The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors[1]

Venkata Krishnan[2] and **Josep Torrellas**

University of Illinois at Urbana-Champaign, IL 61801
Venkata.Krishnan@compaq.com torrellas@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Chip-multiprocessor (CMP) architectures are a promising design alternative to exploit the ever-increasing number of transistors that can be put on a die. To deliver high performance on applications that cannot be easily parallelized, CMPs can use additional support for speculatively executing the possibly data-dependent threads of an application. While some of the cross-thread dependences in applications must be handled dynamically, others can be fully determined by the compiler. For the latter dependences, the threads can be made to synchronize and communicate either at the register level or at the memory level. In the past, it has been unclear whether the higher hardware cost of register-level communication is cost-effective.

In this paper, we show that the wide-issue dynamic processors that will soon populate CMPs, make fast communication 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 toward speculation that it leads to much unutilized hardware under workloads that do not need speculative parallelization. Finally, the scheme allows the system to achieve near ideal performance.

1 Introduction

Advances in VLSI technology now allow over 100 million transistors to be configured on a single processor die [14] and a billion-transistor processor chip is in the offing. Unfortunately, on-chip interconnects are likely to become a significant bottleneck in future processors, where a signal is expected to take multiple clock cycles to traverse the entire chip [17]. Thus, rather than implementing just one high-issue superscalar processor on the chip, many researchers have proposed decentralized architectures wherein multiple simpler processing units are configured on a single chip.

Indeed, the chip-multiprocessor (CMP) architecture has drawn great attention, with architects proposing various related designs [5, 10, 12, 16, 20, 22, 23, 24]. Though the CMP is an ideal platform to run multiple sequential applications or a fully-parallel application, if it is to be fully accepted, it must also be able to give good performance when running a single sequential application or one that cannot be parallelized by the compiler effectively. CMPs handle these applications by resorting to a speculative mode of execution. In this mode, the threads that execute on the on-chip processors do not need to be fully independent; they may have data dependences with each other. Such speculative threads may be identified either at compile time [4, 10, 12, 22, 23, 24] or completely at run-time with hardware support [16, 19].

In these speculative CMPs, additional hardware support is needed to enforce inter-thread dependences and ensure that sequential semantics are not violated. As a result, threads may be squashed and restarted when a dependence violation is identified. For example, this occurs when a thread generates a datum that a speculative thread has already prematurely consumed. The consumer thread must then be squashed.

Not all of the dependences need to be handled in this speculative manner. Some of the cross-thread dependences can be fully determined by the compiler from the static code. In these cases, speculation can be avoided. The hardware can simply synchronize the data transfer between processors, thereby preventing a successor thread from using a stale value. The synchronizing data transfer can occur at the memory level (typically a shared L2 cache) or at the register level with the aid of a fast interconnect to communicate the values.

1.1 Current Status

The performance of these speculative CMPs is the subject of intense current study. It has been shown that there is considerable performance potential for even integer applications using this speculative approach [23].

Some designs are largely specialized towards speculation, like the Multiscalar [22] and Trace [20] processors. They add significant hardware, such as duplicate registers for each processor along with a buffered ring network for communication [3], or a centralized global register set in addition to the per-processor register sets [20].

The other designs have less hardware support for speculation [10, 12, 23, 24]. The philosophy of these "speculative-light" systems is to augment the CMP with just enough support to allow speculative execution, while still maintaining the generic CMP architecture to some degree. Within these designs, there are many differences. While some designs rely heavily on software for speculative thread initiation, commit and squash [10, 23], others rely on hardware [12, 24].

Hammond *et al* [10] have shown that large software overheads in thread initiation and commit are responsible for only modest speedups, and in a few cases slowdowns, of a speculative CMP over using just one processor in the CMP. The authors further state that the grain size of the thread plays an important role in speculative execution. When it is too large, the probability of a dependence violation occurring increases leading to much wasted work. On the other hand, when the grain size is small (< 100 instructions), software overheads become significant enough that they warrant hardware support for thread initiation and commit.

Another factor that affects performance is the communication of compiler-identifiable cross-thread dependent values be-

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

[2]Now at Compaq Computer Corporation, 334 South Street, Shrewsbury, MA 01545.

tween processors. Currently, an important difference between speculative CMP architectures is whether they support fast communication [12, 16, 20, 22] or not [10, 23]. It is possible that register-allocating frequently-shared variables along with adding extra hardware for communicating the values quickly may prove beneficial. Without such hardware support, a speculative processor has to perform additional load and store operations to use the memory system to communicate values.

A study of the performance impact of communication latency in speculative CMPs [23] argues that a fast communication scheme may not be required and that communication through the memory subsystem is sufficient. However, such a study was performed assuming CMPs made of static single-issue processors. In addition, it assumed threads of large grain size, where the instructions were aggressively hand-scheduled to minimize the waiting time for the consumer thread.

However, future speculative CMPs are likely to be populated with wide-issue dynamic superscalars. In these systems, a faster mode of communication, possibly at the register level, may be required. Moreover, in the absence of the aggressive instruction scheduling techniques assumed in [23], CMPs must be able to exploit finer-grain parallelism better if they are to match the performance of a conventional superscalar using the same die area [10].

