transistor density.

From a software perspective, the CMP is an ideal platform to run a multiprogrammed workload or a multi-threaded application. However, if it is to be fully accepted, it must also be able to give good performance when running sequential applications. With parallelizing compilers [4, 29] 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 and assumes the existence of inter-thread dependencies when it cannot guarantee data independence between threads. This is illustrated by a loop segment shown in Figure 1.3. 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 would access distinct array locations. As a result, this loop segment would be marked serial by the compiler. The situation becomes even worse for non-numerical applications which may have dependencies through pointers.

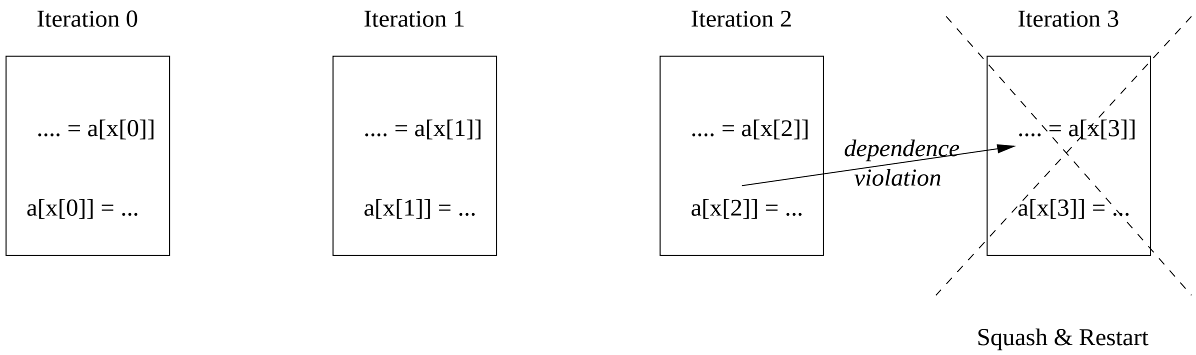



**Figure 1.4** Exploiting partial parallelism on a CMP. Here, only x[2] and x[3] have identical values.

To address this drawback, speculation may be used. In this execution mode, different threads are executed in parallel speculatively. Added software or hardware support enables detection and recovery from dependence violations. Such an approach has been proposed for numerical applications running on a conventional multiprocessor where a dependence violation would result in the code being executed serially [63, 94]. This all-or-nothing approach is more applicable for numerical applications where dependencies are infrequent. However for better performance of general-purpose applications, where dependencies are quite common, do-across or partial parallelism also needs to be exploited. This is the approach used by speculation-based CMPs. In this mode, the threads that execute on the on-chip processing units do not need to be fully independent; they may have data dependencies with each other. Figure 1.4 illustrates how a CMP based with four processing units can exploit the partial parallelism in the loop shown in Figure 1.3 assuming that only $x[2]$ and $x[3]$ have identical values.

To exploit this type of parallelism, speculative threads need to identified in the application. Such speculative threads may be identified either at compile time [17, 30, 76, 78, 82] or completely at run-time with hardware support such as a trace cache [68]. Recently, it has been shown that there is indeed considerable performance potential for even integer applications when using this speculative approach [78]. Overall, CMPs can not only be utilized as a very effective throughput enhancing platform when running multiprogrammed workloads but also be used for achieving substantial speedups for a wide range of applications.

### 1.3.2 Simultaneous Multithreading

In the CMP systems, the main characteristic is that a processing unit in a chip is allocated exclusively to a thread such that all resources have a fixed assignment to a thread. The consequence of this approach is that resources such as the fetch or functional units can be easily wasted, especially when a thread cannot utilize the resource due to a hazard. To utilize the resources better, we can design a more centralized architecture where threads can share all the resources. This is the approach taken by Simultaneous Multithreading (SMT) [84]. This strategy and its variations have been the subject of intense study [25, 32, 33, 62, 84]. Perhaps the most detailed design has been proposed by Tullsen *et al* [83]. They have discussed a SMT architecture with little impact on a conventional superscalar processor design. In this scheme, the processor can support several hardware threads such that instructions from multiple threads can be issued in the same cycle.

The major advantage of the SMT approach is that the resources tend to be highly utilized. If, in a given cycle, a thread is not using a resource, that resource can, typically, be utilized by another thread. This is illustrated in Figure 1.5. Here the SMT processor utilizes the issue slots more effectively leading to a higher IPC when compared to a CMP. Thus, we can get a high throughput if we run multiprogrammed workloads and a high speedup if we run a multithreaded application. However, a drawback of this approach is that it may inherit all the complexity of existing superscalars and, in addition, add extra

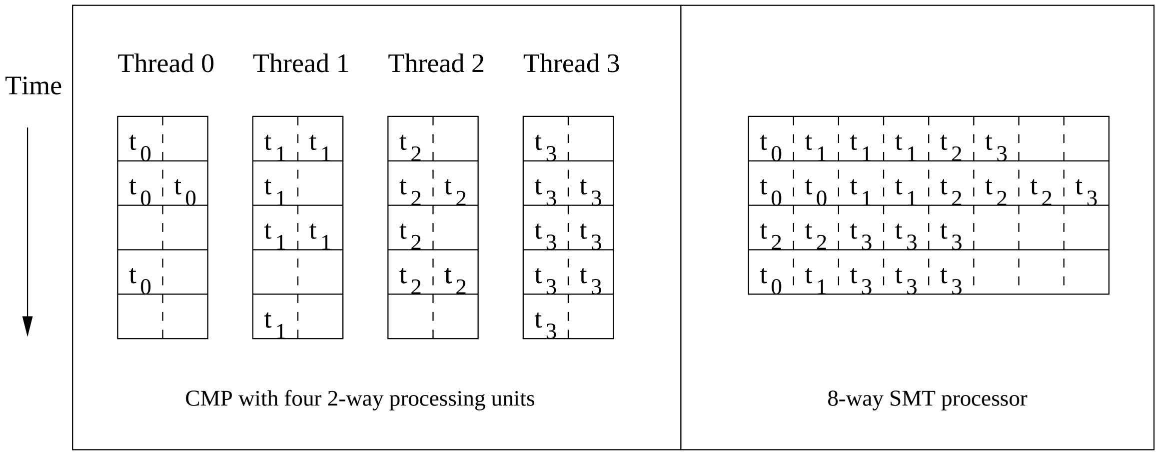



**Figure 1.5** Comparing a CMP with a Simultaneous Multithreading Processor.

hardware. In fact, with the delays in the register bypass network dictating the cycle-time of future high-issue processors [59], a fully centralized SMT processor may likely have a slower clock frequency. This calls for a hybrid approach that uses a decentralized design while at the same time can capture most of the benefits of SMT.

## 1.4 Thesis Contributions

### 1.4.1 Speculation-based Chip Multiprocessors

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.

A major portion of our thesis involves presenting 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. We then show that wide-issue dynamic processors that will soon populate speculative CMPs, make fast communication at the register level a requirement for high performance. Consequently, we propose an effective hardware mechanism to support communication and synchronization of registers between on-chip processors. Our scheme adds enough support to enable register-level communication without specializing the architecture so much towards speculative mode of execution, that it leads to much unutilized hardware under fully-parallel or multiprogrammed loads. We also show that the scheme allows the system to achieve near ideal performance. Furthermore, we propose hardware support to enable detection of memory dependence violations when threads are executed speculatively. This involves a memory disambiguation structure and a modified cache-coherence protocol, along the lines of a directory-based protocol, to maintain data consistency. Finally, we evaluate this hardware-software approach and show that it can indeed lead to high performance when executing sequential binaries.

### 1.4.2 A Clustered Approach to Multithreading

There is much interest in alternate architectures that exploit thread-level parallelism. With future applications likely to be multithreaded, this is a very promising approach. The chip-multiprocessor (CMP) is configured such that the chip resources are partitioned in a fixed manner between the threads. These fixed-assignment architectures risk wasting resources when one of the threads stalls due to hazards or when there is lack of threads in the application. Although basic simultaneous multithreading (SMT) allows resources to be shared, the structure is so centralized that it is likely to slow down the clock frequency. However, given that the amount of thread- and instruction-level parallelism of applications varies widely, the flexibility of the SMT architecture is very valuable.

In the second portion of the thesis, we explore a hybrid design, where a chip is composed of a set of SMT processors. We evaluate such a clustered architecture running parallel applications. We consider both a low-end machine with only one processor chip on which to run multiple threads as well as a high-end machine with several processor chips working on the same application. Overall, we conclude that such a hybrid processor represents a good performance-complexity design point.

### 1.4.3 A Direct-Execution Simulation Framework

Multiprocessor system evaluation has traditionally been based on direct-execution based Execution-Driven Simulations (EDS). In such environments, the processor component of the system is not fully modeled. With wide-issue superscalar processors being the norm in today's multiprocessor nodes, there is an urgent need for modeling the processor accurately. However, using direct-execution to model a superscalar processor has been considered an open problem [57]. Hence, current approaches model the processor by interpreting the application executable. Unfortunately, this approach can be slow.

In this thesis, we propose a novel direct-execution framework that allows accurate simulation of wide-issue superscalar processors without the need for code interpretation. This is achieved with the aid of an *Interface Window* between the front-end and the architectural simulator, that buffers the necessary information. This eliminates the need for full-fledged instruction emulation. Overall, this approach enables detailed yet fast EDS of superscalar processors. Finally, we evaluate the framework and show good performance for uni- and multi-processor configurations.

## 1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 proposes and evaluates the speculative chip-multiprocessor architecture. In Chapter 3 we explore the clustered approach to multithreading. Chapter 4 describes in detail the novel direct-execution based framework which we use for our simulations. Finally, Chapter 5 concludes our thesis and suggests possible future directions.

# CHAPTER 2

# SOFTWARE AND HARDWARE SUPPORT FOR A SPECULATION-BASED CHIP-MULTIPROCESSOR

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 multi-threaded applications, it may not be the case 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 dependencies 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 [69, 72] and Multiscalar [76] 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 [68] 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 statically by recompiling the source program, it can also be done from the binary. At run-time, 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 [7]. Overall, both 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 full re-compilation of the source program. 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) [55, 78, 82], 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 [78]. However, the need to re-compile the source is a handicap, especially when the source is unavailable.

In this chapter, 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 being able to handle sequential binaries quite effectively.

The chapter is organized as follows: Section 2.1 describes our software support for identifying threads in a sequential binary. Section 2.2 motivates the need for register-level communication between threads in a chip-multiprocessor and looks at the impact of restricting communication only through memory; Sections 2.3 and 2.4 describes our hardware support for register communication and enforcing true memory dependencies between threads on a speculative-CMP respectively; Section 2.5 performs the evaluation; and finally, Section 2.8 concludes the chapter.

## 2.1 Software Support: Binary Annotator

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 [17, 30, 76, 78, 82] or in hardware by using special hardware support that identifies threads at run-time [68]. We use a software approach. However, we perform the compilation step on the executable file, thereby avoiding recompiling the source program.

We have developed a binary annotator that identifies units of work for each thread and the register-level dependencies between these threads. Memory dependencies across threads are handled by the additional hardware described in Section 2.4. Currently, we limit the threads to inner loop iterations, where the loop body may have recursive function calls.

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. 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.

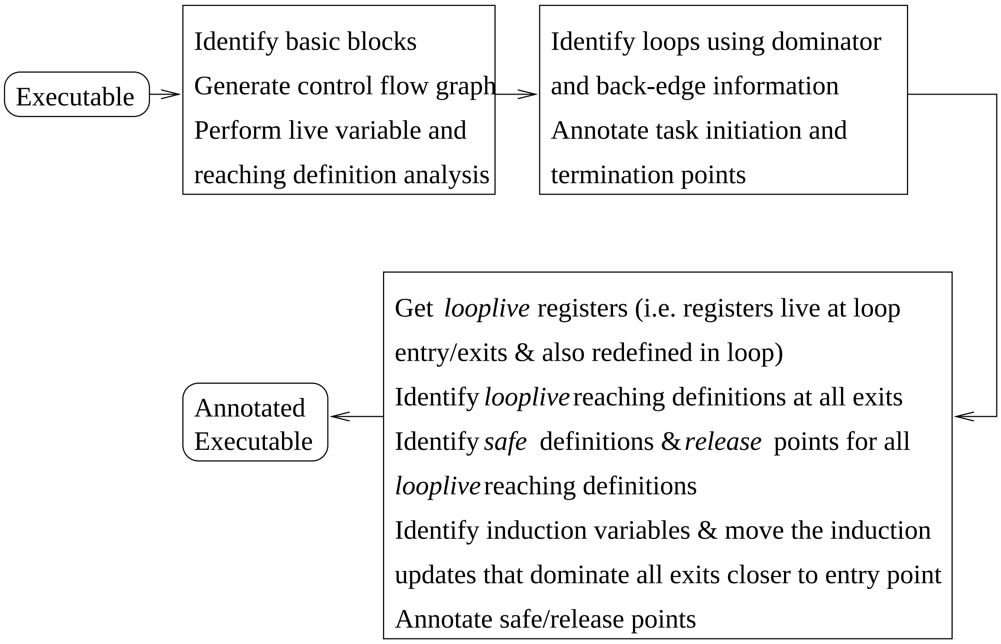



**Figure 2.1** Binary annotation process.

To simplify the hardware support, we follow sequential semantics for thread termination. In other words, 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, when a succeeding iteration loads a memory location before a predecessor stores to the same location, all iterations starting from the iteration that loaded the wrong value are squashed and restarted.

The steps involved in the annotation process are illustrated in Figure 2.1. The approach that we use is similar to that of Multiscalar [76] except that we operate on the

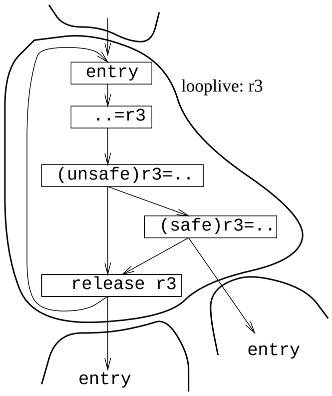



**Figure 2.2** Safe definitions and release points.

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 dependencies 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 2.2 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.

Induction variables, which are identified using the standard techniques given in Aho et al [1], need special consideration. Typically, induction variables are used before being updated. This is illustrated in Figure 2.3-(a) where induction registers $r1$ and $r2$ are used before modification. This would, however, delay the successor threads unnecessarily. To overcome this problem, we assume that the value is modified before being used. This can be achieved by peeling the loop iteration once and moving the update to the beginning

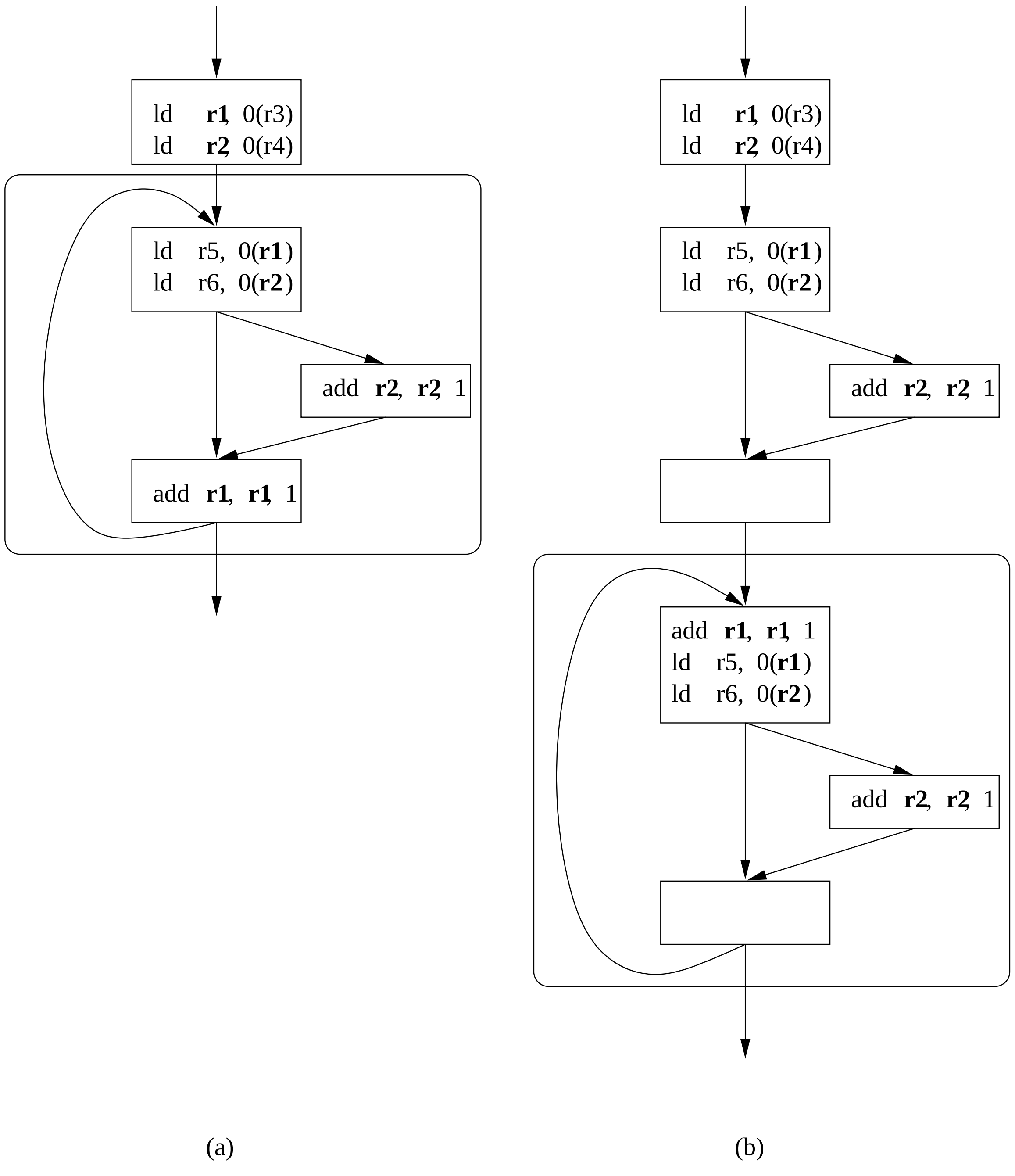



**Figure 2.3** Handling induction variables (a) Induction register $r1$ used before update (b) After loop peeling, $r1$ is updated before use

of the loop body. The modified code is shown in Figure 2.3-(b). In effect, we percolate the loop updates as close as possible to the thread entry point. It must be noted that such a code motion is permitted only for those induction updates that dominate the exit points of the loop. In our example, the update to register $r1$ dominates the loop exit and is therefore eligible for this code motion. However, those induction updates that do not satisfy this criterion are not moved upwards. This is possible when an induction variable is updated conditionally in the loop body. This is the case with register $r2$ in the example. Therefore, it continues to be used before being possibly updated later in the loop. Even though such updates may result in the successor iteration being delayed, nevertheless, we can reduce to a great degree, the waiting time for the successor iterations by moving those updates that dominate the exit points of the loop.

In closing, it must be noted 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.

## 2.2 Register-Level Communication in CMPs

### 2.2.1 Motivation

When loops are run speculatively on CMPs, each iteration is executed assuming there are no inter-iteration data dependencies. In the presence of dependencies, however, this would violate sequential semantics. For this reason, proposed CMP architectures include hardware support to enforce data dependencies, enabling consumer threads to acquire the appropriate value from the producer thread. In addition, these architectures also include hardware to detect dependence violations when they occur, so that threads can be squashed and restarted. Dependence enforcement and violation detection can be done with centralized schemes like the ARB [23] or with distributed schemes that use a modified cache-coherence protocol among the processors in the CMP [24, 56, 78].

```
     while (TRUE) {
        if (prev == NIL) return;
(1)     if (prev->n_flags & LEFT) {
          if (livecdr(prev)) {
            prev->n_flags &= ~LEFT;
            tmp = car(prev)
            rplaca(prev,this);
            this = cdr(prev);
            rplacd(prev,tmp);
            break;
          } else {
            tmp = prev;
(2)         prev = car(tmp);
            rplaca(tmp,this);
            this = tmp;
          }
        } else {
          tmp = prev;
(3)       prev = cdr(tmp);
          rplacd(tmp,this);
          this = tmp;
        }
  }
```

**Figure 2.4** Code segment from *li*.

```
C Code:

          for (i = 0; i < ninputs; i++) {
             aa = a[0]->ptand[i];
             bb = b[0]->ptand[i];
             ....
             ....
          }


Assembly Code:

           (1)   lw   r14, 0(r4)      /* r14 = &a[0] */
           (2)   lw   r15, 0(r5)      /* r15 = &b[0] */
           (3)   lw   r5, 0(r14)      /* r5 = &a[0]->ptand */
           (4)   lw   r7, 0(r15)      /* r7 = &b[0]->ptand */

          LOOP:  ......
                 ......
                 lh   r4, 0(r5)       /* aa = a[0]->ptand[i] */
                 lh   r6, 0(r7)       /* bb = b[0]->ptand[i] */
                 ......
                 ......
                 addu r5, r5, 2
                 addu r7, r7, 2
                 addu r2, r2, 1
                 .....
                 blt  r2, r3, LOOP
```

**Figure 2.5** Code segment from *eqntott*.

Frequently, however, the variable causing the dependence is known to the compiler. This is illustrated in Figure 2.4, which shows one of the most frequently executed loops from SPEC95's *li* application. Each iteration of the loop reads the variable *prev* and may later update it. In that case, we can use more advanced mechanisms than the ones described above. Specifically, depending on the hardware support available, two alternatives are possible.