1.2 Our Contribution

Our contribution is to study the impact of inter-processor communication latencies in speculative CMPs with wide-issue dynamic superscalars. Our design point is a "speculative-light" CMP (in the sense discussed above), but one in which thread initiation, commit and squash are done with low overhead in hardware to allow better exploitation of threads with small grain size. Furthermore, our applications are sequential executable files, from which we automatically extract threads without recompiling the source.

We show that wide-issue dynamic superscalars make fast communication a requirement for high performance. Consequently, we propose a hardware mechanism to support communication and synchronization of registers between on-chip processors. The scheme adds enough support to enable register-level communication without specializing the CMP architecture so much towards a speculative mode of execution that it leads to much unutilized hardware under fully-parallel, compiler-analyzable applications or multiprogrammed loads of sequential programs. Finally, the scheme allows the system to achieve near ideal performance.

This paper is organized as follows: Section 2 motivates the problem; Section 3 describes the basic support for speculation; Section 4 describes the evaluation environment used; Section 5 looks at the impact of communication latency; and Section 6 describes and evaluates our hardware support for register-level communication.

2 Motivation

Proposed CMP architectures include hardware support to enforce data dependences, 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 [7] or with distributed schemes that use a modified cache-coherence protocol among the processors in the CMP [8, 10, 12, 23].

Frequently, however, the variable causing the dependence is known to the compiler. This is illustrated in Figure 1, which shows one of the most frequently executed loops from SPEC95's *li* application. Each iteration of the loop reads 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.

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

Figure 1: Code segment from *li*.

One approach is for the compiler to insert a synchronization step between the producer and the consumer threads through the memory hierarchy. This approach requires an efficient synchronization mechanism, like a hardware-based full-empty bit mechanism [21]. For example, given an iteration, we know that it will not update *prev* any more when it is about to execute any of the statements (1), (2), (3), or (4) in the figure. At any of these points, an iteration can synchronize with the next thread 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 to provide much faster inter-processor communication. Specifically, we can register-allocate *prev* and use an on-chip network [5, 12, 20, 22] to communicate its value between processors. This approach allows very fast communication.

In a CMP with wide-issue dynamic superscalar processors, performing synchronization through memory is slow. Consider, for example, a CMP with four 4-issue processors where the L2 cache has advanced hardware support for full-empty bit synchronization. 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 can execute many instructions in this time.

If, instead, a fast interconnect is used to communicate these values, 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 0.18μm technology, the entire die can be covered in 2 clock cycles, while for 0.13μm technology, this will increase to 4 cycles [17].

It could be argued that CMPs could be based on single-issue processors, thereby allowing more processors to be configured on-chip. In that case, the inter-processor communication latency is not as crucial. However, exploiting both thread- and instruction-level parallelism is critical for the performance of multithreaded applications [15]. Thus CMPs are likely to be based on wide-issue dynamic superscalar processors.

Note that a fast interconnection may be used to communicate values without supporting register-level communication.

An example is the Superthreaded architecture [24]. Here, an on-chip memory buffer holds the dependent values from/to which a processor loads/stores the value. Fast communication of these values between processors is facilitated using a ring network.

Supporting register-level communication, however, has the advantage that fewer instructions need to be executed. Memory-level communication needs instructions to explicitly store and load the communicated values to and from memory. In addition, unless special hardware support is provided, it also needs instructions to synchronize the two communicating threads. In the example of Figure 1, *prev* must be explicitly stored to memory right before points (3) and (4). In addition, extra instructions for synchronization may need to be added at points (1)-(4). All these instructions may degrade performance [10].

3 Support for Speculation

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 [4, 10, 22, 23, 24] or in hardware by using special hardware support that identifies threads at run-time [16, 19]. We use a software approach. However, we perform the compilation step on the sequential executable file. As a result, we do not need to recompile the program and can operate on legacy codes.

We have developed a binary annotator that identifies units of work for each thread and the register-level dependences between these threads. Currently, we limit the threads to inner loop iterations. In our analysis, we mark the entry and exit points of each loop. During the course of execution, when a loop entry point is reached, multiple threads are spawned to begin execution of successive iterations speculatively. However, we follow sequential semantics for thread completion. The binary annotator is discussed in Appendix A.

Unlike register dependences, memory dependences cannot be easily identified from the binary. Therefore, we assign the full responsibility of detecting memory dependences to the hardware. We designed a modified cache-coherence protocol to enforce data dependences as well as to detect their violation in a distributed manner [12]. When a data dependence violation occurs — for example when a thread generates a datum that a speculative successor thread has already prematurely consumed — threads are squashed and then restarted. We squash the thread that violated the dependence and all its successors. Overall, our cache coherence protocol is similar to others [8, 10, 23]. We do not detail it here because it plays no role in evaluating the importance of communication latency for compiler-identifiable dependence values. Finally, threads can also be squashed for control dependence violations. Specifically, as soon as we identify the last iteration of the loop, any iterations that were speculatively spawned after it are squashed.