One approach is for the compiler to insert a synchronization step through the memory hierarchy between the producer and the consumer thread. This approach requires an efficient synchronization mechanism, like a hardware-based full-empty bit mechanism [71]. For example, given an iteration, we know that it will not update *prev* after it has gone beyond any of the statements (1), (2) or (3) in the figure. At any of these points, an iteration can synchronize with the next one through, for example, a shared L2 cache. If, in addition, the producer has just updated *prev*, the new value can be forwarded to the L2 cache in the same synchronization step.

The second approach is for the threads to communicate via registers. It helps that variables like *prev* are typically allocated in registers by any optimizing compiler. Even though this approach requires special hardware support for synchronization and communication at the register level [21, 69, 76], it is very fast to perform.

It could be argued that CMPs could be based on single-issue processors thereby allowing more processors to be configured on-chip. However, exploiting both thread-level and instruction-level parallelism is critical for the performance of multithreaded applications [44]. Thus we assume that CMPs would be based on wide-issue dynamic superscalar processors, rather than static scalar processors. In this context, we argue that the wide-issue superscalar processors that will soon populate CMPs would make register communication a requirement for high performance. In addition, we will show that the hardware support for register communication can be made simple without using specialized mechanisms.

When wide-issue dynamic superscalar processors are used in CMPs, performing synchronization through memory would be slow. Consider, for example, a CMP with four

4-issue dynamic superscalar processors. Synchronizing two threads through the L2 cache has a high relative cost. For example, assume that the L2 cache has advanced hardware support for synchronization like a full/empty bit. The minimum time that it takes to transfer a datum between two threads is a round trip access to the L2 cache. This may easily be 6 cycles, even for on-chip caches. Each thread could have executed many instructions in this time.

If, instead, register communication is used, the latencies can be made much smaller. Depending on the distance between the two threads on the chip, the latency may vary. If we assume a $0.18\mu$ technology, the entire die can be covered in 2 clock cycles, while for $0.13\mu$ technology, this would increase to 4 cycles [49].

A second, less important performance advantage of register-level communication is that it needs to execute fewer instructions. This is because memory-level synchronization often adds extra memory access instructions. For the example in Figure 2.4, *prev* must be loaded explicitly from memory at the three synchronization points. This is also true when there are multiple induction variables in the loop. This can be seen in the most frequently-executed code segment from SPEC92's *eqntott* application (Figure 2.5). In the figure, loads (1) to (4) will have to be re-executed by each processor every iteration. This can be avoided by communicating the induction registers `r5` and `r7` from each processor to the next one. Overall, however, the performance impact of any extra instructions executed by wide-issue processors is likely to be small.

### 2.2.2 Evaluation Environment

Our analysis of the impact of register and memory communication is performed using simulations. In this section, we describe the architectures modeled, the simulation approach, and the way statistics are collected.

### 2.2.3 Architectures Modeled

We model CMPs with 4 processors, where each processor can be a 1-, 2- or 4-issue dynamic superscalar. The superscalar processor is modeled on the lines of the MIPS R10K [50]. This dynamic superscalar core has a large fully-associative instruction window along with integer and floating-point registers for renaming. The actual numbers are shown in Table 2.1. 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. The only exceptions are integer and floating-point multiply and divide operations: integer multiply and divide takes 2 and 8 cycles respectively; floating-point multiplies take 2 cycles, while floating-point divides take 4 cycles for single precision and 7 cycles for double precision.

| Issue Width | Number of Functional Units (int/ld-st/fp) | Entries in Instruction Window | Number of Renaming Registers int/fp) |
|---|---|---|---|
| 1 | 1/1/1 | 16 | 16/16 |
| 2 | 2/1/1 | 32 | 32/32 |
| 4 | 4/2/2 | 64 | 64/64 |

**Table 2.1** Characteristics of the dynamic superscalar core.

We model the memory subsystem 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. The 4-issue processor can have up to 32 outstanding memory accesses, of which 16 can be loads, while for a 2- and 1-issue processor, the limits are 16(8) and 8(4) outstanding memory accesses(loads) respectively. Each processor in the CMP has a relatively small private L1-cache of 16K. So for a 4-processor CMP, the total size occupied by the private L1 caches would be 64K. All processors then share a larger on-chip L2-cache. The characteristics of the caches are shown in Table 2.2.

| Parameter | Value |
|---|---|
| [L1 / L2] Cache Size (Kbytes) | [16 / 1024] |
| [L1 / L2] Cache Line Size (Bytes) | [32 / 64] |
| [L1 / L2] Cache Associativity | [2-Way / 4-Way] |
| L1 Banks | 3 |
| L1 Latency (Cycles) | 1 |
| L2 Latency (Cycles) | 6 |
| Memory Latency (Cycles) | 26 |

**Table 2.2** Characteristics of the memory hierarchy in a 4-processor CMP. All latencies correspond to a contention-free round trip.

### 2.2.4 Simulation Approach

Our simulation environment is built on a modified MINT [88] execution-driven simulation environment. Our back-end simulator is very detailed and performs a cycle-accurate simulation of the different CMP architectures. We will discuss the simulator framework in detail in Chapter 4.

Finally, we use binaries generated on the SGI PowerChallenge for eight applications. The applications, shown in Table 2.3, belong to the SPEC95, MediaBench [39] and Perfect Club [20] suites. We use the train set as input for the SPEC95 applications and the default input for the rest of the applications.

| Application | Suite | Explanation |
|---|---|---|
| *hydro2d* | SPECfp95 | Navier Stokes equations |
| *wave5* | SPECfp95 | Maxwell's equations |
| *ocean* | Perfect Club | Two-dimensional Boussinesq fluid layer |
| *trfd* | Perfect Club | Two-electron integral transformation |
| *li* | SPECint95 | Xlisp interpreter |
| *ijpeg* | SPECint95 | Image compression/decompression on in-memory images |
| *adpcm* | MediaBench | Speech compression and decompression algorithms |
| *epic* | MediaBench | Image data compression |

**Table 2.3** Applications used in our evaluation.

### 2.2.5 Statistics Collection

We gather detailed statistics on an issue slot basis. For each processor in the CMP, we scan the entire instruction window every cycle and record the type of hazard faced by each instruction that is unable to issue. At the end, the wasted slots are divided proportionally among the different types of hazards. The different categories of hazards are: waiting for the synchronizing value (*sync*), data dependencies (*data*), memory access (*memory*), serial sections (*serial*), and instructions squashed on branch mis-predictions or when a thread is squashed and restarted (*squashed*), There is also an *other* category for slots wasted due to structural and control hazards as well as lack of renaming registers. Finally, the issued instruction slots are grouped under *issued*.

### 2.2.6 Impact of Communication Latency in a Speculative CMP

To evaluate the performance impact of memory-level synchronization, we simulate 4-processor CMPs where all the cross-thread dependencies that could use register communication are communicated through the L2 cache. As mentioned in section 2.2.1, the threads are synchronized using a full-empty bit [71] mechanism. We assume special storage beside the L2 cache where the synchronization variables are kept. For our experiments, we do not consider induction variables nor do we model the extra instructions that are needed to evaluate those variables independently in each processor of the CMP.

Note that, unlike register dependencies, we cannot identify memory dependencies easily from the binary. Therefore, we assign the full responsibility of detecting memory dependencies to the hardware. We use a modified cache-coherence protocol that enforces data dependencies as well as detects violations. When a data dependence violation occurs, for example when a successor thread has speculatively loaded a memory location that a predecessor now updates, threads are squashed and then restarted. We squash the thread that violated the dependence and all its successors. Overall, our cache coherence protocol is very similar to those of [24, 56, 78]. We elaborate further on the support for handling memory-level dependencies in Section 2.4. This support is common to both the

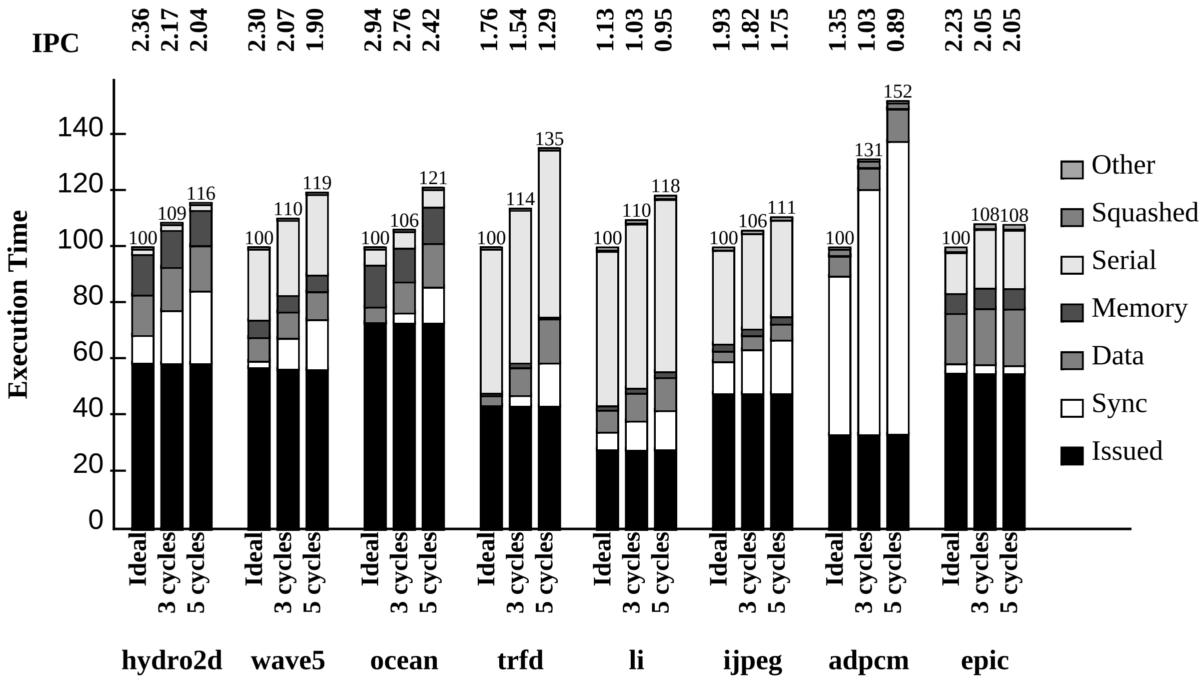



**Figure 2.6** Impact of the L2 latency on the execution time and IPC for a CMP based on four 1-issue dynamic processors.

approaches where communication may occur through register or memory and will not play any role in the performance impact study.

We consider four types of CMPs by increasing the complexity of the processors in the CMP from 1-issue to 4-issue dynamic superscalars. We assume a constant L2 latency by not including any port contention for accesses to the communicated variables. We consider three environments by varying the latency of the L2 cache. Two of them have L2 latencies of 3 and 5 cycles respectively, while in the third environment (*Ideal*), there is no communication latency: any producer update is visible to the consumer instantaneously.

Figure 2.6 shows the execution time of the eight applications on a CMP with four 1-issue dynamic processors. The execution time is divided into the different categories described in Section 2.2.5. For each application, the bars are normalized to the *Ideal* environment. The IPC for each application is given in the top of the figure.

From the figure, we can see that the added latency translates directly into a large synchronization time, thereby resulting in an increased execution time. Note that, in our setup, a processor in the CMP does not block if the access to the full/empty bit does not immediately succeed. Instead, the access remains outstanding and the processor can continue issuing instructions not dependent on that access. In the figure, the *sync* time in *Ideal* shows the time when the consumer arrives at the synchronization point before the producer. Applications such as *wave*, *ocean* and *trfd* have little or no *sync* time in *Ideal*, denoting that the value is always available to the consumer. However, with an increase in communication latency, *sync* time becomes significant enough to cause a performance degradation of 19% to 35%.

*Adpcm* belongs to the other extreme case, where even *Ideal* performs poorly. Though this application has no serial sections and fully exploits the speculative mode, the performance is severely affected by the presence of several cross-iteration dependencies. This invariably results in many instructions being unable to issue, eventually leading up to an execution stall. An increase in the communication latency only exacerbates the problem, resulting in a 50% slowdown relative to *Ideal*. The remaining applications also perform poorly as communication latency increases. Overall, we observe performance losses of 6% to 31% with a 3-cycle L2 latency and 11% to 52% with a 5-cycle latency.

Since CMPs will likely be built out of superscalar processors, Figures 2.7 and 2.8 show the results of the previous experiments for CMPs with 2- and 4-issue dynamic superscalars respectively. Comparing this data to that in Figure 2.6, we see that, for nearly all applications, the performance difference between *Ideal* and the rest has widened. While for the 1-issue processors the average difference in performance is 11% and 23% with a 3-cycle and 5-cycle latency respectively, the difference jumps to 18% (3-cycles) and 35% (5-cycles) for the 2-issue processors and, finally, to 24% (3-cycles) and 46% (5-cycles) for the 4-issue processors. These results show that there is a real need for a fast communication mechanism for these variables in speculative CMPs, and that it becomes more important when higher-issue processors are used as building blocks.

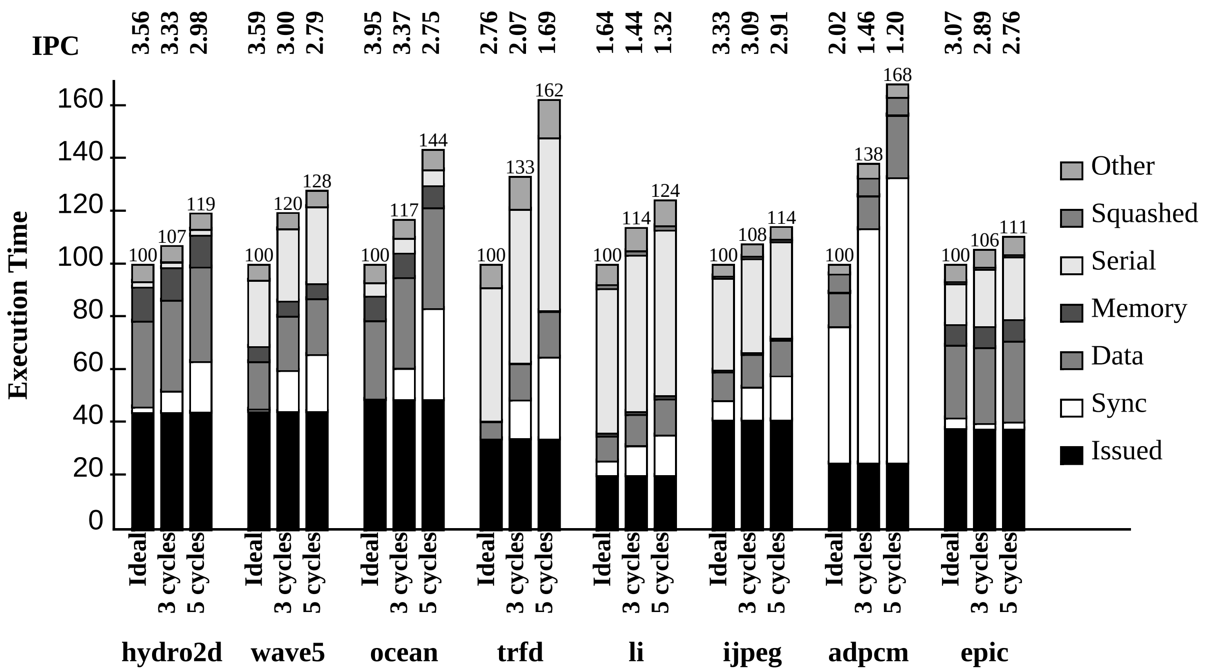



**Figure 2.7** Impact of the L2 latency on the execution time and IPC for a CMP based on four 2-issue dynamic superscalar processors.

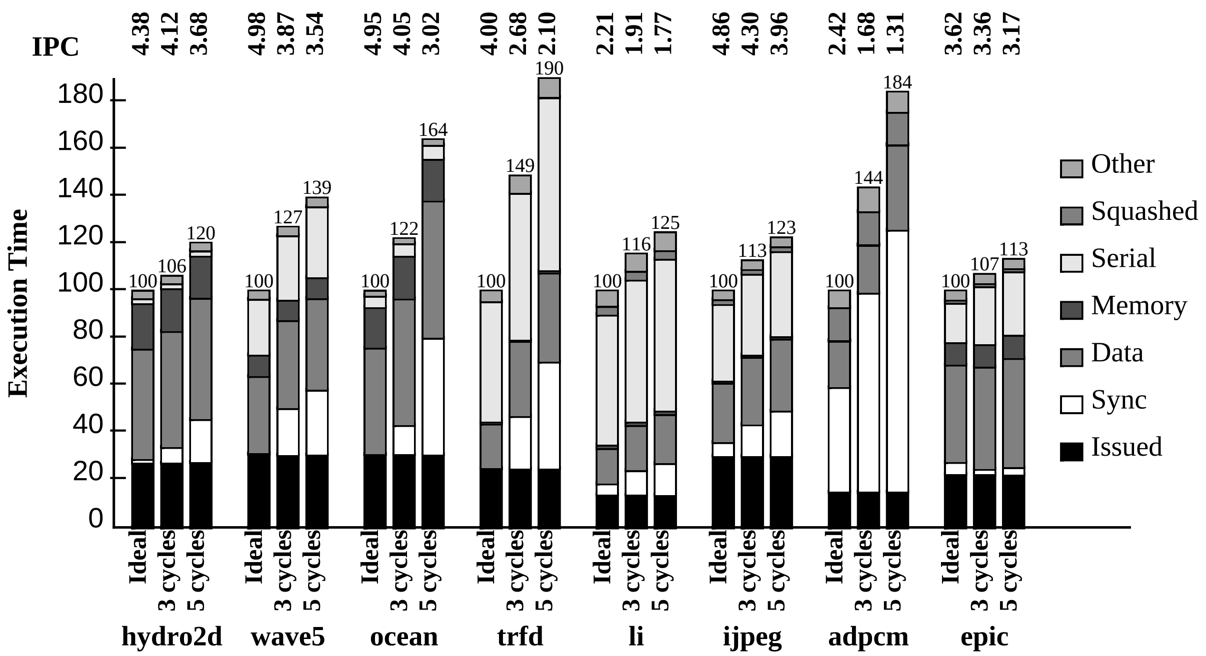



**Figure 2.8** Impact of the L2 latency on the execution time and IPC for a CMP based on four 4-issue dynamic superscalar processors.

## 2.3 Hardware Support for Register-Level Communication

The fast communication needed in speculative CMPs can be provided by register-level transfers. To provide flexible inter-thread register communication, we propose augmenting the CMP with a decentralized hardware data structure that extends the functionality of a scoreboard. We call this augmented module the *Synchronizing Scoreboard (SS)*. In this section, we describe its functionality and evaluate its performance.

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 2.4. 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.

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

**Table 2.4** Status masks maintained by each thread.

### 2.3.1 Synchronizing Scoreboard Functionality

The SS is used by threads to synchronize and communicate register values. It is a scoreboard augmented with additional bits. The SSs in the different processors are connected with a simple broadcast bus, on which register values are transferred. This bus, which we call the *SS Bus*, has a limited bandwidth of 1 word/cycle and one read and one write port for each processor. For a 4-processor CMP, a value written by a

| 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 2.5** Example of the global fields of a SS.

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 $0.13\mu$ chip technology, where a signal would take up to 4 clock cycles to traverse the entire die [49]. In addition, there may be the added delay of contention for the SS bus. The overall hardware setup is shown in Figure 2.9. As shown, the SS bus is a simple broadcast bus, and can be implemented on the chip's outer metal layers to enable inter-processor communication [5]. Note that, unlike the Multiscalar [76] or the Superthreaded approach [82], we do not use a ring network. Thus, data transfer between two successive threads running on processor 3 and processor 0 may take at least 3 cycles.

Each SS has one entry per register. Figure 2.9 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 2.5 shows an 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 2.1). When this occurs, the thread also

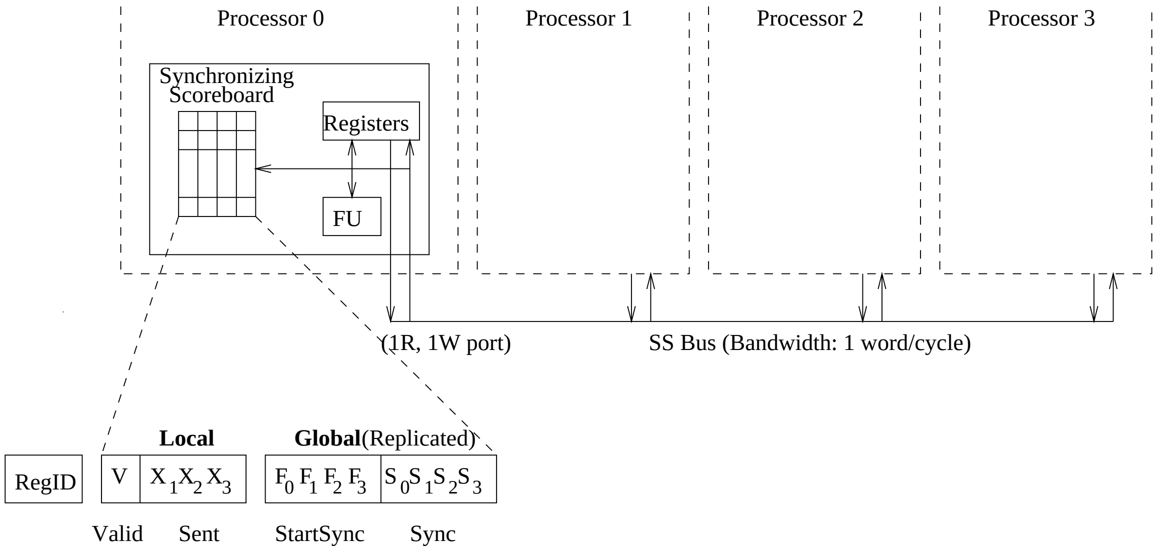



**Figure 2.9** Hardware support for register communication.

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 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 $i$, it 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. We will see its use later.

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 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 two issues, namely how registers are communicated between processors, and the problem of the last copy.

#### 2.3.1.1 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 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 generates the register. We could allow the values to be stored, by using a buffered communication mechanism, rather than using a simple broadcast bus. But this would also require further hardware support in the form of duplicate register sets in each processor to enable recovery from squashes [7]. Alternatively, a global register set may be maintained to store the values [69], but at the cost of maintaining a centralized structure. In our scheme, we add minimal hardware to support a consumer-initiated approach, where communication occurs when the consumer 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. 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:

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

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:

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

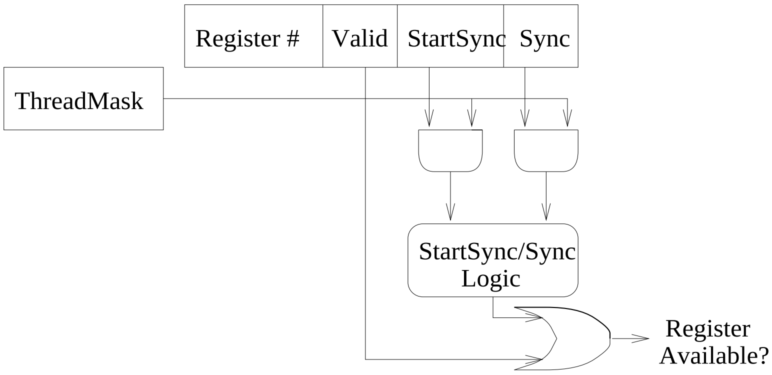



**Figure 2.10** Logic to check register availability.

The accesses to these bits are always masked out with the $ThreadMask$ of Table 2.4. 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 2.4. This means that we are examining only 2 predecessors. The resulting formula 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 2.10 and 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-consumer distance, plus the contention for the SS bus. The bandwidth that is required to obtain optimal performance is evaluated later in the section.

## Example:

The example in Figure 2.11 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 is shown in Figure 2.11-(a). 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.

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 is shown in Figure 2.11-(b).

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 is shown in Figure 2.11-(c).

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, and determines that the value is available in the non-speculative thread $t+1$, and puts a request on the SS bus. This is a consumer-

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

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $F_0$ $F_1$ $F_2$ $F_3$ | $S_0$ $S_1$ $S_2$ $S_3$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 1 0 0 | 0 1 0 0 |

(a)

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

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $F_0$ $F_1$ $F_2$ $F_3$ | $S_0$ $S_1$ $S_2$ $S_3$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 1 0 0 | 0 0 0 0 |

(b)

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

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $F_0$ $F_1$ $F_2$ $F_3$ | $S_0$ $S_1$ $S_2$ $S_3$ |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 1 0 0 | 0 0 0 0 |

(c)

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

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $F_0$ $F_1$ $F_2$ $F_3$ | $S_0$ $S_1$ $S_2$ $S_3$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 1 0 0 | 0 0 0 0 |

(d)

**Figure 2.11** Example to illustrate the working of the SS mechanism.

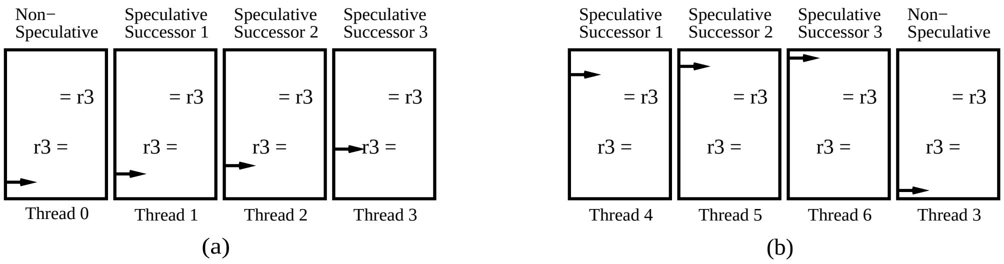



**Figure 2.12** The last copy problem. The arrow points to the instruction currently being executed in each thread.

driven approach. Finally, when processor 1 supplies the value to the bus, $r3$ in processor 0 becomes valid. The scoreboard entry after this operation is shown in Figure 2.11-(d).

#### 2.3.1.2 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 call this the last copy problem.

The last copy problem is illustrated in Figure 2.12. 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 2.12-(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 2.12-(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.

The last copy problem will not occur when using a communication mechanism that buffers live-out values to percolate to the new threads or when a centralized global register set is assumed that maintains live-out values. For instance, the Multiscalar processor [76] 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 and 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 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 [7]. This is in contrast to our scheme, which also supports a consumer-driven approach, thereby simplifying recovery from squashes. Squashed threads that are restarted simply re-request the value from the appropriate producer.

The Trace processor [69] avoids the last-copy problem in the other way, namely 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, however, the architecture is devoted to providing 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. This includes both producer-initiated and consumer-initiated transactions. Consequently, each processor has 3 extra bits per register called the *Sent (X)* bits. They are set if the corresponding thread has been sent the register in the past. These bits are used as follows. Before we retire a non-speculative thread, we will use the *Sent* bits to ensure that no last-copies of registers are going to be lost. For any such last-copy, the thread will simply write it on the SS bus, so that speculative threads 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 its synchronizing registers (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 2.3.1.1. The logic is replicated for each register as in the case of register availability. At thread retire time, each register can be checked in parallel to check for last copy status.

#### 2.3.1.3 Complexity of the Synchronizing Scoreboard

This section describes the potential impact of the SS mechanism on the processor's cycle time and the additional area that is required for its implementation.

Consider the logic to check for register availability as well as for checking last copy status as described in Sections 2.3.1.1 and 2.3.1.2. The AND-OR logic, that is traditionally implemented as a *carry-propagate-kill* function, and the extra gates to selectively choose the $S$ and the $F$ bits (Figure 2.10) requires 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 logical registers.

It must be stressed that, though conceptually a centralized scheme, the SS hardware is completely decentralized wherein each processor maintain its own copy of the SS. The down side is the added space required by the register file. Indeed, the number of register bits for storing the $V$, $X$, $F$ and $S$ bits is not trivial and requires an additional $3n$ bits, where $n$ is the number of processors on chip (Figure 2.9). For a 4-processor CMP with 64-bit registers, this works to around 12% storage overhead. However, we feel that these are modest hardware requirements when compared to replicating the register sets in each processor [7] or using a centralized global register file [69].

As for the impact on cycle time, if we refer to equation (1) in Section 2.3.1.1, we see that in the worst-case scenario, where all the bits would have to be considered, the delay incurred by the SS logic is dependent on the number of threads supported on the processor. For a CMP based on 4 or 8 processors, this checking can be easily done in a single cycle.

The broadcast bus to communicate values between processors can be implemented with staging buffers that would allow the signal to be driven over the entire length of the chip. This is the reason we had assumed multiple cycles for propagation of the value. We also assume only one additional read and write port for each register file. This allows only one value to be written or read from the bus in a given cycle. But as we show later in our evaluations, this has little impact on performance.

#### 2.3.1.4 Support for Dynamic Issue

In our previous discussion, we did not differentiate between the physical and logical register set. However, when each processor in the CMP is a dynamic superscalar, the logical registers get re-mapped into physical registers. Moreover, the mapping will be different in each processor. Allowing each processor to know the mapping of the remaining processors is impractical. Thus, when a producer writes a register value onto the SS bus, it must put the logical, rather than the physical ID, on the bus. The consumer can then use its local register map to determine the actual physical register that needs to be updated.

Though conceptually simple, there is a problem with this approach. For instance, it is possible that the consumer processor may change the mapping when it creates a new instance of the logical register. We illustrate this with an example shown in Table 2.6. In this code sequence, there is an anti-dependence between instructions (1) and (2), and a dependence on r3 across threads. The renamed instruction, along with the mapping, is also given in the table.

| | Instruction | Renamed Instruction | Mapping | **SS** I ID V |
|---|---|---|---|---|
| (1) | r2=r3+r9 | R6=R10+R11 | r3: R10 | 1 10 0 |
| (2) | r3=r4+r5 | R11=R1+R12 | r3: R11 | 1 10 1 |
| (3) | r5=r3+r4 | R14=R11+R1 | | 0 10 1 |

**Table 2.6** Example to illustrate how renaming may result in wrong updates.

Assume that, when the consumer tries to access logical register r3 (whose physical register is R10) in (1), the value is unavailable. However, due to the dynamic issue policy, the thread can continue instruction issue. When it issues instruction (2), it creates a new mapping for r3. From this point onwards, the logical register r3 gets re-mapped to physical register R11.

Now, consider what happens when the producer thread finally writes the value of r3 onto the SS bus. Let us assume that this happens after the consumer has executed instruction (2) and updated the value of R11. Since the mapping for r3 has been changed, the result would be to wrongly update the physical register R11 rather than R10, thereby destroying the result produced by instruction (2). As a result, when instruction (3) is issued, it will not get the value written by instruction (2). On the other hand, instruction (1) would be indefinitely waiting in the instruction queue for R10 to become valid.

This problem appears in all CMPs that use dynamic superscalars. We are not aware as to how this problem is solved in the Multiscalar processor. As for the trace processor, the live registers are mapped to a centralized global register file [86], so that a producer and consumer would use the same mapping.

We solve this problem as follows: We store the physical register identifier in the SS (call it $ID$ in the figure), along with an *Invalid* bit ($I$) that tells that the physical register is not valid. In our example, this is register R10.

In the example above, when the consumer accesses r3 in (1) and finds it unavailable, it sets the $I$ bit and also stores the mapping of the register in $ID$ (in our case R10). When (2) is issued, it sets the $V$ bit in SS for r3 to denote that the processor now has a valid copy. Any future access to r3 needs to check the $I$ bit and the $ID$, before using

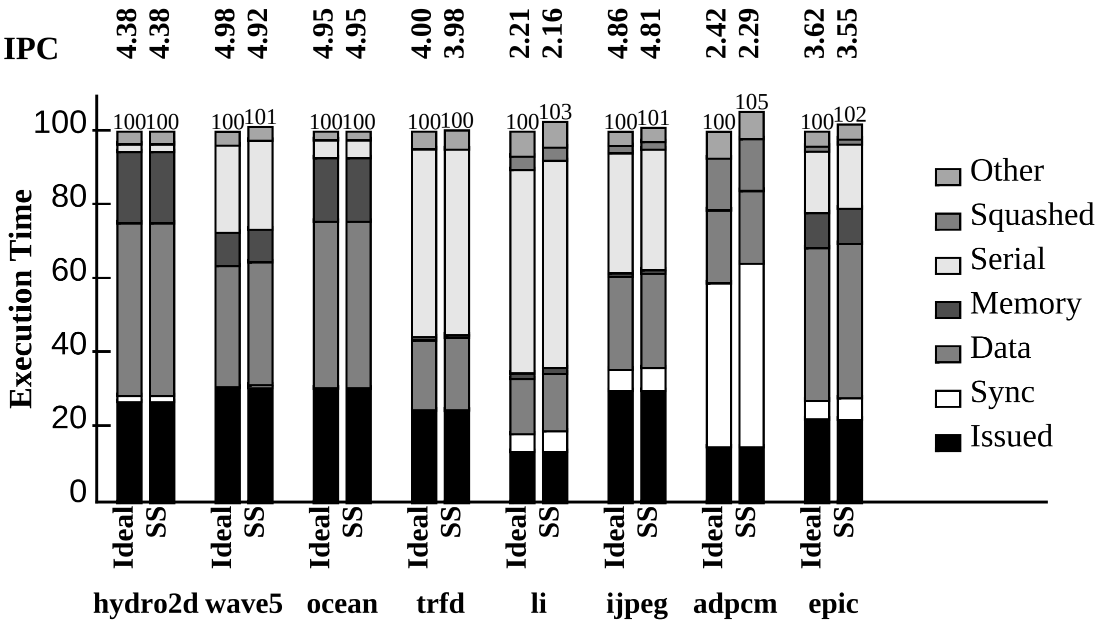



**Figure 2.13** Performance of a CMP based on four 4-issue dynamic superscalar processors, augmented with a SS. The execution time is normalized to the time taken by the *Ideal* environment.

the SS logic. If the $I$ bit is set and $ID$ matches the physical register we want, it implies the register is unavailable. This occurs when (1) tries to check the status of r3. On the other hand, when (3) accesses r3, the physical register R11 does not match the $ID$ value R10. Therefore, instruction (3) can use the copy of r3 generated by the previous instruction (2). Finally, when the producer thread writes r3 on the SS bus, the $ID$ value is used to update the correct physical register, R10, followed by the clearing of the $I$ bit. Simultaneously, the tag for R10 is also driven across the instruction queue enabling instruction (1) to be marked ready for issue.

## 2.3.2 Evaluating the Synchronizing Scoreboard

We now evaluate a CMP augmented with our SS and compare its performance to the *Ideal* environment described in Section 2.2.6. Recall that in *Ideal* there is no communication latency: any producer update is visible to the consumer instantaneously. Figure 2.13 shows the execution time of the eight applications on a CMP with four 4-issue processors under our SS hardware and under *Ideal*. In our simulations, we assume a SS bus with high bandwidth (5 words per cycle), so that we can factor out the effect of contention. Registers are 1-word wide. We will reduce the bandwidth later.

The figure shows that, for all the applications, the performance of SS is very close to that of *Ideal*. This means that the SS introduces very little overhead. This should be compared to Figure 2.8 from Section 2.2.6, which compared *Ideal* to CMPs where all communication occurs via the L2 cache. That plot showed that communication via the L2 cache slows down the applications by an average of 24% (for a 3-cycle L2) and 46% (for a 5-cycle L2). These results, therefore, indicate that register communication is very beneficial.

We note that the fact that there is a 3-cycle latency between processors that are located far apart does not seem to affect the performance to a great degree. This may suggest that most of the time the producer and consumer threads are in adjacent processors of the CMP. This would be consistent with [22], which showed that in nearly 70-80% of the cases, the register instances are used up by the immediate successor thread. Another factor that helps reduce the effect of latency is that producer threads put their values on the SS bus, enabling the consumers' registers to be updated. This avoids unnecessary delay when the consumers finally access the values.

We now change the bandwidth of the SS bus to determine its on performance. Figure 2.14 shows the execution time of the applications for values of bus bandwidth ranging from 1 word/cycle to infinite bandwidth. For each application, the execution time is normalized to that taken when the bandwidth is 1 word/cycle. From the figure, we can see that there is little performance gain in increasing the bandwidth of the SS bus. In fact, an environment with infinite bandwidth is less than 5% faster than one with a bandwidth of 1 word per cycle. Consequently, we suggest a bus no wider than 1 register/cycle.

Overall, therefore, based on these plots and the previous ones, we conclude that support for register-level communication is quite beneficial for CMPs with wide dynamic superscalars. In addition, we have shown that this support can be provided with modest hardware requirements.

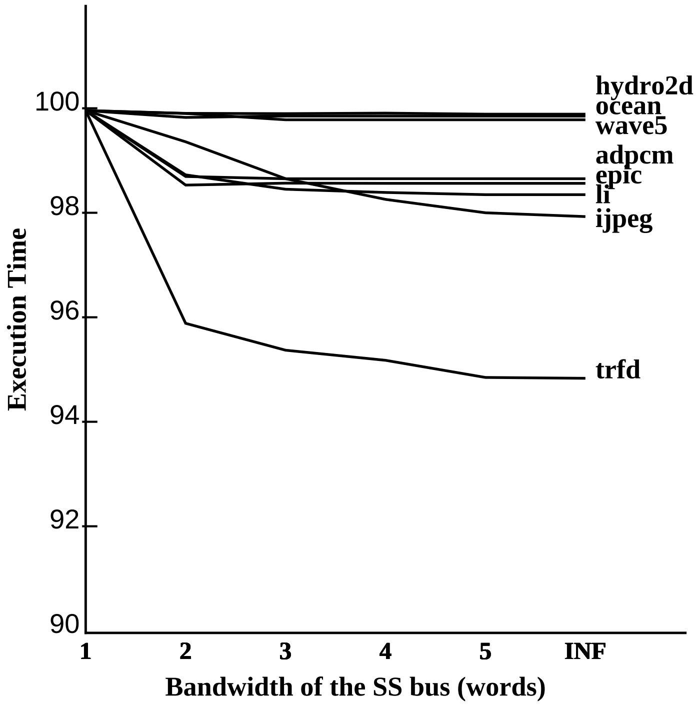



**Figure 2.14** Impact of changing the bandwidth of the SS bus on the execution time of the applications. The execution time is normalized to the time taken with a bandwidth of 1 register/cycle.

## 2.4 Hardware Support for Handling Memory-Level Dependencies

Unlike register dependencies, we do not identify memory dependencies in the binary annotation phase and, therefore, assign the full responsibility to hardware. In this section, we explain the hardware support that is required to identify and recover from memory operations that violate sequential semantics.

When an application runs in a speculative mode, it could generate potentially wrong data. Thus, we need to separate the speculative from the non-speculative data. Also, only the non-speculative thread should be 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 will greatly ease the recovery from mis-speculations and exception handling.

A straightforward approach to satisfy this requirement in the CMP architecture would be to assume a separate storage for each processing unit to store speculative writes, while allowing the non-speculative writes to update the memory subsystem. Furthermore, the writes by the speculative thread should not be released to the memory subsystem until the thread acquires non-speculative status. Finally, hardware support must be present that would detect dependence violations. This would occur when a speculative thread prematurely reads a value before a predecessor has written to the location. If necessary, the hardware must also permit fetching of data from the speculative storage, rather than the memory subsystem, to the appropriate consumer thread. For instance, assume a CMP with four processors in which processor 0 is executing the non-speculative thread, while processors 1, 2 and 3 are executing the speculative threads 1, 2 and 3 respectively. If processor 3 requests a value, it may be present in the speculative storage of its predecessors. Thus, added support is required to determine if the data needs to be fetched from the the speculative storage or from the memory subsystem.