4 Evaluation Environment

4.1 Architectures Modeled

We model CMPs with 4 processors, where the processors can be 1-, 2-, or 4-issue dynamic superscalars. The superscalar core is modeled on the lines of the MIPS R10000 [18]. This core has a large fully-associative instruction window along with integer and floating-point registers for renaming. Some of its characteristics are shown in Table 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 multiply and divide operations. Integer multiplies and divides take 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 Funct. Units (int/ld-st/fp)	Entries in Instruction Window	Number of Renaming Regs. (int/fp)
1	1/1/1	16	16/16
2	2/1/1	32	32/32
4	4/2/2	64	64/64

Table 1: Characteristics of the dynamic superscalar core.

We model the memory subsystem in detail. Caches are non blocking and support full load bypassing. We assume a perfect I-cache for all our experiments and model only the D-cache hierarchy. The 4-, 2-, and 1-issue processors can have up to 32, 16, and 8 outstanding memory accesses respectively, of which 16, 8, and 4 can be loads respectively. Each processor in the CMP has a relatively small private L1 cache of 16 Kbytes. All processors share a larger on-chip L2 cache. The characteristics of the memory hierarchy are shown in Table 2.

Parameter	Value
[L1 / L2] Cache Size (Kbytes)	[16x4 / 1024]
[L1 / L2] Cache Line Size (Bytes)	[32 / 64]
[L1 / L2] Cache Associativity	[2 / 4]
L1 Banks	3
L1 Latency (Cycles)	1
L2 Latency (Cycles)	Variable
Memory Latency (Cycles)	26

Table 2: Characteristics of the CMP memory hierarchy. All latencies correspond to a contention-free round trip from the processor.

4.2 Simulation Approach

We evaluate the architectures using an execution-driven simulation environment [11]. Our environment includes MINT as a front-end [25]. The environment captures both application and library code and generates events by instrumenting binaries. The back-end simulator is very detailed and performs a cycle-accurate simulation of the different CMP architectures.

As applications, we use highly-optimized sequential binaries generated by the MIPS compiler. The applications include four programs from the SPEC95 suite (*hydro2d*, *wave5*, *li*, and *ijpeg*), two programs from the MediaBench suite [13] (*adpcm* and *epic*), and two programs from the Perfect Club suite [1] (*trfd* and *ocean*). Table 3 lists the applications. We use the train set as input for the SPEC95 applications and the default input for the rest of the applications. We chose these applications because they are good candidates for speculative execution. The integer applications have many cross-iteration dependences. The floating-point applications, except for *hydro2d*, cannot be parallelized effectively even with advanced parallelizing compiler techniques [2, 9].

Table 3 gives additional loop-level details for each application. The data is collected while executing the applications in sequential mode. The third column (A) gives the total number of loops that are identified and annotated by our binary annotation pass. These are the inner loops of the code. The fourth column gives the percentage of time that is spent in these loops relative to overall execution time. From the percentage of time spent in these loops, we see that the four processors in the CMP will be active over 94% of the serial

Application	Integ. or Float.	Number of Inner Loops (A)	Percent. of Serial Time in (A)	Number of Loops w/ Compiler-Identifiable Cross-Iteration Dependences (B)	Percent. of Serial Time in (B)	Weighted Iteration Grain Size (Instruct.)	Weighted Number of Cross-Iteration Dependences
hydro2d	Float	128	100	23	34	46	1.53
wave5	Float	86	86	25	53	50	1.50
ocean	Float	58	98	16	69	31	1.91
trfd	Float	20	65	13	64	35	3.37
li	Integ	18	50	7	27	31	2.98
ijpeg	Integ	112	79	43	26	70	2.46
adpcm	Integ	1	100	1	100	71	6.00
epic	Integ	25	94	12	35	53	1.50
Harmonic Mean			80		42	44	2.15

Table 3: Loop-level profile of the applications used.

time in four applications, namely *hydro2d*, *ocean*, *adpcm*, and *epic*. They will be active over 79% of the serial time in *wave5* and *ijpeg*. For *li* and *trfd*, there is a large portion of the code that is run serially on one processor. On average, for all the applications, we are able to perform speculative parallelization on over 80% of the serial execution time using our binary analysis.

The fifth column in the table shows how many of the annotated loops have compiler-identifiable cross-iteration dependences. For this category, we ignored dependences caused by induction variables. It must be mentioned that, even though *hydro2d* can be fully parallelized by Polaris [2] or SUIF [9], the optimized sequential binary generated by the MIPS compiler introduces some dependences by assigning to registers those memory locations that are repeatedly loaded in successive iterations. This strategy avoids redundant loads to be issued to the L1 cache, although it forces artificial cross-iteration dependences to appear. This effect occurs to a certain extent in the other floating-point applications, even though these other applications also have true dependences. The sixth column gives the percentage of serial time that is spent in these loops with identifiable dependences. It is this part of the execution time that can be affected by using a fast inter-processor communication scheme as discussed before.

Finally, the last two columns give the iteration grain size and the number of cross-iteration dependences on a weighted basis. Rather than giving a simple average value of all the relevant loops, we give a weighted average: we assign a weight to each relevant loop based on the fraction of serial time spent in that loop. Hence a loop that dominates the execution time will tend to contribute more to the grain size and number of cross-iteration dependences. The fine grain sizes shown in the table highlight the need to use a hardware-based approach to thread initiation and termination as opposed to a software-based one. Furthermore, the last column tells us that, most often, an iteration only needs to communicate a couple of variables to successor iterations.

4.3 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 of the program, the total wasted slots are divided proportionally among the different types of hazards recorded. The different types of hazards that we consider are: waiting on a datum transferred from a predecessor thread (*sync*), data dependences (*data*), waiting on a memory access (*memory*), waiting due to a serial section (*serial*), and instructions squashed on branch mispredictions or when a thread is squashed (*squashed*). There is also an *other* category, which includes slots wasted due to structural hazards, control hazards (restarting the pipeline after a branch misprediction), and due to lack of renaming registers. Finally, the issued instruction slots are grouped under *issued*.

5 Impact of Communication Latency in a Speculative CMP

To evaluate the performance impact of memory-level communication, we simulate CMPs where all the cross-thread dependences that could use register communication are communicated through the L2 cache. The threads are synchronized using a full-empty bit mechanism [21]. We assume special storage beside the shared 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. We consider three types of CMPs that differ in the issue width of their four processors: 1, 2, and 4-issue (Section 4.1). We neglect any port contention to access the communicated variables in the L2 cache. We consider three environments by varying the latency of the L2 cache. In these environments, the one-way trip latency from the processor to the L2 cache is 5, 3, and 0 cycles respectively. In the latter environment, called *Ideal*, there is no communication latency: any producer update is visible to the consumer instantaneously.

Figure 2 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 4.3. For each application, the bars are normalized to the *Ideal* environment. The IPC for each application and L2 cache latency is given at the top of the figure.

From the figure, we can see that the added L2 cache 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 *wave5*, *ocean* and *trfd* have little or no *sync* time in *Ideal*. This denotes 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* has *sync* time. Though this application has practically no serial sections and fully exploits the speculative mode, the performance is severely affected by the presence of several cross-iteration dependences. This invariably results in many instructions being unable to issue, eventually leading up to an execution stall. An increase in the communication latency

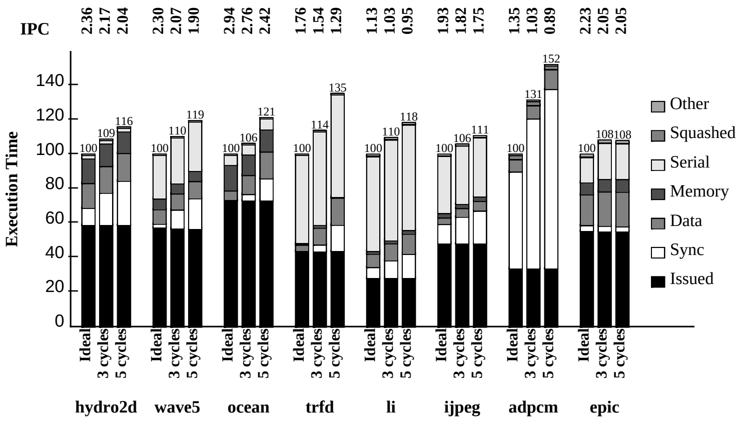

Figure 2: Impact of the L2 cache latency on the execution time and IPC for a CMP based on four 1-issue dynamic processors.

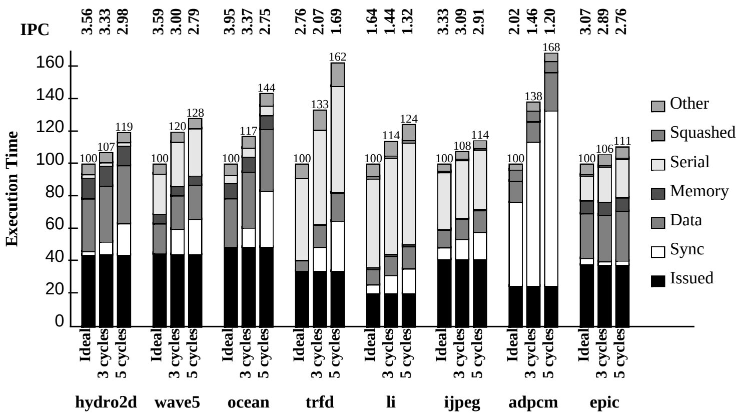

Figure 3: Impact of the L2 cache latency on the execution time and IPC for a CMP based on four 2-issue dynamic superscalar processors.

only exacerbates the problems, resulting in a 50% slowdown relative to *Ideal*. The remaining applications also perform poorly as communication latency increases. Overall, we observe performance losses of 6-31% with a 3-cycle L2 cache access latency and 8-52% with a 5-cycle latency.

Since CMPs will likely be built out of superscalar processors, Figures 3 and 4 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, we see that, for nearly all applications, the performance difference between *Ideal* and the rest has widened. For example, while for the 1-issue processors the average difference in performance between *Ideal* and *5-cycles* is 23%, the difference jumps to 34% for the 2-issue processors (Figure 3) and, finally, to 45% for the 4-issue processors (Figure 4). These results show that there is a 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.