This naturally raises the question as to which part of the memory subsystem should hold the sequential state of the program. If the L1 cache is used to hold the sequential

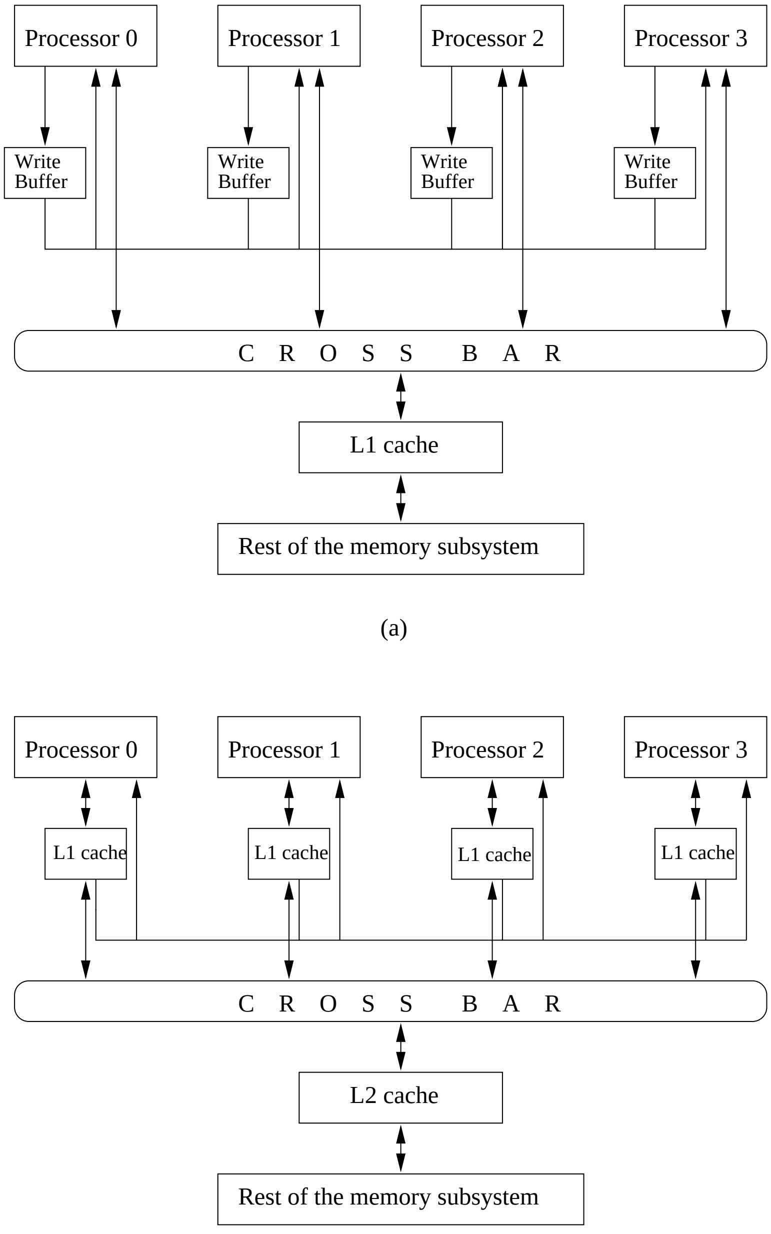



**Figure 2.15** Two approaches for storing the speculative data (a) Shared L1-cache and separate write buffers (b) Private L1 caches for speculative data with a shared L2 cache holding the non-speculative state.

state, it would require all the processing units to share the complete memory hierarchy including the L1 cache, with special storage buffers needed by each unit to store the speculative state. This is illustrated in Figure 2.15-(a). A drawback of this scheme is that it increases the L1 cache latency due to the interconnect between the processors and the L1 cache [53]. In addition, the speculative threads are restricted by the size of the write buffer. If the buffer becomes full, it would require stalling the speculative threads.

Alternatively, we could allow each processor in the CMP to have a private L1 cache, with the rest of the memory subsystem shared by all processors in the CMP. In this case, the private L1 cache holds the speculative state while the L2 cache holds the sequential state of the application. This is shown in Figure 2.15-(b). Unlike the shared L1 cache approach, this approach scales quite well and does not involve complex on-chip hardware such as an interconnect between the processors and the various banks of the L1 cache. Additionally, using the L1 cache to store speculative data provides a much larger storage.

We use this approach in our proposal. We also propose additional hardware support, called the Memory Disambiguation Table (MDT) that identifies dependence violations, whenever they occur. The scheme we propose allows data to be fetched from the appropriate location, either from the L1 cache of a predecessor or from the L2 cache, and also permits data coherence to be maintained in this speculative environment.

In our scheme, each processor in the CMP has a private L1-cache and a shared L2-cache. During speculative execution, speculative threads are in a restricted write-back mode i.e. the L2 cache is not updated. When a dirty line needs replacement, the thread stalls. Although we do not assume one, a victim cache may be used to store the dirty lines, which when full, would stall execution. 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 details on the MDT is explained later in this section.

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

| | | | | |
|---|---|---|---|---|
| Flush (F) | Invalid (I) | Dirty (D) | SafeWrite (SW) | SafeRead (SR) |

**Table 2.7** 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, that tracks memory dependence violation. 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 a dirty line without informing the MDT. When a thread performs a store 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 line 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 loads from the same location. This is done when the load request by a successor is forwarded by the MDT. Thus, if the thread stores again to the same dirty line 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 line without informing the MDT. This bit is set when the thread loads from the location for the first time and is cleared, when a new thread is initiated on that processor. Thus, when a thread reuses valid data in its private L1-cache for the first time, it reads the data, sets the $SafeRead$ bit and also notifies the MDT about the load. Future loads to the same address can be done without informing the MDT as long as the $SR$ bit is set. 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 has to be invalidated in all its successors. As for the predecessors, information needs to be maintained to denote that the location would become stale after the current thread completes. The $F$ bit helps to identify those private L1-cache lines that need to be invalidated across thread initiations, while allowing reuse of the remaining lines. When a thread is initiated on a processor, the new value of the $Invalid$ bit will be: $I_{new} = I_{old} \vee F$. 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 would result in false sharing, and consequently lead to unwanted squashing of threads. So, we maintain information at a word level, except the $Dirty$ bit, which is maintained at the line level.

### 2.4.1 Operations from the Processor Endpoint

This section describes what happens when the following events occur: the processor performs a load/store, a request forwarded by the MDT reaches the processor or a new thread is initiated on the processor.

### Load Operation

```
if read miss (I = 1) then
  if cache line pending then
    if thread is speculative AND SR = 0 then
      inform MDT about the load operation i.e. send address;  SR <- 1
    endif
    wait for pending cache line reply
  else  /* cache line not pending */
    get replacement block
    if thread is speculative then
      if all replacement lines dirty then
        block thread /* speculative threads cannot replace dirty lines */
      endif
      send read miss request to MDT; SR <- 1
    else
      send read miss request to L2 cache /* non-speculative loads need not go thru MDT */
    endif
  endif

if read hit (I = 0) then
  if thread is speculative AND SR = 0 then
    inform MDT about load operation (send address); SR <- 1
  endif
  read cached value
endif
```

Note that on a read hit, only speculative threads check the $SR$ bit and inform the MDT as they may perform unsafe loads. Also, only the first unsafe load request for the address reaches the MDT. Subsequent loads from the same address need not inform the MDT.

### Store operation

```
if write miss (I = 1) then
  if cache line not pending then
```

```
    get replacement block
    if thread is speculative AND all replacement lines dirty then
      block thread /* speculative threads cannot replace dirty lines */
    endif
  endif
  I <- 0; SW <- 1; SR <- 1 /* can safely read value in future without informing MDT */
  if thread is speculative then
    update word in L1 cache only /* write back mode */
    D <- 1 /* set dirty bit */
  else
    update word in L1 as well as L2 cache /* write-thru mode for non-speculative stores */
  endif
endif

if write hit (I = 0) then
  if SW = 0 then
    inform MDT about the store operation i.e. send address
    SW <- 1; SR <- 1 /* can safely read value in future without informing MDT */
  endif
  if thread is speculative then
    update word in L1 cache only /* write back mode */
    D <- 1 /* set dirty bit */
  else
    update word in L1 as well as L2 cache /* write-thru mode for non-speculative stores */
  endif
endif
```

### Forwarded Request

```
supply data
SW <- 0 /* future writes would cause squashing */
```

### Thread Initiation

```
I = I + F /* selective invalidation of cache lines */
F <- 0; SW <- 0; SR <- 0 /* clear all bits */
```

### 2.4.2 Memory Disambiguation Table

The disambiguating mechanism is performed with the help of a memory disambiguation table (MDT). The table maintains entries on a cache line basis. As with the cache lines, we also maintain information on a word basis. For each word, there is a *Load (L)* and *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 2.8 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 performance is not affected by the latency of the MDT. Moreover, 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 | 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 2.8** Memory Disambiguation Table.

#### Load Operation

A non-speculative thread need not perform any special action and its loads proceed normally. A speculative thread, however, may perform unsafe loads and has to access the table to find an entry that matches the load's address. If an entry is not found, a new one is created and the $L$ bit corresponding to the thread is set. However, if an entry is already present, it is possible that a previous thread could have speculatively updated the location. Therefore, the store bits are checked to determine the id of 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 $S_0S_1S_2$ 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. 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 $W$ 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.

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 processor 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 for an entry corresponding to the address being updated. If the entry is not found, speculative threads create a new entry with their $S$ bit set.

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 2.4 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 2.8 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. Otherwise, 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 and also conservatively to the non-speculative thread as it doesn't have any entries in the MDT. This would result in their corresponding cache lines to remain valid only during the course of execution of their current task. Also, an invalidate message is sent to all successors (up to, but not including the 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 entries are available, the thread stalls until one becomes available. Since a non-speculative thread can continue to perform load and store operations without an entry in the MDT, their 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. Nevertheless, the size of the MDT need not be large for optimal performance. This is discussed in Section 2.6.2.

### 2.4.3 What Happens on a Thread Initiation or a Squash?

At the point of thread initiation, all cache lines whose $F$ bits are set are invalidated. The $F$, $SW$ and the $SR$ bits are also 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 would be initiated on those processors and all the actions described above would be performed. The only difference is that there may be cache lines that may remain valid in some processors whose value was forwarded from the private cache of another processor. Consequently, we need to invalidate lines which contain data forwarded from squashed threads. This is achieved by adding a $Forward$ ($FD$) bit to each cache line. This bit is set when the MDT informs the processor performing a load that the data is forwarded from another processor's L1-cache. In addition, dirty lines also need to be invalidated. Thus, for squashed threads, the $FD$ and the $D$(dirty) bits are used for deciding the validity of the cache line i.e $I_{new} = I_{old} + D + FD$.

Overall, the additional overhead to each cache line is minimal. The coherence protocol is like that of a conventional directory-based NUMA multiprocessor. Finally, having a separate dependence tracking hardware (MDT) leaves room for further improvements, such as enforcing memory dependencies, thereby disallowing premature loads from occurring.

### 2.4.4 Comparison with Other Schemes

There have been other schemes that have been proposed earlier or concurrently with our proposal that permits speculation at the memory level. The "address resolution buffer" (ARB) [23] is one such scheme that uses the L1 cache to store the sequential state of the program. The ARB is a specialized hardware that not only provides storage for speculative data, but also identifies any dependence violations. However this scheme forces all memory accesses from each processing unit to necessarily be streamed through the ARB before they can reach the L1 cache. Furthermore, this is a very centralized scheme and, as mentioned earlier, would hurt performance due to the added latency for L1 cache accesses. There have three other schemes that allow the CMPs to have their private L1 cache. Olukotun et al [56] have proposed a simple hardware scheme, where they assume extra buffers, rather than the L1 cache, to hold speculative data. Though this simplifies the protocol, the buffer may become full and stall the speculative threads. Mowry et al [78] outline a scheme that would allow the L1 cache to buffer the speculative state. However, their description lacks detail and a precise cache coherence scheme for handling data dependencies is absent. The Speculative Versioning Cache (SVC) [24] is a detailed architectural proposal for handling dependencies. However, unlike our scheme which uses a directory-based approach, this scheme uses snooping bus-based coherent caches for detecting memory dependence violations.

## 2.5 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 also show the performance of a conventional 4-issue processor to see the extra gain from the multiple processor in the CMP.

We assume an aggressive dynamic superscalar core for the CMP and the 4-issue/12-issue/16-issue processors. The core is modeled on the lines of the R10K as in Section 2.2.3 The parameters for the 4-/12- and 16-issue processors are shown in Table 2.9. The memory system is the same as that described in Section 2.2.3. The default parameters for the MDT and SS bus are shown in Table 2.10. We will vary them later.

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

| 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 2.9** 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 2.10** Characteristics of the MDT and SS bus.

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 2.11.

| 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 2.11** Characteristics of the memory hierarchy. Latencies refer to round trips.

**Simulation Approach**

Our simulation environment and method of collecting statistics are the same as described in Sections 2.2.4 and 2.2.5. We use MIPS binaries for 4 integer (*compress*, *ijpeg*, *mpeg* and *eqntott*) and 4 floating-point (*swim*, *tomcatv*, *hydro2d*, and *su2cor*) 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*.

## 2.6 Evaluation

We compare our `4x4-issue` CMP to the aggressive superscalars in Section 2.6.1, and then evaluate the requirements imposed by the MDT for handling memory dependencies.

### 2.6.1 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 2.6.2 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 [59] have argued that the register bypass network will dominate the cycle time of future high-issue processors. Based on their observations, the `4x4-issue` CMP would have a tremendous cycle time advantage over the 12-issue and 16-issue dynamic superscalars, when implemented in advanced technologies lesser than $0.18\mu$.

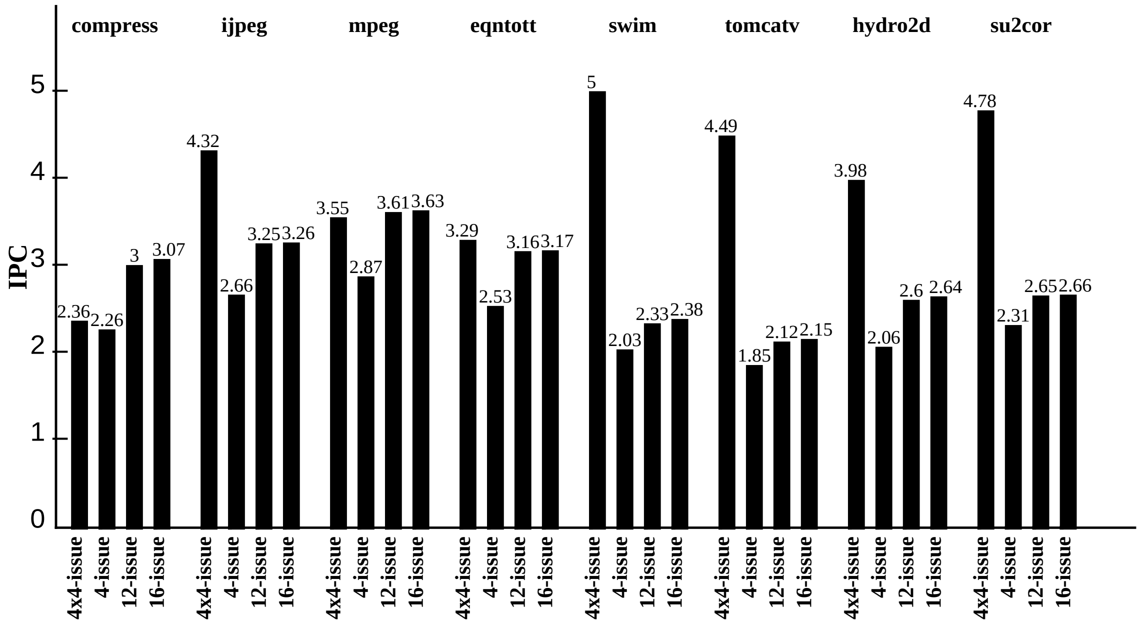



**Figure 2.16** Comparing the IPC for the applications under the 4x4-issue CMP and a 4-issue, 12-issue and 16-issue conventional superscalar. No cycle time differences are considered.

Figure 2.16 shows the IPC for the eight 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 2.17 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.

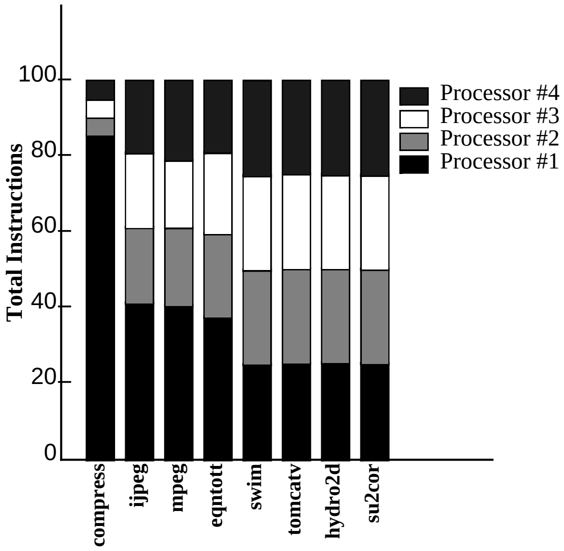



**Figure 2.17** Fraction of instructions issued by each of the four processors in the CMP.

To understand the two charts, we examine the floating-point applications first. From Figure 2.17, we see that they have much speculative parallelism as there is a big difference in IPC between the 4-issue conventional superscalar and the 4x4-issue CMP. This is because these applications are fully loop-based, with most of them having few or no loop-carried dependencies. 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 time, can independently issue instructions without being affected by dependencies 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 2.16, the IPC of the `4x4-issue` CMP is over twice that of the superscalars.

The integer applications behave differently. According to Figure 2.17, the amount of speculative parallelism extracted by the binary annotator. The difference in IPC between a conventional 4-issue superscalar and a 4x4-issue CMP is not as large as that of floating-point benchmarks. 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 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 2.16 shows that they have around 25% higher IPCs for this application.

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 [78]. 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 [59], 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.

### 2.6.2 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 2.18 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 2.19 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,

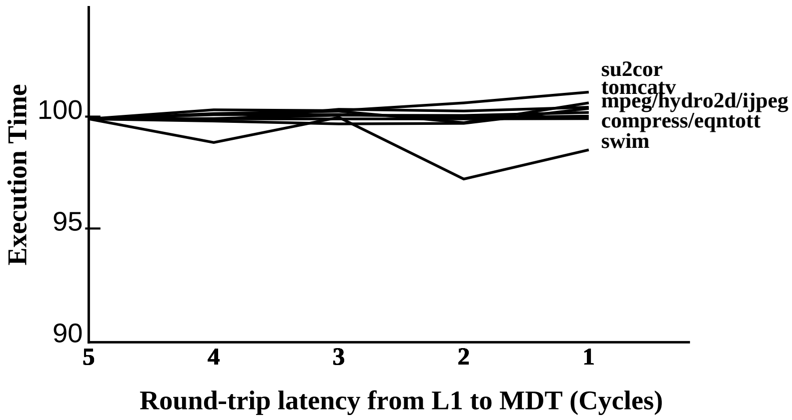



**Figure 2.18** Effect of the L1 to MDT latency.

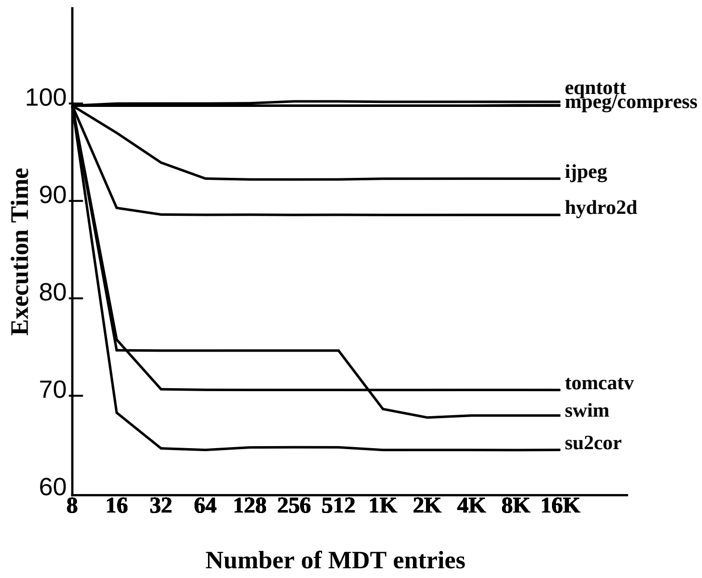



**Figure 2.19** Effect of the MDT size.

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.

### 2.6.3 Shared L1-cache vs. Private L1-caches

Recall that in the CMP, each processor has a private L1-cache, with the L2-cache shared by all the processors. This allows the L1-cache to serve as a repository for speculative updates, while the L2-cache gets updated only when the speculation is proved correct. Though this design is highly scalable, it comes at the cost of reducing the effective L1-caching space. For instance, data may be replicated in the L1-caches of different processors. An obvious solution, that mitigates this problem, is to have a shared L1-cache. This would not only prevent data duplication, but would also allow beneficial prefetching, when data required by a processor would have already been loaded by another processor. However, this scheme requires a more complex on-chip hardware in terms of a large interconnect between the processors and the various banks of the L1-cache. Moreover, the chances of bank conflicts would be higher as any memory access by any processor would reach the shared L1-cache. Finally, since the shared L1-cache can no longer be used to store speculative data, each processor has a large write buffer for storing speculative updates.

In this section, we compare the two schemes. We assume a shared L1-cache and evaluate it with the performance of private L1-caches. The parameters are as specified in Table 2.11. Figure 2.20 shows how the applications fare under the shared cache system. The execution time is normalized to that of the private L1-cache system. We also evaluate the execution time when the interconnect latency between the processors and the multiple banks of the L1-cache is increased from 1 to 3 cycles. We increase the latency for the shared cache and not for the private ones because the latter are simpler to design. From the figure we see that, even when a perfect interconnect is assumed, the performance of the applications under the shared L1-cache is worse than the private case. We see that contention at the banks nullifies any advantage that the shared L1-

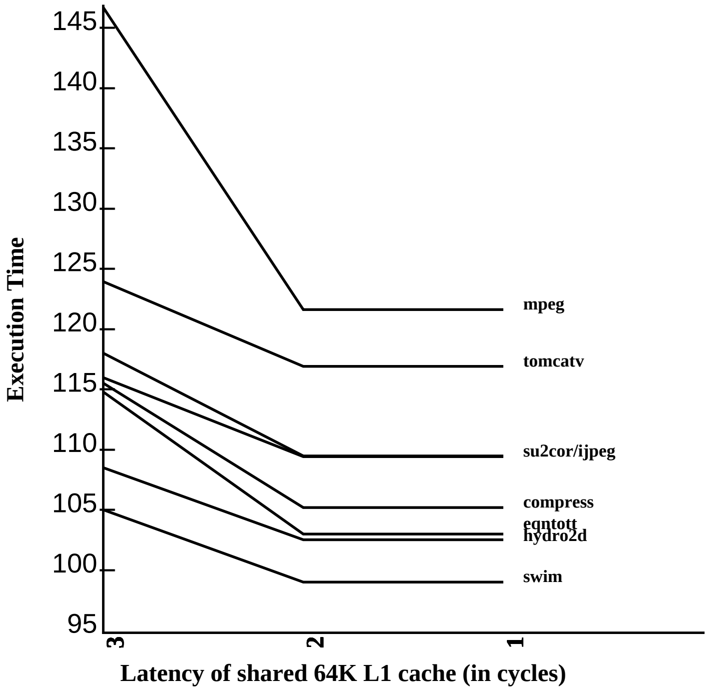



**Figure 2.20** Evaluating the shared 64K L1-cache and the impact of its interconnect latency on execution time. The execution time is normalized to the time taken when each of the processor in the CMP has a private 16K L1-cache with a 1 cycle latency.

cache offers. As a result, applications show a performance degradation of up to 25% even without including the interconnect latency. Overall, we conclude that a CMP with private L1-caches is a better alternative to a shared cache system, not only from the design perspective but also from a performance viewpoint.

## 2.7 Extra Support for Speculation

In this section, we summarize the additional support that is required for running applications speculatively on a CMP.

First, we need hardware support that predicts the starting address of the next thread and initiates it on a processing unit. The predicted address is verified when a thread completes and if necessary, threads are squashed. Note that, in our current work, we are restricting the threads to loop iterations only. Hence, the starting point of the next thread remains the same as the current one viz. the beginning of the loop iteration. Squashing occurs when iterations beyond the last one are initiated. At the time a thread is initiated, the hardware also initializes the synchronizing scoreboard entries with the appropriate bits. At the same time, the cache lines are also initialized. This hardware is along the lines of the sequencer unit in the Multiscalar [76].

Register level communication requires a simple broadcast bus and an additional read/write port to the register file. In addition, the scoreboard needs to be augmented with extra bits. As mentioned in Section 2.3.1.2, the additional storage overhead is $3n$ bits where $n$ is the number of processors on-chip. Furthermore, additional logic is also required to check for register availability as specified in Figure 2.10. Assuming that this is replicated for each register, this adds a minor overhead of a few hundreds of gates.

Finally, as was shown in Section 2.6.2, handling memory dependencies requires hardware support in terms of a relatively small MDT along with the related logic that checks for memory dependence violations. Overall, we require only a modest amount of hardware for enhancing a CMP so as to execute applications in a speculative fashion.

## 2.8 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 chapter, 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. For speculative CMPs that are based on high-issue dynamic superscalar processors, communication latency is one critical factor in deciding performance. We have shown that relying only on the memory subsystem for communication between processors degrades the performance and that hardware support for register-level communication is required. We also proposed a hardware scheme that augments a CMP and enables it to perform communication and synchronization at the register level. Overall, our evaluation shows that this architecture delivers high performance for sequential binaries with very modest hardware requirements.

# CHAPTER 3

# A CLUSTERED APPROACH TO MULTITHREADING

Today's superscalar processors focus all their resources in extracting ILP from a single thread in the application. In their quest for extracting ILP more effectively, these processors have additional hardware such an aggressive branch predictor and a large associative instruction window that enable them to look farther in the instruction stream. However, branch mispredictions and true dependencies in the program limit the effectiveness of this traditional approach of exploiting ILP in the program. Some proposals continue to focus on extracting ILP from only one thread but resort to speculating data values to overcome the limits posed by true dependencies [43, 75] However, there is also much interest in alternate architectures that exploit thread-level parallelism. Moreover, with future applications likely to be multithreaded, this is a very promising approach.

The chip-multiprocessor (CMP) is one such approach that naturally lends itself to multithreaded application. There have been several CMP-based approaches that have been proposed[21, 30, 69, 76, 78, 82]. However, in all these proposals, the architecture is configured such that the chip resources are partitioned in a fixed manner between the threads. In other words, resources have a fixed-assignment (FA) in this architecture. These fixed-assignment architectures (FA) risk wasting resources when one of the threads stalls due to hazards or when there is a lack of threads in the application.

In contrast to the FA approach, the simultaneous multithreading (SMT) approach is flexible and allows resources to be shared by all threads. This is especially useful when only one thread is available, as would be the case in serial sections when running parallel applications. Here, most of the chip resources in the FA architectures would be unutilized. On the other hand, the SMT can devote all its resources in speeding up this section. Taking Amdahl's law into account, this results in an improved performance for the application. Although basic simultaneous multithreading (SMT) is helpful in this respect, as discussed in chapter 1, the structure is so centralized that it may slow down the clock frequency. However, given that the amount of thread- and instruction-level parallelism of applications varies widely, the flexibility of the SMT architecture is very valuable.

A very intuitive alternative to these two architectures is a hybrid of the FA and the centralized SMT approaches, namely a clustered SMT architecture. In this architecture, the chip has several independent processing units, with each unit having the capability to perform simultaneous multithreading. Given that the effort to enhance a conventional dynamic superscalar to perform simultaneous multithreading is small [83], a low-issue SMT processor is feasible. A clustered SMT architecture would be a simple design as it would involve replicating several such low-issue SMT processors on a die. Though it might be argued that this separation of processing units would restrict the sharing of resources, it may not necessarily be true. So, in this chapter, we study whether this architecture is still able to extract most of the benefits that can be achieved with the fully centralized SMT approach.

In this chapter, we explore such a clustered SMT architecture when running parallel applications. We show the performance of both a low-end machine with only one processor chip to run the multiple threads, as well as a high-end machine with several processor chips working on the same application. The clustered SMT architecture often delivers a performance close to that of a fully dynamic centralized SMT. Even this limited form of simultaneous multithreading results in high performance due to the ability to adapt to the specific thread- and instruction-level parallelism of the application. Though we evaluate this architecture only for parallel applications, it can also be shown that the conclusions hold for applications that are executed speculatively parallel on the above architectures,

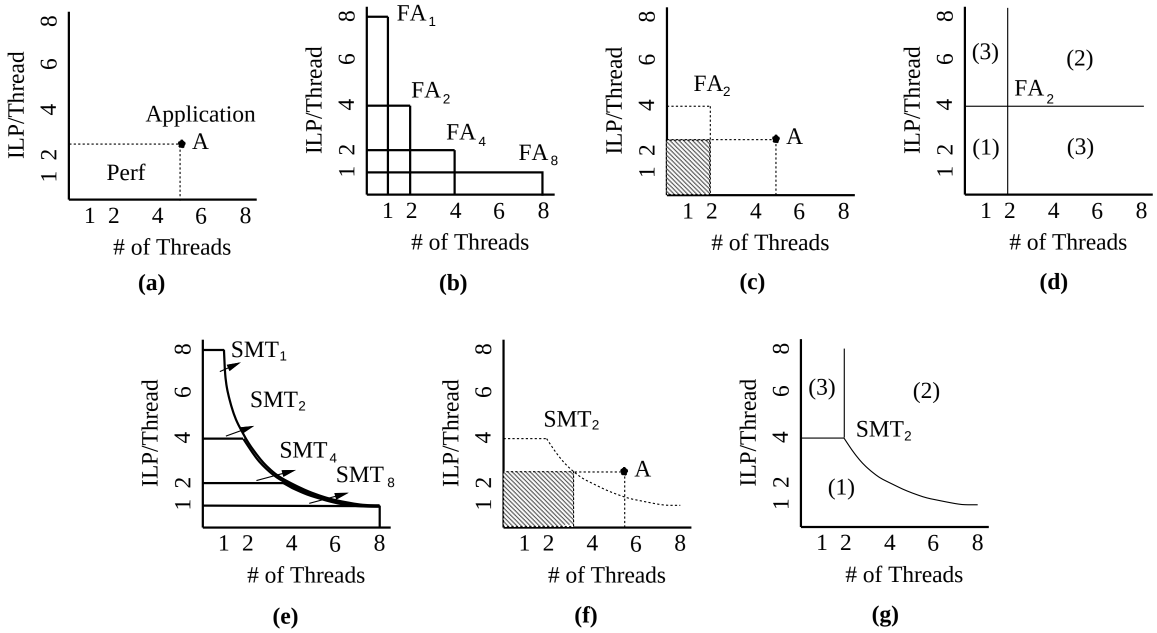



**Figure 3.1** Model of parallelism.

with the extra speculation hardware as discussed in Chapter 2 [36]. Consequently, the hybrid architecture represents a good performance-complexity design point.

This chapter is organized as follows: Section 3.1 presents a model of parallelism that relates instruction- and thread-level parallelism; Section 3.2 describes the architectures analyzed; Section 3.3 discusses the evaluation methodology; Section 3.4 evaluates the architectures under several environments; and finally, Section 3.5 concludes the chapter.

## 3.1 A Model of Parallelism

Before performing any architectural analysis, we propose a simple model of parallelism to gain insight into the performance of applications on processors that exploit both thread- and instruction-level parallelism. This model is simple and does not take into account any cycle-time variations that may occur between different designs. It just serves to illustrate the potential of each architecture. The model uses a chart that graphs the number of threads versus the ILP per thread (Figure 3.1-(a)). If we consider average conditions for simplicity, a given application $A$ will fall somewhere in this chart. The area of the rectangle formed by the axes and the $X$ and $Y$ coordinates of $A$ is proportional to the performance that can be extracted from this application.

It must also be mentioned that the ILP and number of threads that we use in our model are the upper bound for the application. This is because, an application could trade off threads to exploit ILP and vice versa. For instance, an application that consists of a loop with all iterations that are independent could be executed by independent threads. Alternatively, loop unrolling or software pipelining may be performed to exploit ILP across iterations. Consequently, in our model, such an application would be represented as having a high ILP as well as number of threads. Moreover, during the course of the application the ILP and the number of threads may fluctuate. For simplicity, we assume a fixed point that may be considered an average value for the application.

For the data points, we consider the upper bound for the ILP and thread parallelism in the application. This is because one could tradeoff ILP for more thread parallelism and vice versa. For instance, independent loop iterations enables either a high thread parallelism or a high degree of ILP, especially when loop unrolling or software pipelining is performed.

Consider now the chart from a processor architecture point of view. We restrict ourselves to 8-issue processors. The different fixed assignment architecture (FA) configurations are shown in Figure 3.1-(b). For example, a FA with 8 single-issue processors is shown as the $FA_8$ rectangle. The rectangle means that any application that falls inside it will be fully exploited by the processor. (i.e the maximum ILP and the thread parallelism present in the application is fully exploited by it) The area of the rectangle is proportional to the maximum performance that can be delivered by the processor. In the figure, we show boxes for a FA processor with four 2-issue processors ($FA_4$), two 4-issue processors ($FA_2$), and a conventional 8-issue superscalar ($FA_1$). Consider application $A$ again (Figure 3.1-(c)). $FA_2$ is unable to fully exploit the application. The performance delivered is proportional to the shaded area only, not to the whole rectangle under vertex $A$. In addition, the processor is under-utilized. While it can deliver as much performance as the size of its box, it is only delivering the performance of the shaded area. Overall, we can distinguish 3 regions depending on the relative position of the architecture and application boxes (Figure 3.1-(d)). If the application falls into region (1), the application is fully exploited and the processor is under-utilized. This means that the maximum performance for that application is achieved. In region (2), the application is under-exploited and the processor is fully utilized. This means that the processor is functioning at its optimum. Finally, in region (3), the application is under-exploited and the processor is under-utilized. Since we want to utilize the processor completely, we want applications to fall in (2). We call region (2) the *optimal* region.

SMT processors and clustered SMT processors can be represented as rectangles with the lower left vertex in the (0,0) coordinate and the opposite vertex sliding along a hyperbola (Figure 3.1-(e)). We consider SMT processors that can handle up to 8 threads. Consider first a centralized SMT processor. This is simply an 8-issue 1-cluster SMT processor that can issue instructions from any of the 8 threads. We refer to it as $SMT_1$, where the subscript 1 stands for the number of clusters in the SMT processor. This processor has the flexibility to adapt to the application's demands and appear as a $FA_1$, $FA_2$, $FA_4$ or a $FA_8$ processor. Furthermore, it can also appear, for example, as a FA with an average of 5 processors and an average issue width of 1.6 per processor. Since the product of processors and issue width per processor is constant, we follow the hyperbola shown in the figure. The rectangle slides its upper right vertex along the hyperbola. The maximum performance that can be delivered is constant, since the area of the rectangle is constant, but the rectangle can move and *adapt to the demands of the application*. The other, more limited clustered SMT processors are also represented by the same moving rectangle. However, their rectangle cannot follow the hyperbola all the way. Instead, it stops at a certain *(X, Y)* coordinate on the hyperbola and cannot move to higher $Y$ coordinates. For example, a clustered SMT with two 4-issue processors ($SMT_2$), where each cluster can handle 4 threads, stops at (2,4). This is because it cannot exploit ILP in a thread that is higher than 4. In the figure, we marked this limitation with a horizontal line at Y = 4.

Consider now application $A$ running on the $SMT_2$ processor (Figure 3.1-(f)). The shaded area is the performance delivered. The shaded area is larger than the one in Figure 3.1-(c). Therefore, the clustered architecture extracts more performance from the same application than the FA processor. We distinguish 3 regions (Figure 3.1-(g)). If the application falls into (1), the application is fully exploited and the processor is under-utilized. In region (2), which is the optimal region, the application is under-exploited and the processor is fully utilized. In region (3), the application is under-exploited and the processor is under-utilized.

From this model, therefore, we can see that SMT and clustered SMT processors are likely to deliver more performance than FA processors. The reason is that the optimal region for the former two processors is a superset of the optimal region for the FA processors.

## 3.2 Architectures Evaluated

In this section, we describe the different architectures evaluated.

### 3.2.1 Base Superscalar Core

We assume an aggressive dynamic superscalar core as described in Section 2.2.3 as the building block for all the architectures considered. The caches are non-blocking with up to 32 outstanding loads allowed, with full load bypassing enabled.

### 3.2.2 Centralized and Clustered SMT Architectures

The fully centralized SMT architecture is on the lines of [83]. It can support up to 8 threads and can issue up to 8 instructions per cycle (Table 3.1). The instruction fetch unit is shared by all threads, with each thread having a program counter. The fetch unit fetches instructions from a different thread every cycle in a round-robin fashion. In a given cycle, up to 8 instructions can be fetched. Instructions from different threads

| Processor Type | Number of Clusters, Max. IPC/cluster | Threads per Cluster [Chip] | Number of Functional Units (int/ld-st/fp) per Cluster [Chip] | Entries in Instruction Queue & Reorder buffer per Cluster [Chip] | Number of Renaming Registers (int/fp) per Cluster [Chip] |
|---|---|---|---|---|---|
| $FA_8$ | 8, 1 | 1 [8] | 1/1/1 [8/8/8] | 16 [128] | 16/16 [128/128] |
| $FA_4$ | 4, 2 | 1 [4] | 2/2/2 [8/8/8] | 32 [128] | 32/32 [128/128] |
| $FA_2$ | 2, 4 | 1 [2] | 4/4/4 [8/8/8] | 64 [128] | 64/64 [128/128] |
| $FA_1$ | 1, 8 | 1 [1] | 6/4/4 [6/4/4] | 128 [128] | 128/128 [128/128] |
| $SMT_4$ | 4, 2 | 2 [8] | 2/2/2 [8/8/8] | 32 [128] | 32/32 [128/128] |
| $SMT_2$ | 2, 4 | 4 [8] | 4/4/4 [8/8/8] | 64 [128] | 64/64 [128/128] |
| $SMT_1$ | 1, 8 | 8 [8] | 6/4/4 [6/4/4] | 128 [128] | 128/128 [128/128] |

**Table 3.1** Description of the different types of architectures evaluated.

are held in a common 128-entry associative instruction window from where they may be issued in any order. Finally, instructions are committed on a per-thread basis. Although this is a very flexible architecture, resource centralization can make it quite complex to design. Specifically, one of the major bottlenecks in the architecture is the large interconnection network between the functional units and the register file. Part of this network, namely the bypass network, may actually determine the clock cycle-time of high-issue processors [59].

### 3.2.3 Fixed Assignment Architectures

To alleviate the bypass delay problem and also reduce the resource centralization, we partition the SMT processor into several clusters. For instance, $SMT_2$ has 2 clusters where each cluster is a SMT processor that supports 4 threads. Each cluster has its own fetch unit, with a thread capable of fetching up to 4 instructions/cycle. The ability to bypass across clusters (with an additional delay) could also be provided, though we do not assume one for our evaluations. Overall, unlike in the centralized scheme, no resource sharing is done across clusters. Therefore, with a single thread, this processor can achieve at most 4 IPC. Similarly, the $SMT_4$ processor has 4 clusters, each supporting 2 threads and able to issue 2 instructions per cycle (Table 3.1).

Note that the cycle-time of these clustered architectures is much smaller than that of the centralized SMT. Indeed, Palacharla et al [59] show that there would be over a difference of 25% in cycle-time between an 8-issue processor and a 4-issue processor when using $0.18\mu$ technology, with the difference becoming even larger when moving to much finer implementations. However, for our experiments, we ignore this effect and compare the performance assuming the same cycle-time.

We consider 3 variations of the FA architecture. In this architecture, each individual thread runs in its own cluster using its own set of resources. The first type ($FA_2$) is a processor with two 4-issue clusters. Each cluster handles a single thread, for a total of 2 threads per processor. The second type ($FA_4$) is a processor with four 2-issue clusters. Each cluster handles a single thread, for a total of 4 threads per processor. Finally, $FA_8$ is a processor with eight 1-issue clusters, for a total of 8 threads/processor. In terms of cycle-time and space requirements, $SMT_2$ is similar to $FA_2$, while $SMT_4$ is similar to $FA_4$. This is because, unlike in a 8-issue superscalar, the register bypass delay does not decide the cycle-time in a 4-issue superscalar [59].

Details on the configuration of the various processor architectures are given in Table 3.1.

### 3.2.4 Memory Subsystem & Multiprocessor Configuration

Typically, each cluster in a processor would have its own private primary cache and share the secondary cache. In our work, however, we wanted to avoid the results being influenced by different memory hierarchies in different processors. Consequently, we choose a shared primary cache for all our configurations. The characteristics of the cache hierarchies are shown in Table 3.2.

In our simulations, we model contention in great detail. The no-contention latency of round-trip accesses to different levels of the memory hierarchy is shown in Table 3.2. The remote memory and remote cache latencies are applicable only when we simulate a machine with multiple processor chips. The instruction misses in the parallel applications used are minimal. Consequently, we assume a perfect instruction cache.

| Parameter | Value |
|---|---|
| [L1 / L2] cache size (Kbytes) | [64 / 1024] |
| [L1 / L2] cache line size (Bytes) | [64 / 64] |
| [L1 / L2] cache associativity | [2-way / 4-way] |
| [L1 / L2] cache fill time (Cycles) | [8 / 8] |
| Number of banks in [L1 / L2] cache | [7 / 7] |
| [L1/ L2] cache Read or Write occupancy | [1 / 1] |
| L1 latency (Cycles) | 1 |
| L2 latency (Cycles) | 10 |
| Local memory latency (Cycles) | 40 |
| Remote memory latency (Cycles) | 60 |
| Remote L2 latency (Cycles) | 75 |

**Table 3.2** Characteristics of the memory hierarchy. All latencies correspond to a contention-free round trip.

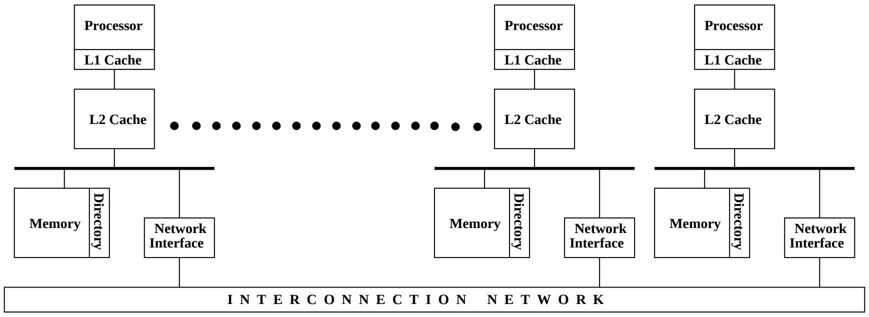



**Figure 3.2** Multiprocessor configuration.

For the low-end machine, we model a simple workstation. For the high-end machine, we model a scalable shared-memory multiprocessor similar to DASH [40] as shown in Figure 3.2. Each node has a portion of the global shared memory and directory. The remote latencies, specified in Table 3.2, are low because we only model a 4-node machine.