6 Hardware Support for Register-Level Communication

The fast communication needed in speculative CMPs may or may not support register-level communication. However, as mentioned in Section 2, supporting register-level transfers has added benefits. Consequently, we propose to support flexible inter-thread register communication by augmenting a conventional scoreboard to what we call a *Synchronizing Scoreboard (SS)*.

For our hardware to work, each thread maintains its status in the form of a bit mask (called $ThreadMask$) in a special register. The status of a thread can be any of the four values shown in Table 4. Inside a loop, the non-speculative thread executes the current iteration. Speculative successors 1, 2, and 3 execute the successor iterations, which we call the first, second, and third speculative iteration respectively. As threads complete, the non-speculative *ThreadMask* will move from one thread to its immediate successor. In the following, we describe the SS, assess its complexity, and evaluate its performance.

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

Table 4: Possible status of a thread.

6.1 The Synchronizing Scoreboard (SS)

We propose a fully-decentralized structure, where each processor has its own SS. The SS is a scoreboard augmented with additional bits. It is used by a thread to synchronize and communicate register values with other threads. The SSs in the different processors are connected with a broadcast bus, on which register values are transferred. This bus, which we call the *SS Bus*, has a limited bandwidth of 1 register per cycle and one read and one write port to the register file of each processor. For a 4-processor CMP, a value written by a processor onto the bus takes between 1 to 3 cycles without contention to get to the destination processor, depending on the physical distance between the producer and the receiver processors. The latency assumes a 0.13μm chip technology,

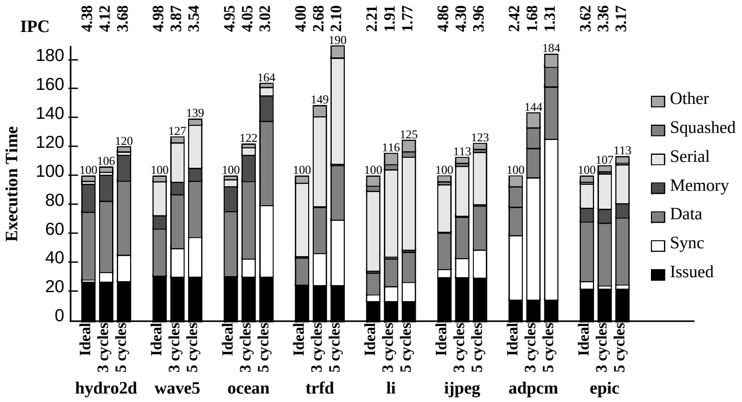

Figure 4: Impact of the L2 cache latency on the execution time and IPC for a CMP based on four 4-issue dynamic superscalar processors.

where a signal takes up to 4 clock cycles to traverse the entire die [17]. After each cycle, the values are latched before being driven to the next stage in the following cycle. Thus, arbitration for the bus is performed one stage at a time. Depending on the direction of the message, the value is stored in one of two directional latches at each stage. A processor cannot write a new value onto the bus when a value is pending in its corresponding staging latches. The overall hardware setup is shown in Figure 5.

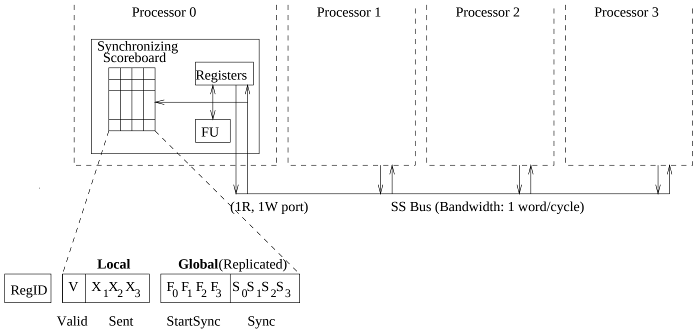

Figure 5: Hardware for register communication.

6.1.1 Data Structures

As in a conventional scoreboard, each SS has one entry per register. Figure 5 shows the different fields for one entry. The fields are grouped into *local* and *global* fields. The local fields are private to each processor. To avoid centralization, the global fields are replicated but easily kept coherent across the SSs in the different processors. This is described later in the section. The global fields include the *Sync (S)* and the *StartSync (F)* fields. Each of these fields has one bit for each of the processors on chip. Table 5 shows an example of the global fields of a SS.

RegID	StartSync	Sync
	$F_0F_1F_2F_3$	$S_0S_1S_2S_3$
...		
13	0 1 0 0	0 1 0 0
14	1 0 1 0	1 0 0 0
...		

Table 5: Example of the global fields of a SS.

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

The F_i and S_i bits for all the registers are automatically initialized with dedicated hardware. They are set in the SS of all processors when a thread starts on processor i. The F bit simply keeps the value that S was given to when the thread was initiated in the processor. From then on, the F bit remains unchanged throughout the execution of the thread. The F_i bits are used to indicate the looplive registers that may be generated at any time during the iteration by the thread running on processor i.

The private fields include the *Valid (V)* and *Sent (X)* fields. We will consider the X field later. The V bit for each register 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 (the latest speculative thread), it sets the V bit for each of its registers as: $V = V - \cup F_{pred}$. This invalidates any registers that are written by any of the three predecessor threads.