## 3.3 Evaluation Environment

The simulation environment is built upon the MINT [88] execution-driven simulation environment. For a parallel application, we generate as many threads as are required by the processor. Thus, the application runs in sequential mode when executing on a conventional superscalar ($FA_1$) where only one thread is created. For $FA_8$ as well as for SMT processors, 8 threads are created.

We choose six applications. Three applications are SPEC95 benchmarks, namely, *swim*, *tomcatv* and *mgrid*; one is a NASA7 kernel, *vpenta*, and the remaining two are SPLASH-2 applications [91], namely *fmm* and *ocean*. The two SPLASH-2 applications are explicitly-parallel programs written in C and use ANL m4 macros [46] for parallel constructs. The SPEC95 benchmarks and the NASA7 kernel are sequential programs written in Fortran. We use the Polaris automatic parallelizing compiler [4] to identify parallel sections of the code. Polaris uses several techniques, such as inter-procedural symbolic program analysis, scalar and array privatization, symbolic dependence analysis, advanced induction and reduction variable recognition and elimination, to parallelize a Fortran application. In these Fortran applications, little or no programmer modification of the sequential code is required. The default data set was used for all applications. However, the number of time steps was reduced to let the simulations complete faster. The statistics include both serial as well as parallel sections of the applications.

### 3.3.1 Statistics Collection

We gather detailed statistics on an issue slot basis. For each processor, we scan the entire instruction window every cycle and record the type of hazard faced by each instruction that is unable to issue. At the end, the wasted slots are divided proportionally among the different types of hazards. The different categories of hazards are: lack of functional units (*structural*), memory access (*memory*), data dependencies (*data*), branch mispredictions (*control*), spinning on barriers or locks (*sync*) and no instructions for a thread in the instruction window (*fetch*). There is also an *other* category for slots wasted due to instructions squashed on a branch misprediction or when there is a stall due to lack of renaming registers. Finally, the useful instruction slots are grouped under *useful*.

## 3.4 Evaluation

We now evaluate the performance of the clustered SMT, FA, and centralized SMT architectures. In our evaluation, we model two environments, namely a low- and high-end machine. In the former, there is only one processor chip to run the multiple threads, while in the latter, there are several processor chips working on the same application, each running the multiple threads. Section 3.4.1 considers the clustered SMT and the FA architectures, while Section 3.4.2 considers the clustered and centralized SMTs.

### 3.4.1 Clustered SMT and FA Architectures

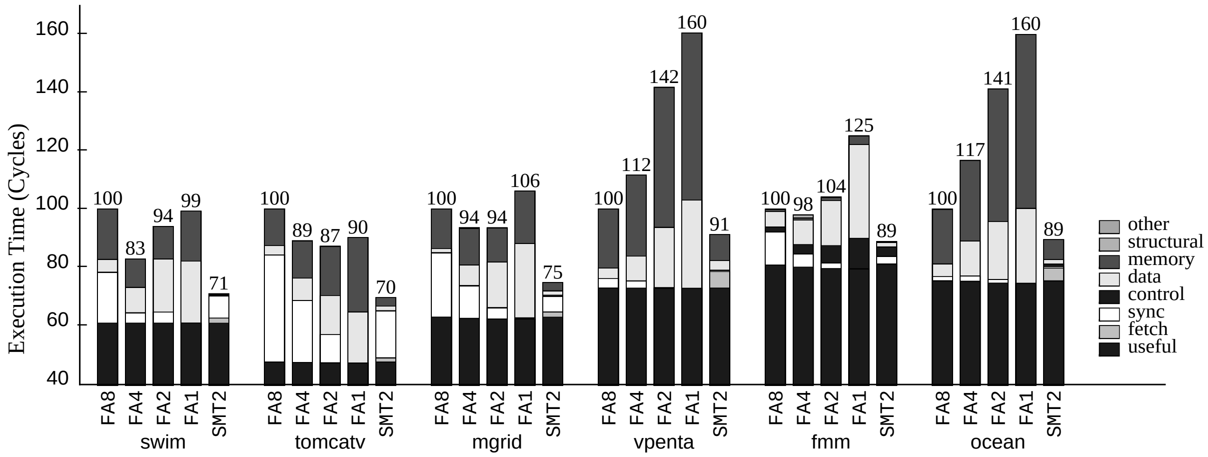



**Figure 3.3** Comparing FA processors to a clustered SMT for a low-end machine. The chart assumes the same cycle-time for all the architectures.

Figure 3.3 shows the execution time of the applications on the FA processors and the $SMT_2$ clustered processor. We have selected $SMT_2$ as the representative for the clustered SMT architectures. The reason for this choice will become obvious in the next section. Figure 3.3 corresponds to a low-end machine environment. For each application, the bars are normalized to those of $FA_8$. Recall that, in all the charts, we measure the execution time in number of cycles. We will take into account the actual cycle time later.

We start by comparing the FA processors. As the number of processors per chip decreases, the contribution of the *sync* hazard steadily decreases, while the *data* and *memory* hazards steadily increase. The contribution of *sync* decreases for two reasons. First, with fewer threads, the relative impact of the serial section of the code or the load imbalance decreases. Secondly, the thread that executes the serial section can speed it up more because it can exploit more ILP. Unfortunately, the benefits of the higher issue per thread provide diminishing returns. Soon, *data* and *memory* hazards appear.

In each application, the bars often take a `U`-shape as we increase the issue width of the processors in the FA processors from one (in $FA_8$) to eight (in $FA_1$). The configuration with the lowest execution time or *sweet-spot* depends on the application. When the application is highly parallelizable, there is little cost involved with using many threads. In this case, $FA_8$ has the minimum execution time. *Vpenta* and *ocean* belong to this class of applications. However, when the application has low levels of parallelism, the benefits of exploiting parallelism across threads are not as high as those of exploiting ILP. Therefore, fewer wider-issue processors will perform better. For example, the $FA_4$ is the best for *swim* and *fmm*, while $FA_2$ is the best for *tomcatv* and *mgrid*. Overall, therefore, no FA processor is clearly the best.

We now consider the clustered $SMT_2$ processor. From the figure, we see that it takes the fewest cycles for all the applications. In addition, it delivers the most stable performance: on an average, it takes 13% lesser execution cycles than the best FA processor for a particular application. The main advantage of the $SMT_2$ processor is its ability to adapt to the thread parallelism and ILP requirements of the application. For applications with high thread parallelism like *ocean*, it performs better than $FA_8$, while for applications with low thread parallelism like *tomcatv*, it performs better than $FA_2$ and $FA_1$.

We now examine a high-end machine. We model a machine with four processor chips working on the same application. In such an environment, two major changes occur. The first one is that, because of Amdahl's law, sections that are serial or have load imbalance become more important. It is therefore crucial to speed them up. This argues for using a wide-issue processor. The second change is that the parallel sections suffer more hazards. This is because memory latencies are higher and synchronization occurs more frequently. This argues for SMT-based processors, where the issue cycles can be shared between threads. Overall, this should mean that SMTs are more attractive because, in the serial section, they can become wide-issue processors while, in the parallel section, they allow issue cycles to be shared between threads.

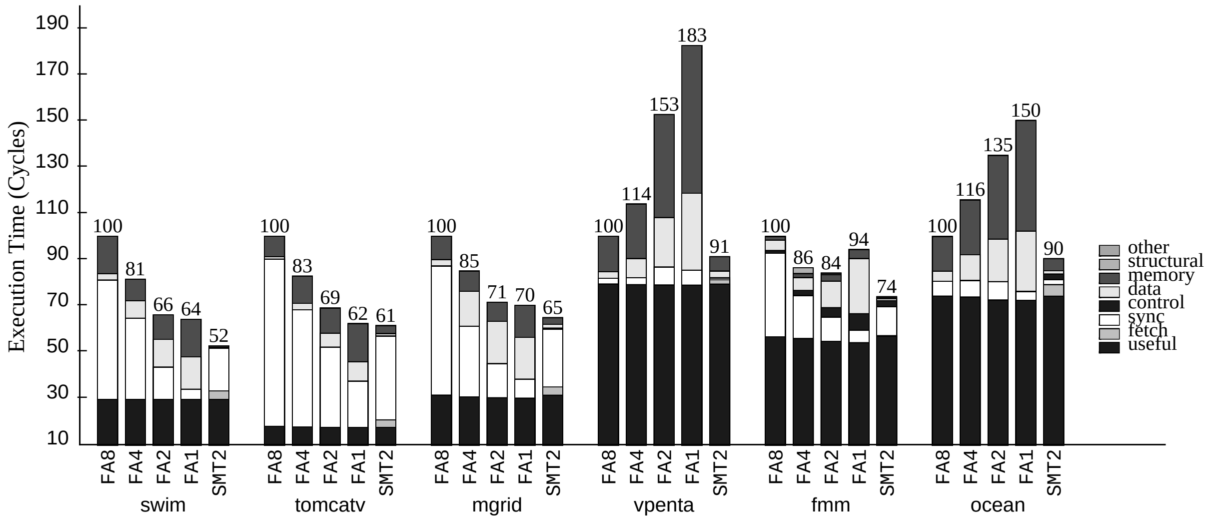



**Figure 3.4** Comparing FA processors to a clustered SMT for a high-end machine. The chart assumes the same cycle-time for all the architectures.

Figure 3.4 shows the execution time of a 4-processor high-end machine configured with the five types of architectures shown in Figure 3.3. Since there are 4 chips, $FA_8$ and $SMT_2$ run with 32 threads, while $FA_4$, $FA_2$ and $FA_1$ run with 16, 8 and 4 threads respectively. The figure is organized in the same way as Figure 3.3.

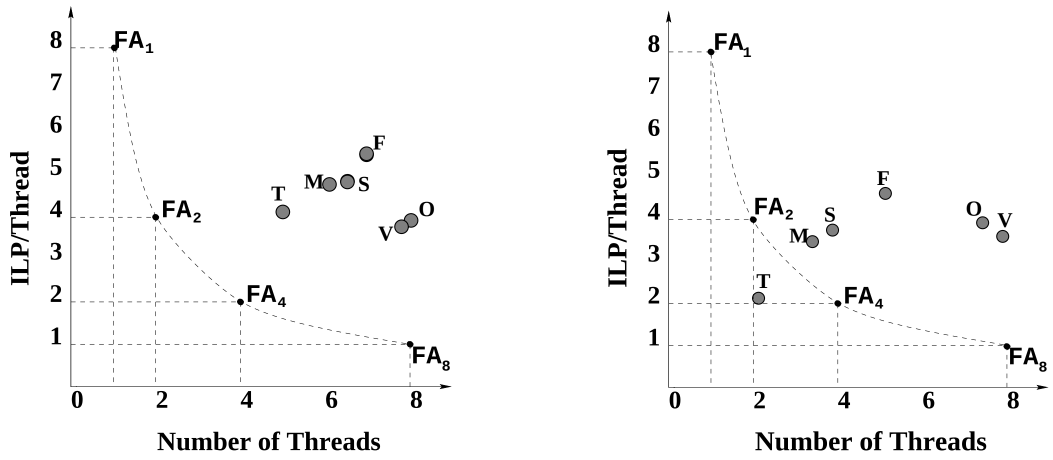
We focus first on the FA processors. For the least parallel applications, namely *swim*, *tomcatv* and *mgrid*, we see that the *sweet-spot* of their U-shape has moved to the wider issue FA processor. Indeed, while in Figure 3.3, $FA_4$ or $FA_2$ performed best, in Figure 3.4, $FA_1$ performs best. This is because it is now critical to speed up serial sections with wide-issue processors. This shows that, more than in a low-end system, exploiting

ILP is very important in a high-end system. On the other hand, if we examine highly parallel applications with little synchronization like *vpenta*, often the opposite occurs. $FA_1$ becomes relatively worse than in Figure 3.3. This is because parallel sections become more hazardous. Overall, therefore, in a multi-chip environment more than in a single-chip one, no single FA processor performs best. The variations in performance among the FA processors have exacerbated. In contrast, the $SMT_2$ processor has the lowest execution time of all processors. In addition, it delivers the most stable performance by far. We conclude, therefore, that $SMT_2$ processors are even more attractive in high-end systems.

Recall that, for the above evaluation, we did not consider variations in clock cycle-time between different architectures. However, even if we considered them, the broad conclusion would not change. This is because the $FA_2$ architecture has largely the same cycle-time as $SMT_2$. The reason is that both architectures have 4-issue superscalar clusters. These 4-issue superscalars do not have as much cycle-time constraints as higher-issue processors [59]. In addition, with current technology [16], the difference in cycle-time between 4-issue processors and lower-issue processors ($FA_4$ and $FA_8$) is very unlikely to be significant enough to make up for the large difference in performance shown in the figure. Finally, $FA_1$, which is a conventional superscalar, is likely to have a larger cycle-time than $FA_2$ or $SMT_2$ [59].

#### 3.4.1.1 Comparing the Results to the Model

We now qualitatively compare the results obtained to the performance predicted by our model of parallelism in Section 3.1. For each application, we estimate the thread parallelism and the ILP roughly. We estimate the thread parallelism by measuring the average number of threads running on the $FA_8$ processor. Such a processor enables the highest thread parallelism. We estimate the ILP by measuring the average ILP in the $FA_1$ processor. Such a processor enables the highest ILP. Based on these measurements for a low- and a high-end machine environment, we map the applications on the chart of Figure 3.5-(a) and 3.5-(b) respectively.



(a) Low-end machine.


(b) High-end machine.

**Figure 3.5** ILP versus thread parallelism for our applications. Each application is denoted by the first letter in its name.

We focus first on the low-end machine. The performance of the $FA$ processors in Figure 3.3 is consistent with the chart in Figure 3.5-(a). In Figure 3.5-(a), we see that *ocean* and *vpenta* fall closer to the lower right corner. In Figure 3.3, these are the two applications where $FA_8$ performs best while $FA_1$ performs abysmally. The rest of the applications fall more to the center, so $FA_4$ tends to perform best in Figure 3.3. The leftmost point in Figure 3.5-(a) is *tomcatv*. This is consistent with the fact that $FA_2$ performs best for *tomcatv* in Figure 3.3. We now consider the $SMT_2$ processor. All the points in Figure 3.5-(a) fall in the center of its optimal region. Therefore, $SMT_2$ performs very well in Figure 3.3, better than the $FA$ processors.

If we now consider the high-end system (Figure 3.5-(b)), we see that, compared to the low-end system, the data points have moved to the left and to the bottom in the chart. The number of threads decreases because serial sections are more important and the ILP decreases because parallel threads suffer more hazards. Focusing on the data points in the chart we see that, since *ocean* and *vpenta* have moved little, $FA_8$ is still the best $FA$ processor for these two applications in Figure 3.4. *Swim*, *fmm* and *mgrid* have moved to the left side and, therefore, $FA_1$ or $FA_2$ work best. Finally, *tomcatv* has moved much to the left and, therefore, the best design is $FA_1$ in Figure 3.4. As for $SMT_2$, all the data points except for *tomcatv* still fall within its optimal region. Therefore, it continues to perform better than the best $FA$ processor for these applications in Figure 3.4.

### 3.4.2 Clustered and Centralized SMT Architectures

We now justify the choice of $SMT_2$ in our previous section. Towards this, we evaluate the three clustered SMT and the centralized $SMT_1$ architectures for the low- and high-end machines. Figure 3.6 shows the performance of the four architectures for a low-end machine. For each application, the execution time is normalized to that of the $SMT_8$ processor. The $SMT_8$ processor is a special case of the clustered SMT processor in that it is the same as the $FA_8$ processor. Note that, as usual, our charts show the execution time in terms of number of cycles, without considering the different cycle-times of the different architectures. We will consider the different cycle-times later.

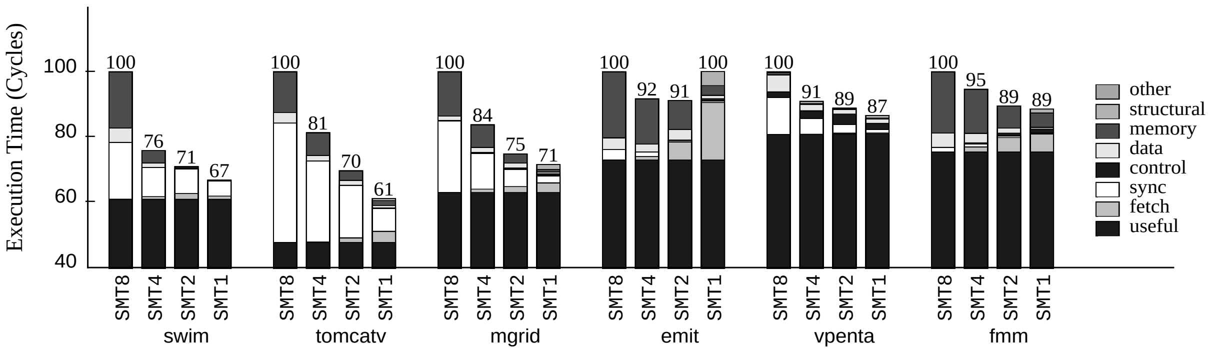



**Figure 3.6** Comparing the centralized and clustered SMT processors for a low-end system. The chart assumes the same cycle-time for all the architectures.

The figure shows that the performance of the SMT processors is usually fairly predictable across applications. It increases from $SMT_8$ to $SMT_1$. This effect, of course, is due to the higher adaptability of the wide-issue SMTs to variations in the ILP and thread parallelism of the application. For most applications, we can see that there is also a steady increase in the *fetch* hazards, going from $SMT_4$ to $SMT_1$. This is due to the unified instruction queue that we use for dynamic instruction issue and the nature of the fetch mechanism. For example, instructions issued by a thread may remain in the queue for a long time waiting on a cache miss. Since they clog the queue, other threads are unable to fetch instructions. This fetch bottleneck has been discussed in great detail by Tullsen *et al* [83]. They suggest several alternatives, such as partitioning the fetch unit or using instruction count feedback techniques to use the fetch unit more intelligently. The centralized SMT is more susceptible to this problem than the clustered SMTs.

Overall, the figure shows that the $SMT_2$ processor achieves nearly the same performance as the centralized SMT. The difference between the two ranges from $0 - 9\%$ normalized to $SMT_8$. This shows that the full issue bandwidth that is available in the centralized SMT processor is hardly used by the multithreaded applications that we used. As a result, limiting the level of ILP that may be exploited decreases the performance only to a limited extent.

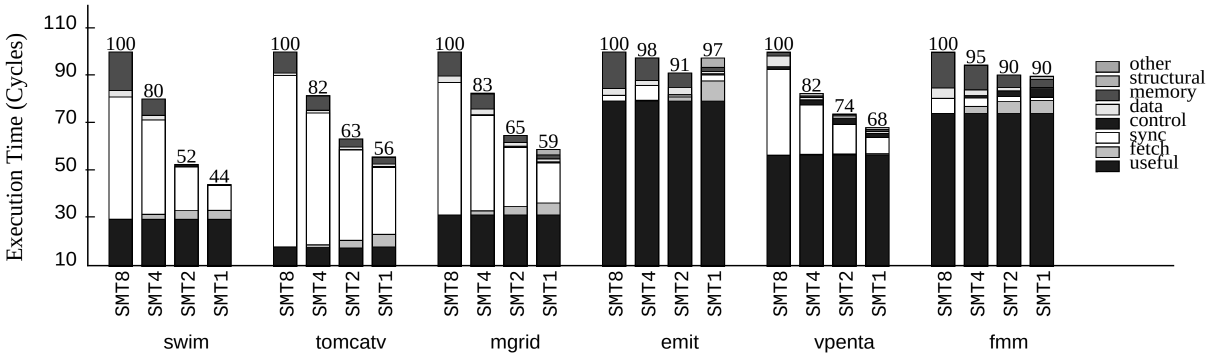



**Figure 3.7** Comparing the centralized and clustered SMT processors for a high-end system. The chart assumes the same cycle-time for all the architectures.

Finally, figure 3.7 shows the performance of the four SMT processors for a high-end system. In each application, the bars are again normalized to $SMT_8$. The data in this figure is very similar to that in Figure 3.6. Therefore, the conclusions for the low-end system apply fully here. The $SMT_2$ architecture is again only slightly slower than $SMT_1$.

If we now consider the differences in clock cycle-time, the picture changes dramatically in favor of the $SMT_2$ architecture. According to [59], 4-issue clusters are likely to have a factor of 2 in clock frequency advantage over the centralized 8-issue cluster. This gives a very large clock frequency advantage to $SMT_2$ over $SMT_1$. Given that the $SMT_2$ and $SMT_1$ bars in Figures 3.6 and 3.7 are so close, $SMT_2$ is a much better design point. Furthermore, the 4-issue and the lower-issue clusters (represented by $SMT_4$ and $SMT_8$) would have similar clock frequencies with current technology [16]. Consequently, $SMT_2$ is the most cost-effective organization.