Note that the startup overhead for speculative task initiation involves just setting a register bit-mask for the corresponding thread ID in the SS and initializing the program counter to start execution at a specific location. This can be achieved with modest hardware in a single cycle.

6.1.2 Communication

Register communication between threads can be *producer-initiated* or *consumer-initiated*. The producer-initiated approach has already been outlined. When a thread performs a safe definition for a register or executes a release instruction for a register, it clears the S bit for the register and writes the register on the SS bus. At that point, in hardware, each of the successor threads checks its own V bit for the register and also the F bits for all the threads between the producer (non-inclusive) and itself (inclusive) for the same register. If all these bits are zero, the hardware in the successor thread automatically loads the register and sets the V bit of the corresponding register to 1. At the same time, the hardware also clears the S bit corresponding to the producer thread in all the SSs. The F bits, however, remain unchanged.

It is possible that the consumer thread is not yet running when the producer generates the register. We could allow the values to be stored by using a buffered communication mechanism, rather than using a simple broadcast bus. The buffer would have to potentially hold all the live registers after the last speculative thread until a new thread is initiated on the successor. In addition, this approach would require further hardware support in the form of duplicate register sets in each processor to enable recovery from squashes [3]. Alternatively, a global register set may be maintained to store these values [20], but at the cost of maintaining an additional centralized structure.

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

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

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

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

The accesses to these bits are always masked out with the $ThreadMask$ of Table 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 4. This means that we are examining only 2 predecessors. The function is:

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

Overall, the complete logic to determine whether a register is available is shown in Figure 6. This 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 F bit is set (the thread should generate the register) and S bit is clear (the thread has already generated it). 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 distance between producer and consumer, plus the contention for the SS bus.

Since the S and F bits are decentralized, it is the responsibility of the hardware in each of the processors to automatically update the bits. Since there is a delay in the SS bus, for a short period of time, the bits may be inconsistent across processors. However, the protocol has been designed such that there is no effect on the correctness of the overall mechanism. For example, when a producer processor writes a register on the SS bus, the S bit for the producer will be reset. During a short period of time, the bit will be 0 in the processors closer to the producer while it is still 1 in the processors far away from it. However, when the register value from the producer reaches the end processor, the S bit for the producer processor in the end processor is zeroed out.

6.1.3 Example

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

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

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

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

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

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

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

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

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

Now, when $t+4$ tries to read $r3$, it checks 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 from the non-speculative thread $t+1$. At that point, it puts a request on the SS bus. This is a consumer-driven approach. Finally, when processor 1 supplies the value to the bus, processor 0 reads register $r3$.

6.1.4 The Last Copy Problem

When the last speculative thread updates a looplive register, it has no successors to which it can send the value. As a result, any future consumer threads will have to explicitly request 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 7. In the example, register $r3$ is live across all threads. Each thread reads $r3$ before writing it. Therefore, any thread will invalidate its local copy of $r3$ on initiation. In Figure 7-(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; and

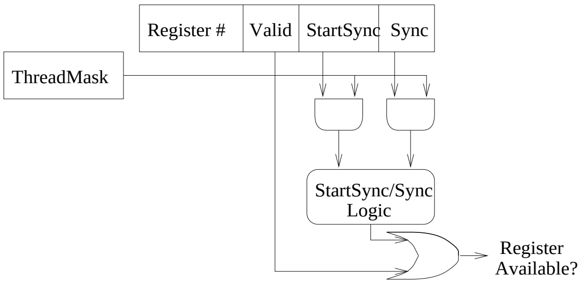

Figure 6: Logic to check register availability.

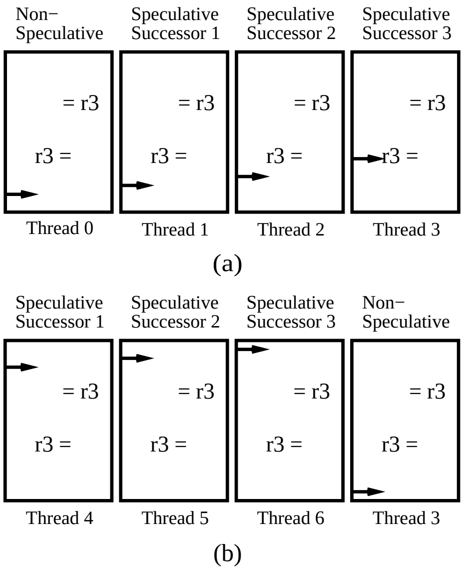

Figure 7: The last copy problem. The arrow points to the currently-executing instructions.

thread 3 has acquired non-speculative status and is about to finish (Figure 7-(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, while thread 7 will clear its valid bit for $r3$ upon initialization.

The last copy problem will not occur if we use a communication mechanism that buffers live-out values to percolate to the new threads, or when there is a centralized global register set that maintains live-out values. For instance, the Multiscalar processor [22] uses the first approach. A ring structure is used to forward register values. 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 must be 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 [3]. This is in contrast to our scheme, which supports both a producer- and a consumer-driven approach, thereby simplifying recovery from squashes. Squashed threads that are restarted simply re-request the register values from the appropriate producer.