## 3.5 Conclusions

Since aggressive superscalar processors are becoming so complex, there is much interest in alternate designs that exploit thread-level parallelism. With future applications likely to be multithreaded, this is a very promising approach. One of the current proposals is to configure the architecture such that the chip resources are partitioned in a fixed manner between the threads. These fixed-assignment architectures (FA) risk wasting resources when one of the threads stalls due to hazards or when there is a lack of threads in the application. Although basic simultaneous multithreading (SMT) allows resources to be shared, the structure is so centralized that it is likely to slow down the clock frequency. However, given that the amount of thread- and instruction-level parallelism of applications varies widely, the flexibility of the SMT architecture is very valuable.

The architecture that we have explored in this chapter is a hybrid of the FA and the SMT approach, namely the clustered SMT architecture. We have shown that this restricted level of simultaneous multithreading is able to capture most of the performance benefits of the fully centralized approach while, at the same time, allowing the SMT design to cycle at high frequency. Therefore, we conclude that the hybrid organization is the most cost-effective one.

# CHAPTER 4

# A DIRECT-EXECUTION FRAMEWORK FOR PROCESSOR-LEVEL SIMULATION

Software simulation plays an integral role in the design and validation of high-performance uni- and multi-processor systems. Two major simulation methodologies used are Trace-Driven Simulation (TDS) and Execution-Driven Simulation (EDS). In TDS, hardware probes [80, 85] or software instrumentation of the application [66, 74, 77] allow information like basic block or data addresses to be collected in a trace buffer during the application's execution. The generated information is later used to drive a simulator of the system under study. Given that an application may execute billions of instructions, to reduce the storage requirements of the traces, sampling [15, 65, 79] is often used.

While TDS is an effective methodology in the study of high-performance uniprocessor systems, it is not applicable to multiprocessor systems. The reason is that, in the latter systems, it is very important to faithfully model the interleaving of the memory accesses of the different processors. The actual interleaving depends, among other things, on the memory system used. Since the trace is generated using an existing memory system and the TDS uses a different, simulated memory system, TDS may result in the incorrect interleaving of the traces [3].

It is in this area that EDS plays a crucial role. Here, the execution of the application and the simulation is completely interleaved. The application is instrumented at appropriate points so as to generate events for the back-end simulator. When the simulator is called, it processes the event. Then, it returns control to the application. This method facilitates feedback from the simulator to guide the execution of the parallel application.

There have been many EDS-based multiprocessor system simulators for both RISC and CISC architectures [8, 14, 18, 52, 54, 61, 67, 88]. Most of these EDS-based systems are based on *direct execution*. Here, the instrumented application is executed directly on the host machine where the simulation runs. This results in fast simulation. Unfortunately, using simulation based on direct execution comes at a price. Since there is no simulated processor model, it can be used only in those studies where the processor need not be modeled accurately.

Unfortunately, though direct-execution based EDS is used commonly for multiprocessor environments where the main focus is to study memory subsystems, not modeling the processor in detail may give rise to inaccurate results [58]. This is because current multiprocessor systems are typically based on complicated processors like wide-issue dynamic superscalars. Accurately simulating current dynamic superscalar processors using direct execution has been considered an open problem [57]. Consequently, when it is necessary to model such a processor, the simulators have to resort to interpreting the application executable [58, 67]. This often results in slowdowns of orders of magnitude.

We addressing this problem in this chapter. The contribution here is the design and evaluation of a novel direct-execution framework that allows accurate simulation of advanced processor architectures in both uni- and multi-processor environments without incurring the slowdowns of an interpretational approach. This is achieved with the aid of an *Interface Window* between the front-end and the architectural simulator, that buffers limited instruction information. This eliminates the need for full-fledged instruction emulation. Overall, our direct-execution framework addresses the need for having a single, efficient simulation environment that allows the detailed modeling of the processor architecture in both uni- and multi-processor environments.

This chapter is organized as follows: Section 4.1 provides some background on direct-execution; Section 4.2 details our simulation framework, Section 4.3 evaluates its performance and, finally, Section 4.4 concludes the chapter.

## 4.1 Direct Execution

Direct-execution based EDS systems typically work by instrumenting the source, assembly or executable code to generate events to the back-end simulator. This allows events to be sent to the simulator when the modified program is executed directly on the host. This scheme does not require storing traces and is ideal for a multiprocessor environment.

A drawback of the direct-execution approach is that it can be used only in those studies where the processor need not be modeled accurately. However, in multiprocessor environments, even if our main focus is to study memory subsystems, not modeling the processor in detail may give rise to inaccurate results [58].

Another factor that weighs against some direct-execution systems is that they require source re-compilation, including library code. The application also has to be re-linked whenever the simulator is modified. These problems can be overcome by using only executables for instrumentation. However, this may require emulation of the code, in contrast to native execution on the host machine as performed by direct execution. Unfortunately, emulation often slows down the system significantly. For this reason, we want to stay away from emulation.

Interestingly, with current advances made in instruction-set emulations [66, 88], the slowdowns with smart emulation-based approaches are of the same degree as code annotation [64].

In light of this fact, and also with the added flexibility of using executables, it makes sense to use this approach as a front-end to a simulator framework. To save time in writing a front-end that reads and instruments the executable, we decided to use the

MINT [88] front-end. This also made the transition easier as most of our back-end memory simulators were written for the MINT environment.

MINT uses a hybrid approach of emulation and native execution. It executes in native mode straight-line code that has no memory references or branch instructions while emulating code that cannot be executed natively. Furthermore, MINT can also be used for multiprocessor simulations. It supports the traditional UNIX `fork/sproc` constructs but simulates multiple processes on the target machine within a single address space. Data relocation is avoided by dynamically translating the addresses. To get further details, refer to [87]. Finally, for our simulation framework, we modified the front-end so as to enable basic block instrumentation and also handle MIPS-II binaries.

## 4.2 A Direct-Execution Framework for Processor-Level Simulation

This section presents our framework. We first describe the methodology for modeling the execution of an instruction, then present a more complete view of the framework, and finally examine the way we collect statistics.

### 4.2.1 Modeling Instruction Execution

Our primary aim is to model advanced processor architectures for both uni- and multiprocessor environments. One way of achieving this would be to interpret each instruction in a simulated processor. However, this is likely to lead to significant slowdowns in many cases [57, 67]. Our approach, instead, is to perform a limited functional emulation. We elaborate on this approach in the rest of this section.

We instrument the application to generate events at basic-block boundaries. This tracks the control flow of the application. Furthermore, all load and store instructions are also instrumented to generate events for the simulator. Finally, for each instruction in a basic block, we need some limited information to change the state of the simulated

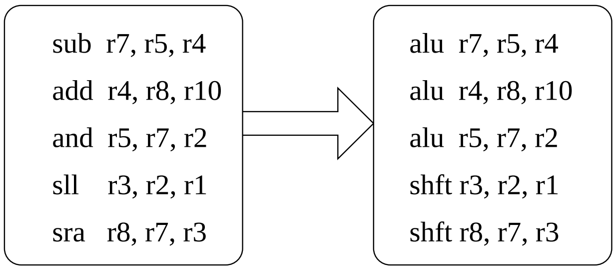



**Figure 4.1** Mapping a RISC instruction to a RISCier instruction.

processor. This information is register usage and partial opcode information. There is no need for any full-fledged instruction emulation.

More specifically, we map the host ISA to an internal instruction set, which we term the *RISCier* ISA. The RISCier ISA captures the full functionality of the native instruction set without being concerned by the different flavors of each instruction type. For instance, we combine into a single class all ALU instructions that take the same number of cycles to execute in the simulated processor and that can use the same functional units in the simulated processor. In the RISCier ISA, one such instruction is represented as a generic `alu` operation along with the registers that it uses. Also, branch delay slot scheduling is undone when the mapping to the RISCier code is performed. Figure 4.1 illustrates the mapping. The complete set of RISCier instructions are shown in Table 4.1.

| RISCier Instruction | Operations covered |
|---|---|
| alu | add,sub,set,and,or,xor,move |
| shift | all logical and arithmetic shifts |
| md | integer multiply and divide |
| uncond | unconditional branch |
| cond | conditional branch |
| call | function call |
| ret | function return |
| ld | load |
| st | store |
| fpadd | floating point add, sub, mov, abs, trunc, floor, comparisons etc. |
| fpmult | floating point multiply |
| fpdiv | floating point square root and division |

**Table 4.1** Description of the RISCier instruction set.

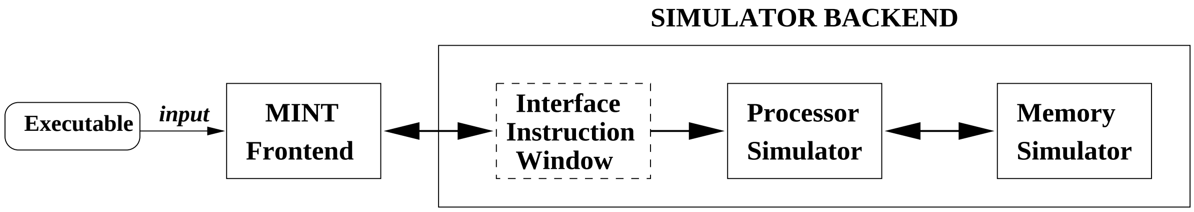



**Figure 4.2** A high-level view of the simulation framework.

Overall, our simulation framework consists of three modules. First, we have a front-end that instruments and then directly executes an executable file. For this, we use MINT [88], a well-known system that operates on MIPS executables. We have modified MINT to handle MIPS-II binaries. Then, unlike a conventional simple direct-execution simulator, events are no longer passed directly to the memory simulator back-end. Instead, they are intercepted by the processor simulator. The processor simulator changes its internal state based on these events and later releases memory references to the memory simulator.

Incorporating the processor simulator into existing MINT-based memory simulators is extremely simple. The only requirement for the memory simulator is that it should have the ability to support hit-under-miss and also handle multiple accesses in the same cycle. The information passed between modules is detailed next.

Finally, the design allows the memory back-end to be run independently of the processor simulator. This provides a convenient method to test the back-end before running the complete processor-level simulation, which undoubtedly will be slower. The actual slowdowns are reported in the evaluation section (Section 4.3).

### 4.2.2 The Interface Instruction Window

#### 4.2.2.1 Instruction Scheduling in Window

When the application is executed using the modified MINT front-end, it generates a trace of basic-block and memory addresses. However, the processor simulator cannot be invoked for each and every event. There are two reasons for it. First, the simulation would be too slow. Secondly and more importantly, the simulator cannot consume just one instruction at a time. Indeed, since the size of a basic block is only a few instructions in integer applications [89], wide-issue superscalars use advanced branch-prediction schemes [60, 93] to fetch instructions across multiple basic blocks. Thus our model should permit sufficient information from the front-end to be gathered as dictated by the branch-prediction scheme of the simulated processor, before invoking the simulator.

This effect is achieved with an interface module between the front-end and the processor simulator called the *Interface Window*. The interface window buffers the events issued by the front-end (basic block and memory addresses) before calling the processor simulator (Figure 4.2). When the outcome of a branch in the program is known, we use the branch prediction mechanism of the simulated processor to see if the latter would have predicted the branch in the correct way. If so, we continue filling the interface window with events from the front-end. Otherwise, we stop filling the window and call the processor simulator. The simulator will then consume all the events in the window. Note that the processor simulator has access to an encoded representation of the application that contains, for each basic block and on an individual instruction basis, the limited opcode information and the register usage information explained above.

When the simulator reaches the mispredicted branch instruction, it continues executing instructions from the wrong path. Note again that the basic blocks in the wrong path are known to the simulator because it has access to the application file. When the simulator reaches a point where the mispredicted branch is finally resolved in the simulated processor, the simulated pipeline is flushed and the front-end invoked again. With this approach, we correctly model the speculative execution of the simulated processor.

It is possible that the interface window is filled completely before reaching a mispredicted branch. This typically occurs only when running floating-point applications, where the branch prediction is close to ideal. In that case, the front-end stops and the processor simulator is invoked. When the window is low in instructions, the front-end is invoked again. In our studies, we set the interface window to 512 entries.

We see that the interface window allows the correct modeling of different branch-prediction schemes, independently of the processor model. In addition, it enables instruction scheduling and correct modeling of speculative execution. We consider each issue in turn.

#### 4.2.2.2 Instruction Scheduling

Typically, the application is compiled to run optimally on the host processor. The latter is often less aggressive and has a narrower issue width than the simulated processor. Consequently, we need to generate the appropriate instruction schedule for the simulated processor. Otherwise, we may degrade the performance of the simulated processor, especially when simulating in-order issue superscalars, which completely depend on good compiler scheduling.

To solve this problem, we exploit the fact that we have many basic blocks in the interface window, and reschedule the instructions. A typical compiler for a superscalar or VLIW machine performs inter basic-block code scheduling in a window of instructions that represents the most-frequently executed path. The most-frequently executed path may be either estimated statically or acquired through profiles. The Superblock in the IMPACT compiler [11] or the Trace Window in the Multiflow compiler [45] are typical examples of this approach. In our work, we take the instructions in the interface window and perform a resource-constrained list-scheduling of instructions [51] as well as limited register renaming to remove false dependences, before invoking the processor simulator. Furthermore, loops with small bodies are automatically unrolled inside the window, with the amount of unrolling constrained by the size of the window.

Overall, therefore, we perform compile-time optimizations at run-time without using any run-time information such as load or store addresses. The only information that we use is the branch-prediction of the simulated processor that is used for growing the window. Unfortunately, unlike a compiler, which can look at huge instruction windows spanning thousands of instructions and can perform advanced optimizations such as software pipelining [37], we are restricted by the size of the run-time interface window in our scheme. Thus, our approach falls short of a pure compiler-based approach to scheduling. Nevertheless, as we show in our evaluations, our approach considerably improves the ILP when modeling in-order issue superscalars. It must be noted that a similar strategy was adopted by the UltraSparc team to generate code for the in-order issue UltraSparc-I processor [81].

#### 4.2.2.3 Speculative Execution

We model speculative execution by using the branch prediction mechanism of the simulated processor as indicated before. This allows multiple branch predictions to be performed when there are pending unresolved branches. In addition, it realistically allows the pipeline to be polluted with instructions that will later be squashed once the outcome of the mispredicted branch is known. This allows the simulation to proceed somewhat independently of the native execution of the application.

Without further analysis, however, we do not know the addresses of the data accessed by the wrongly-speculated instructions. This is because the application, running under MINT, will never execute those instructions. To address this problem, we can perform a careful analysis of the value of the registers of the wrongly-speculated instructions and, in most cases, determine the data addresses. However, to simplify the problem, in our current setup, we neglect this issue and assume that the memory accesses of the wrongly-speculated instructions always hit in the L1 cache and do not pollute it.

### 4.2.3 The Simulator Framework

Our framework is quite flexible and allows different processor types and configurations to be modeled. The base processor models that are supported include an in-order issue superscalar that can speculatively fetch across branches and an out-of-order issue superscalar that is modeled on the lines of the MIPS R10000 [50] and is shown in Figure 1.1.

The model is quite detailed and incorporates all superscalar features such as a register renaming mechanism, a large associative instruction queue for out-of-order issue, speculation across multiple branches and in-order instruction retirement.

We model the pipeline in great detail. A scoreboard keeps track of operand availability. An instruction is dispatched when its operands are ready and the corresponding functional unit is available. We use event queues to speed-up the simulation. Consider, for instance, the instruction queue. Rather than scanning it every cycle, we do so only when there is a possibility that an instruction changes status. For example, if a load instruction that misses in the cache causes dependent instructions to clog the instruction queue, we can skip checking the instruction queue until the load data becomes available.

Our framework also uses the superscalar core as a building block to model advanced processor architectures such as a chip multiprocessor (CMP), where multiple superscalar cores share a single chip [30]; a simultaneous multithreaded (SMT) processor, which adds multiple threads to a superscalar and allows instructions from different threads to be issued in the same cycle [83]; and finally a hybrid of the two architectures that supports several SMT processors on a single chip [35]. By varying parameters such as the number of on-chip processors and the number of threads in a SMT processor, we can model different architectures. For instance, a conventional superscalar has just one processor on chip and supports only one thread, while a CMP has $n$ on-chip processors, where each processor supports one thread. For modeling the speculation-based architectures described in chapter 2, we enhanced the framework to allow multiple instruction windows to be maintained for providing trace data for each processing unit.

We also model a multiprocessor system, where each node in the system can be any of the above processor types. The memory subsystem for the uniprocessor is based on a conventional L1-L2 cache hierarchy, while the multiprocessor system (Figure 3.2) supports a DASH-like [40] directory-based cache-coherence protocol. Some of the processor and memory subsystem parameters that can be specified at startup time for modeling the machine are given in Table 4.2.

### 4.2.4 Statistics Collection

The simulator allows detailed statistics collection each cycle for each thread and processor. For the processor, we gather statistics on an issue slot basis. The total

| Parameter | Subtypes |
|---|---|
| Processor type | Number of processors/chip (CMP-level)<br>Threads/Processor (SMT-level) |
| Issue mechanism | Issue policy (in-order/out-of-order)<br>Issue width |
| Dynamic issue mechanism | Renaming registers<br>Instruction queue entries |
| Branch prediction (correlating) [60] | Prediction table entries<br>Mispredict penalty (cycles) |
| Functional units (int, load/store, fp) | Number of units<br>Type of instructions handled<br>Instruction latency and repeat rate (for non-pipelined units) |
| Multiprocessor support | Number of processing nodes |
| Memory-related | Number of outstanding loads and stores<br>L1 and L2 cache size, associativity, block size, banks<br>Local and remote memory latency |

**Table 4.2** Sample simulation parameters that can be specified at startup.

number of slots is the product of the total number of cycles taken by the application and the maximum issue rate. Wasted or non-*useful* slots are classified according to the type of hazard that prevented the slot from being used. The different categories of hazards are: lack of functional units (*structural*), memory access (*memory*), data dependences (*data*) and branch mispredictions (*control*). In addition, when executing parallel applications, the threads can also spin on barriers and locks (*sync*). Finally, we have an *other* category for less frequent events such as running out of renaming registers.

In an in-order issue processor, the hazard type that is responsible for unfilled slots in a given cycle can be uniquely identified. An extension needs to be made when running multiple threads as in in-order issue SMT processors. Here, since several threads can compete for a given issue slot, we assign the non-*useful* slots proportionally. For example, consider 8 threads in an 8-issue SMT processor. Further assume that, in a given cycle, four slots are filled with useful instructions. Thus we have a total of four wasted slots in this cycle. Assume that the threads cannot issue more instructions because, say, four are stalled on memory accesses, three are stalled on a data hazard and the remaining one cannot issue because of a structural hazard. In our statistics, we assign the four non-*useful* slots to the different reasons on a proportional basis. We use the formula:

$$type_i \ wasted \ slots \ = \ total \ wasted \ slots \ \times \ threads \ stalled \ on \ reason_i \ / \ total \ threads$$

In the above example, the four wasted slots are assigned as follows: $4 \times 4/8 = 2$ *memory* slots, $4 \times 3/8 = 1.5$ *data* slots and $4 \times 1/8 = 0.5$ *structural* slots.

In an out-of-order issue processor, it is usually impossible to attribute only one hazard type in a given cycle. Therefore, we scan the instruction queue every cycle and record the type of hazard faced by each instruction that is unable to issue. If, in addition, the processor has several threads, we do this for all the threads. At the end, the wasted slots are divided proportionally among the different types of hazards.

### 4.2.5 Other Features

Finally, our framework has the ability to dynamically turn on and off the simulation. This allows sampling at regular intervals. We can also selectively turn off the processor simulator while allowing memory references to the memory back-end. This is particularly helpful when running parallel applications in which the initialization phase need not be processor-simulated, while the caches have to be warmed-up before the program enters the parallel section [91]. Additionally, it allows the development and testing of the memory back-end independently of the processor simulator.

## 4.3 Evaluation

The complete framework is implemented in C/C++ and is around 20K lines of code. Both accuracy and speed were kept in mind during the implementation. Unlike in interpretation-based simulations, where the simulator can be validated based on the results produced by the application, simulations based on direct execution are difficult to validate as they model only the timing aspect of the application. To mitigate this problem, we have implemented our simulator so that it can be executed in two different modes: a safe mode with assertions added at every stage and a fast mode that disables the checking. The safe mode validates the implementation of the processor model during the course of execution. We use this mode for a set of sample inputs before using the fast mode for our experiments.

We evaluate our framework for both a uni- and a multi-processor environment. The evaluation for the uniprocessor environment is performed with two SPECint95 (*compress* and *ijpeg*), one multimedia [39] (*mpeg*) and four SPECfp95 (*swim*, *tomcatv*, *hydro2d* and *mgrid*) applications. Two SPLASH-2 [91] (*radix* and *cholesky*) applications are used to evaluate the multiprocessor environment.