The Trace processor [20] avoids the last-copy problem by keeping a centralized global register set that is visible to all processors. This is in addition to the local register set in each processor. Live-out register values are sent to the global registers, from where any processor can read them.

In both of the above approaches, the architecture must provide significant hardware support for speculation. Unfortunately, all these resources remain unutilized when running applications that do not need speculative parallelization.

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 forwarded the register to. This includes both producer- and consumer-initiated transactions. Consequently, each processor has 3 private bits per register called the *Sent (X)* bits. They are set if the register has been sent to the corresponding thread. 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 the looplive registers (those with the F_0 bit set) that it produces, 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 looplive value has reached up to the thread that kills the value. If F_i is set, then that thread kills the value. 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 for last-copy status. Finally, when a new thread is initiated on a processor, the remaining processors clear the corresponding X bit, thereby noting that the value is yet to be sent to the new thread.

6.2 Complexity of the SS

To understand the cost of the SS mechanism, we examine its area and its potential impact on the processor's cycle time. To estimate the area, we need to consider first the logic to check for register availability and last-copy status (Sections 6.1.2 and 6.1.4). The AND-OR logic, which is traditionally implemented as a carry-propagate-kill function, and the extra gates to selectively mask out some of the S, F, and X bits require only a few gates. Replicating this logic for each register implies an extra overhead of only a few hundreds of gates even for a processor with a large number of registers.

In addition, the register file in each processor needs some extra space to store the V, X, F, and S bits. The number of extra bits per register is $3n$, where n is the number of processors on chip (Figure 5). For a 4-processor CMP with 64-bit registers, this works to around 12% storage overhead.

Finally, we need to include a SS bus in the chip. Overall, however, we feel that these are modest hardware requirements when compared to replicating the register sets in each processor [3] or using a centralized global register file [20].

As for the impact on cycle time, if we refer to equation (1) in Section 6.1.2, it may seem that, in the worst-case scenario where all the bits have to be considered, the delay incurred by the SS logic increases quickly with the number of processors in the CMP. However, by using a binary-tree approach, the logic can be implemented using only $\log_2 n$ levels of gates, where n is the number of processors in the CMP. Consequently, this circuitry is shallow and unlikely to affect the cycle time. Furthermore, the SS bus is implemented with staging buffers. By pipelining the bus in this manner, we likely eliminate any adverse impact on the processor cycle time.

Finally, we note that the complexity that we add to the register files is modest: we add only one read and one write port to each register file.

6.3 Evaluating the SS Performance

We now evaluate a CMP augmented with our SS and compare its performance to the *Ideal* environment of Section 5. Recall that, in *Ideal*, there is no communication latency: any producer update is visible to the consumer instantaneously. Figure 8 shows the execution time of the eight applications on a CMP with four 4-issue processors under our SS hardware and under *Ideal*. The execution time is normalized to *Ideal*. In our simulations, we assume a SS bus with a high bandwidth (5 words per cycle), so that we can factor out the effect of contention. We will reduce the bandwidth later. Registers are 1-word wide. Recall that the request and reply messages in the SS can take 1-3 cycles each.

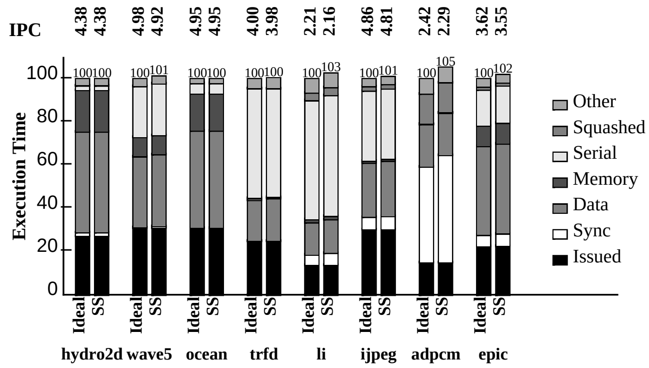

Figure 8: Performance of a CMP with four 4-issue processors under our SS hardware and under *Ideal*.

The figure shows that, for all the applications, the performance of SS is very close to that of *Ideal*. The SS introduces very little overhead. This figure should be compared to Figure 4, which compared *Ideal* to CMPs where all communication occurred via the L2 cache. That figure showed that communication via the L2 cache slows down the applications by an average of 23% (for a 3-cycle one-way access to the L2 cache) and 45% (for a 5-cycle access). These results, therefore, indicate that fast communication is very beneficial.

The fact that there is a 3-cycle latency between processors that are located far apart does not seem to affect the performance much. 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 [6], which indicated that, in 70-80% of the cases, the register values are consumed by the immediate successor thread. Another factor that helps reduce the effect of latency is the support for producer-initiated communication. It avoids unnecessary delay when the consumer finally needs the value. Finally, register communication is also faster because it needs fewer instructions than memory communication.

We now change the bandwidth of the SS bus to determine its impact on performance. Figure 9 shows the execution time of the applications for values of bus bandwidth ranging from 1 word per cycle to infinite bandwidth. For each application, the execution time is normalized to the time taken when the bandwidth is 1 word per cycle. From the figure, we can see that there is little performance gain in increasing the bandwidth of the SS bus beyond one word per cycle. In fact, an environment with infinite bandwidth is less than 5% faster. Consequently, we suggest a bus bandwidth no higher than 1 register per cycle.

Overall, we conclude that support for fast communication is quite beneficial for CMPs with wide-issue dynamic superscalars. In addition, we have shown that this support can be provided at the register level with modest hardware requirements.

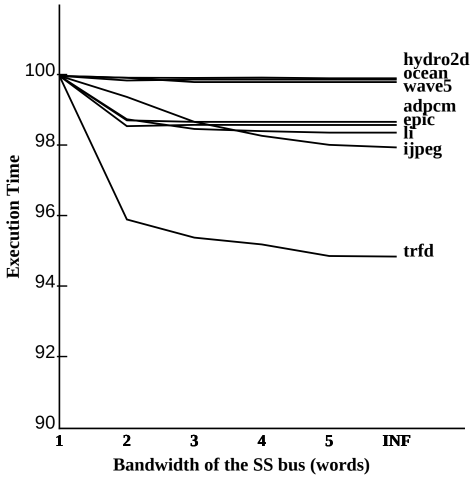

Figure 9: Impact of changing the bandwidth of the SS bus on the execution time.

7 Conclusions

Chip-multiprocessor architectures (CMP) are a promising design alternative to exploit the ever-increasing number of transistors that can be put on a die. Since CMPs must also handle applications that are difficult to parallelize, much effort has gone into providing support for speculative parallelization. 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 a plain memory subsystem for communication between processors degrades the performance and that hardware support for fast communication is required. We also proposed a hardware scheme that enables a CMP to perform communication and synchronization at the register level. The hardware support is modest, yet effective enough to allow the applications to deliver near ideal performance.

References

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

[2] 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.

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

[4] 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.

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

[6] 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, December 1992.

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

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

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

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

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

[12] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *Special Issue on Multithreaded Architecture, IEEE Transactions on Computers*, December 1999.

[13] 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.

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

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

[16] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *13th International Conference on Supercomputing (ICS)*, June 1999.

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

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

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

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

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

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

[23] 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.

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

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

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

Appendix A: Binary Annotation

The steps involved in the annotation of the binary are illustrated in Figure 10. The approach that we use is similar to that of Multiscalar [22], except that we operate on the binary code instead of on the intermediate code. First, we identify loop iterations and annotate their initiation and termination points. Then, we identify the register-level dependences between these threads. This involves identifying *looplive* registers, which are those that are live at loop entry and exits and may also be redefined in the loop. We then identify the *reaching definitions* at loop exits of all the looplive registers. From these looplive reaching definitions, we identify *safe definitions*, which are definitions that may occur but whose value will never be overwritten later in the loop body.

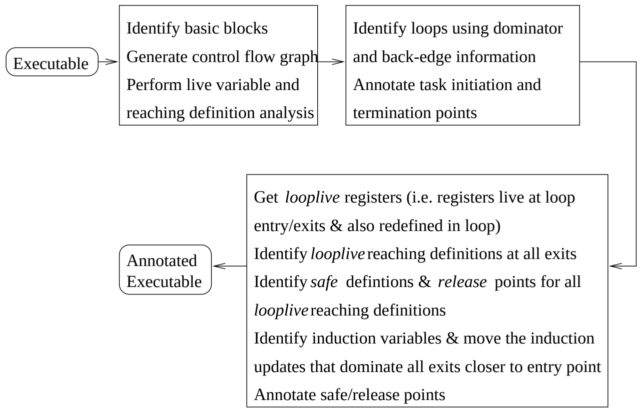

Figure 10: Binary annotation process.

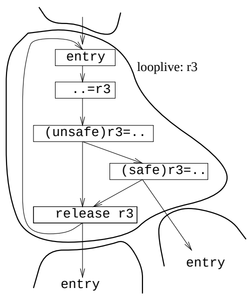

Figure 11: Safe definitions and release points.

Similarly, we identify the *release points* for the remaining definitions whose value may be overwritten by another definition. Figure 11 illustrates the *safe definitions* and release points for a looplive register $r3$. These points are identified by first performing a backward reaching definition analysis. This is followed by a depth-first search, starting at the loop entry point, for each and every looplive reaching definition. Finally, induction variables are identified and their updates are percolated closer to the thread entry point provided the updating instructions dominate the exit points of the loop. This reduces the waiting time for the succeeding iteration before it can use the induction variable. However, to evaluate the effect of communication latency, we do not consider induction variables. Instead, we consider only true cross-iteration dependence variables.

Incorporating these additions in a binary is quite simple and requires only minor extensions to the ISA. Additional instructions are needed only to identify thread entry, exit, and register value release points.

At present, our current approach of analyzing sequential binaries is restricted to inner loop iterations. Consequently, we can only examine applications which are largely loop-based. However, we believe that the approach can be easily expanded to include other sections of the code by using heuristics similar to those used for task selection in the Multiscalar processor [26].