The applications are compiled for a MIPS R4400 processor, using the highest level of optimization and the *-mips2* option. The MIPS R4400 is a simple single-issue processor. Since we want to model a more aggressive architecture, we will have to reschedule the instructions in the interface window.

The architecture modeled is a 4-way dynamically- or statically-scheduled superscalar processor. The value of some of its architectural parameters is specified in Table 4.3. The processor has a 2K-entry direct-mapped correlating branch prediction table [60] that allows multiple branch predictions in a cycle. All functional units are fully pipelined, with most instructions taking 1 cycle to complete. The only exceptions are as follows: 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 processor can have up to 16 outstanding memory accesses, of which half can be loads. The characteristics of the memory hierarchy are shown in Table 4.4.

| Issue Width | Number of Functional Units (int/ld-st/fp) | Entries in Instruction Queue | Number of Renaming Registers (int/fp) |
|---|---|---|---|
| 4 | 4/2/2 | 32 | 32/32 |

**Table 4.3** Characteristics of the superscalar processor.

In the following, we perform two sets of experiments. First, we measure the IPC increases that result from performing instruction scheduling in the interface window as described in Section 4.2.2. Then, we evaluate the slowdowns that result when the

| Parameter | Value |
|---|---|
| [L1 / L2] size (Kbytes) | [64 / 1024] |
| [L1 / L2] line size (Bytes) | [32 / 64] |
| [L1 / L2] associativity | [2-way / 4-way] |
| L1 banks | 7 |
| L1 latency (Cycles) | 1 |
| L2 latency (Cycles) | 6 |
| Memory latency (Cycles) | 40 |

**Table 4.4** Characteristics of the memory hierarchy. Latencies refer to round trips.

simulator is incorporated to the MINT front-end. For the latter experiments, we only include the processor simulator, not the memory back-end simulator. This is done to isolate the slowdowns caused by the processor simulator alone.

### 4.3.1 Interface Instruction Window Scheduling

*Ideal Memory System*

| Application | ILP Before | ILP After |
|---|---|---|
| *compress* | 1.89 | 2.35 |
| *ijpeg* | 1.65 | 2.09 |
| *mpeg* | 1.63 | 2.24 |
| *swim* | 1.35 | 1.52 |
| *tomcatv* | 1.33 | 1.62 |
| *hydro2d* | 1.17 | 1.56 |
| *mgrid* | 1.24 | 1.38 |

*Real Memory System*

| Application | ILP Before | ILP After |
|---|---|---|
| *compress* | 1.72 | 1.99 |
| *ijpeg* | 1.65 | 2.06 |
| *mpeg* | 1.63 | 2.19 |
| *swim* | 0.89 | 0.97 |
| *tomcatv* | 0.83 | 0.97 |
| *hydro2d* | 0.93 | 1.18 |
| *mgrid* | 1.21 | 1.29 |

**Table 4.5** Impact of the ILP-enhancing rescheduler.

To see the impact of rescheduling the instructions in the interface window as described in Section 4.2.2, we simulate the statically-scheduled processor with and without instruction rescheduling. Table 4.5 shows the average IPC for the applications with and without instruction rescheduling. The table has two sets of data, namely one assuming the memory system characteristics of Table 4.4 (real memory system) and one where all data accesses hit in the first-level caches (ideal memory system). From the table, we

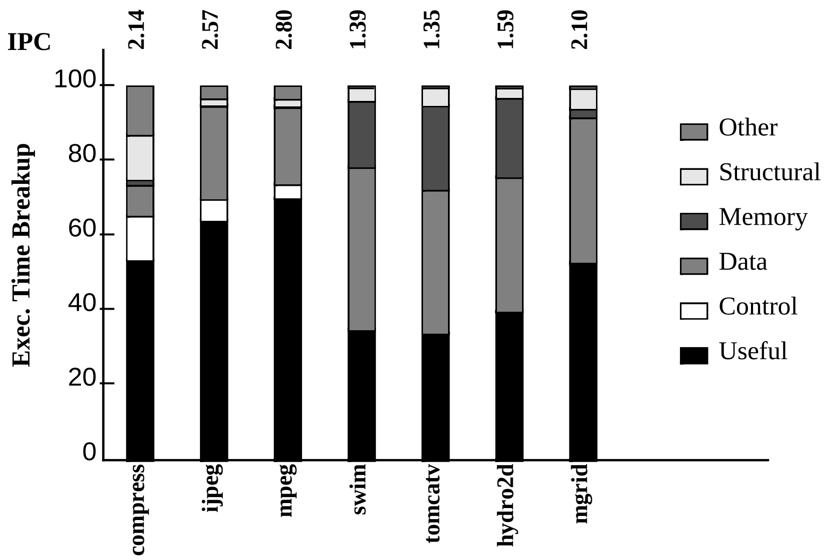



**Figure 4.3** IPC and execution time breakup in a 4-way dynamic superscalar.

see that, thanks to the rescheduling, the IPC increases for all applications and memory systems. The increases range from 7% to 37%, and are on average 22%.

Instruction rescheduling in the interface window does not have much effect for out-of-order issue processors. This is because the processor hardware itself already aggressively reorders the instructions. The result is that the IPC changes are negligible. Figure 4.3 shows the IPC for the out-of-order issue superscalar with the real memory system of Table 4.4. The average IPC is now 1.99. The figure also shows a detailed breakup of the total execution time in terms of the different categories mentioned in Section 4.2.4. This provides valuable information in identifying performance bottlenecks. For instance, a large *data* contribution in the loop-intensive floating-point applications highlights the need for software pipelining [37].

### 4.3.2 Slow-downs for Uni-Processor and Multi-processor Simulation

In this section, we evaluate the slowdown of the simulator when modeling a uni- and multi-processor system. To isolate the slowdowns caused by the processor simulator, we do not include the memory back-end in these experiments. The original MINT and the version enhanced with the processor simulator were linked with empty memory back-end functions. For each application, we report the average time taken by 5 runs. All measurements are made on one node of a SGI PowerChallenge based on MIPS R10000 processors.

Columns 2 and 3 of Table 4.6 give the slowdown over MINT when modeling the 4-issue static and dynamic superscalars. From the table, we can see that the slowdown varies widely across the applications. Overall, the average slowdown for these experiments is around 10.

Column 4 shows the MINT slowdown over the native execution of the applications. MINT has been reported to have a slowdown of around 40-70 [88]. However, the applications used in [88] are quite different from those that we use. In our case, we see slowdowns in the range of 70-200. Factoring this, the full simulator is around 1300 times slower than native execution. This is in contrast to the several thousand-time slowdowns of interpretation-based systems, such as the MXS processor simulator in SimOS [67].

In addition, since our processor simulator is independent of the front-end, we have the flexibility of using a faster front-end such as Shade [66], which utilizes dynamic compilation and caching for improved performance.

| Application | Static | Dynamic | MINT slowdown |
|---|---|---|---|
| *compress* | 4.5 | 5.9 | 115 |
| *ijpeg* | 18.5 | 23.8 | 139 |
| *mpeg* | 14.6 | 18.7 | 152 |
| *swim* | 5.4 | 7.4 | 70 |
| *tomcatv* | 1.5 | 1.8 | 131 |
| *hydro2d* | 3.3 | 4.3 | 123 |
| *mgrid* | 15.4 | 18.1 | 199 |
| Average | 9.0 | 11.4 | 133 |

**Table 4.6** Slowdown compared to MINT with empty memory back-end when simulating a 4-way static and dynamic superscalar. The last column denotes MINT slowdown over native execution.

Finally, we examine the slowdowns of the simulator when simulating a multiprocessor configuration in which each node is the 4-way dynamic superscalar. Two parallel applications, namely *radix* and *cholesky* are used for this purpose. Table 4.7 gives the results.

Columns 3 and 4 of Table 4.7 show the slowdown of MINT and our simulator relative to the simulation of a single node on the original MINT simulator. Recall that, in all experiments, we use an empty back-end memory simulator. As expected, the slowdown increases with more processors.

More importantly, the last column is the ratio between the previous two columns. It shows the slowdown of our system relative to MINT for the same number of processors. We can see that the slowdowns have increased relative to the uniprocessor slowdowns of Table 4.6. They are now in the high twenties. However, the numbers vary little with different numbers of processors. This indicates that our simulator scales quite efficiently with the number of processors.

| Application | Number of Nodes | MINT Slowdown (A) | Full Slowdown(B) | Relative Slowdown (B/A) |
|---|---|---|---|---|
| *radix* | 1 | 1.0 | 28.2 | 28 |
| | 8 | 1.1 | 31.1 | 29 |
| | 16 | 1.2 | 34.3 | 29 |
| | 32 | 1.5 | 41.9 | 28 |
| *cholesky* | 1 | 1.0 | 27.0 | 27 |
| | 8 | 1.4 | 31.1 | 22 |
| | 16 | 1.5 | 39.5 | 26 |
| | 32 | 2.8 | 67.2 | 24 |

**Table 4.7** Slowdown when simulating a multiprocessor configuration based on a 4-way dynamic superscalar. The time taken by MINT for a single-node configuration is taken as the base time. The last column denotes relative slowdown of the full simulator over MINT for equal number of simulated nodes.

## 4.4 Conclusion

Trace-Driven simulations have typically been used to evaluate high-performance uniprocessor systems. However, it is not applicable to multiprocessor systems, which require EDS to model the exact interleaving of memory accesses. A common approach for modeling such systems has been to use a direct-execution based EDS. However, using such as a system to model superscalar processors has long been considered an open problem. Consequently, existing schemes resort to application code interpretation, which results in extremely slow simulations. This chapter proposed and evaluated a novel approach to using a direct-execution based EDS for modeling advanced processor architectures, both in a uniprocessor and multiprocessor setup. Our evaluations showed that we can model the processor in great detail without resorting to application code interpretation while, at the same time, achieve fast simulations.

# CHAPTER 5

# CONCLUSIONS

## 5.1 Thesis Summary

Chip-multiprocessors (CMP) are a promising approach for exploiting the increasing transistor count on a chip. For our research, we proposed a software-hardware approach that enables speculative execution of a sequential binary on a CMP architecture without source re-compilation. The software involves a binary annotator that performs detailed compiler analysis to identify units of work for each thread and inter-thread register-level dependences. The hardware includes support for inter-thread register communication and memory disambiguation.

We have also shown that fast communication at the register level is a requirement for high performance, especially for CMPs based on high-issue dynamic superscalars. To enable such flexible inter-thread register communication, we proposed augmenting a conventional scoreboard to a decentralized module called the *Synchronizing Scoreboard (SS)*. Overall, our scheme adds modest support to enable register-level communication without specializing the architecture so much towards speculative mode of execution, that it leads to much unutilized hardware under fully-parallel or multiprogrammed loads.

We also proposed hardware support in the form of a *Memory Disambiguation Table (MDT)* to enable detection of memory dependence violations. Furthermore, we use each processor's primary cache to store the speculative data and propose a cache-coherence mechanism, along the lines of a directory-based protocol, to maintain data consistency. This system allows natural scalability to a cluster of CMPs. We evaluated this hardware-software approach and showed that it can indeed lead to high performance when executing sequential binaries.

We also explored related approaches for configuring a multithreaded architecture so as to achieve maximum performance. Specifically, we combined the chip multiprocessor (CMP) and a simultaneous multithreading (SMT) processor. In this clustered SMT architecture, each chip has independent processing units, with each unit having the capability to perform simultaneous multithreading. We evaluated this architecture when running parallel applications and showed that this restricted level of simultaneous multithreading is nevertheless able to capture most of the performance benefits of the fully centralized approach. We also showed that the results hold for speculatively executed sequential applications with the added software and hardware support as described in the previous section. Overall, we concluded that the hybrid organization was a cost-effective one.

Finally, we also proposed and implemented a novel direct-execution based framework that allows accurate simulation of wide-issue superscalar processors without the need for code interpretation. Overall, this approach enables detailed yet fast EDS of superscalar processors. We evaluated this framework and showed good performance for uni- and multi-processor configurations.

## 5.2 Future Directions

The dissertation work opens up quite a few avenues for research in the immediate future. In this section, we outline some of the possible extensions to our current work.

We can easily extend our memory dependence handling mechanism to a multiprocessor system based on a cluster of CMPs, thus providing us the ability to extract speculative parallelism both within and across chips. Indeed, the MDT proposal allows natural scalability to a cluster of CMPs. For instance, consider an application that has multiple levels of loop-nesting. For such loops, there are two levels of speculative parallelism that may be exploited. The outer loop may be run speculatively parallel on a conventional multiprocessor such that each node node in the multiprocessor system would handle a loop iteration. This requires a MDT-like structure structure to be configured along with the memory modules to handle any “outer loop” dependence violations. Furthermore, each node in the multiprocessor system may further exploit speculative parallelism in the inner loop present within the outer loop iteration. This can be achieved by using a CMP that incorporates the MDT hardware that we discussed in an earlier chapter.

Such a separate dependence tracking mechanism, in the form of the MDT, leaves room for further improvements. For instance, rather than just checking for memory dependence violations, we can use a prediction mechanism that would speculate on memory dependences. Most often, the dependences tend to recur. Thus it would be preferable to stall a successor thread from performing a load before the store to the location is completed. This avoid unnecessary squashing and restarting of threads. The stalling can be done with the aid of a prediction mechanism that maintains the instruction addresses of the possibly unsafe load instructions and the corresponding store instruction. The predictor may be a simple 2-bit up/down counter on the lines of a traditional branch predictor. More information such as the dependence distance between threads may also be incorporated to make the prediction mechanism more effective.

The strict sequential semantics which we use for retiring threads on a CMP may be relaxed with the help of additional hardware. This would be extremely useful especially when running threads with severe load imbalance. Thus, as soon as a thread completes, a new thread can be immediately initiated on the processor.

Support for communication at the register level opens up a plethora of opportunities for a compiler that is aware of the speculation hardware. For instance, dependences at the memory level can be converted to register-level to allow partial (do-across type) parallelism to be exploited more efficiently.

Finally, we have limited the threads to loop iterations. This needs further enhancements to cover all sections of the code. An interesting extension would be to provide support for predication, without requiring extensions at the ISA level. In other words, we could execute both paths of a conditional branch on different threads, and eventually squash the one executing in a mispredicted path. This should be achievable with modest effort since we already have a framework for speculation.

# REFERENCES


[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, Mass, 1988.

[3] P. Bitar. *A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors*, pages 37–52. Kluwer Academic Publishers, Editors: M. Dubois and S. Thakkar, May 1990.

[4] 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.

[5] M. Bohr. Silicon Trends and Limits for Advanced Microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[6] M. T. Bohr. Interconnect Scaling - The Real Limiter to high-performance ULSI. In *International Electron Devices Meeting Technical Digest*, pages 241–244, 1995.

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

[8] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS-TR-516, MIT Laboratory for Computer Science, September 1991.

[9] D. Burger and J. Goodman. Billion-Transistor Architectures. *IEEE Computer*, 30(9):46–50, September 1997.

[10] P. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *28th International Symposium on Computer Microarchitecture (MICRO-28)*, pages 252–257, November 1995.

[11] P. P. Chang, S.A. Mahlke, W. Y. Chen, N. J. Water, and W.W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[12] R. Colwell and R. Steck. A $0.6\mu$m BiCMOS Processor with Dynamic Execution. In *ISSCC Proceedings*, 1995.

[13] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *22nd International Symposium on Computer Architecture*, pages 333–344, June 1995.

[14] H. Davis, S.R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing using Tango. In *International Conference on Parallel Processing*, pages II99–107, August 1991.

[15] D.A.Wood, M.D.Hill, and R.E.Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.

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

[17] 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.

[18] S. Dwarkadas, J.R.Jump, and J.B.Sinclair. Execution-Driven Simulation of Multiprocessors: Address and Timing Analysis. *ACM Transactions on Modeling and Computer Simulation*, 4(4):314–338, October 1994.

[19] J.H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, April 1995.

[20] M. Berry et al. The Perfect Club Benchmarks. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

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

[22] M. Franklin and G. Sohi. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors. In *25th International Symposium on Microarchitecture (MICRO-25)*, pages 236–245, Dec 1992.

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

[24] 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.

[25] M. Gulati and N. Bagherzadeh Emer. Performance Study of a Multithreaded Superscalar Microprocessor. In *2nd International Symposium on High-Performance Computer Architecture*, pages 291–301, February 1996.

[26] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14):11–16, October 1996.

[27] L. Gwennap. Intel, HP Make EPIC Disclosure. Microprocessor Report, October 1997.

[28] L. Gwennap. *Intel's Merced and IA-64: Technology and Market Forecast*. MicroDesign Resources, 1998.

[29] 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.

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

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

[32] G.E. Daddis Jr and H.C. Torng. The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors. In *International Conference on Parallel Processing*, pages I:76–83, August 1991.

[33] S.W. Keckler and W.J. Dally. Processor Coupling: Integrating Compile time and Runtime Scheduling for Parallelism. In *19th International Symposium on Computer Architecture*, pages 202–213, May 1992.

[34] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[35] V. Krishnan and J. Torrellas. A Clustered Approach to Multithreaded Processors. In *12th International Parallel Processing Symposium (IPPS)*, April 1998.

[36] V. Krishnan and J. Torrellas. Executing Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support. In *4th High Performance Computer Architecture (HPCA) Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC'98)*, February 1998.

[37] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Processors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[38] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[39] 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.

[40] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[41] G. Lesartre and D. Hunt. PA-8500: The Continuing Evolution of the PA-8000 Family. In *COMPCON*, 1997.

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

[43] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[44] J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, August 1997.

[45] G. Lowney, S.M. Freudenberger, T.J. Karzes, W. D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[46] E. Lusk et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., New York, 1987.

[47] S. A. Mahlke, R. E. Hank, J.E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *22nd International Symposium on Computer Architecture*, pages 138–150, June 1995.

[48] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control For Energy Reduction. In *25th International Symposium on Computer Architecture*, June 1998.

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

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

[51] S.S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[52] S.S. Mukherjee, S.K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and its Impact on Design (PAID) (held in conjunction with ISCA'97)*, June 1997.

[53] B. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocesor. In *23rd International Symposium on Computer Architecture*, pages 67–77, May 1996.

[54] A-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *International Conference of Computer Design*, pages 486–490, October 1996.

[55] 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.

[56] J. Oplinger, D. Heine, S-W. Liao, B.A. Nayfeh, M.S. 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.

[57] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[58] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

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

[60] S.-T. Pan, K. So, and J.T. Rameh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–84, September 1992.

[61] D. Poulsen and P.-C. Yew. Execution-Driven Tools for Parallel Simulation of Parallel Architecture and Applications. In *Proceedings of Supercomputing*, pages 860–869, November 1993.

[62] R.G. Prasadh and C.-L. Wu. A benchmark evaluation of a Multithreaded RISC Processor Architecture. In *International Conference on Parallel Processing*, pages I:84–91, August 1991.

[63] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[64] R.A.Uhlig and T.N.Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(1):78–103, June 1997.

[65] R.E.Kessler, M.D.Hill, and D.A.Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Transactions on Computers*, C-43:664–675, June 1994.

[66] R.F.Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[67] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[68] E. Rotenberg, S. Bennett, and J.E. 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.

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

[70] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *23rd International Symposium on Computer Architecture*, pages 90–101, May 1996.

[71] B. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE (Real-Time Signal Processing IV)*, 298:241–248, 1984.

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

[73] J.E. Smith and G.S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[74] M. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Center for Integrated Systems, Stanford University, November 1991.

[75] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *24th International Symposium on Computer Architecture*, pages 194–205, June 1997.

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

[77] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[78] 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.

[79] T.M.Conte and W.W.Hwu. Systematic Prototyping of Superscalar Computer Architectures. In *3rd IEEE International Workshop on Rapid System Prototyping*, June 1992.

[80] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 162–174, October 1992.

[81] M. Tremblay, D. Greenley, and K. Normoyle. The Design of the Microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1995, December 1995.

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

[83] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[84] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous Multithreading: Maximizing on-chip Parallelism. In *22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[85] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design Tradeoff for Software-Managed TLBs. *ACM Transactions on Computer Systems*, 12(3):206–235, August 1995.

[86] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *24th International Symposium on Computer Architecture*, pages 1–12, June 1997.

[87] J. Veenstra and R. Fowler. MINT Tutorial and User Manual. Technical Report TR-452, University of Rochester, Computer Science Department, May 1993.

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

[89] D.W. Wall. Limits of Instruction Level Parallelism. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–189, April 1991.

[90] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass, 1989.

[91] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[92] W.W.Hwu and Y. Patt. HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *13th International Symposium on Computer Architecture*, pages 297–307, June 1986.

[93] T. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *19th International Symposium on Computer Architecture*, pages 124–134, May 1992.

[94] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *4th International Symposium on High-Performance Computer Architecture*, pages 162–173, February 1998.