UNIVERSIDAD DE ZARAGOZA

Tesis Doctoral

Prebúsqueda Hardware, Soporte para Reducción, y Almacenamiento de Estado Especulativo en Multiprocesadores de Memoria Compartida.

Autor: María Jesús Garzarán Arnau

Directores: Dr. D. Víctor Viñals Yúfera

Dr. D. José María Llabería Griñó

Dr. D. Josep Torrellas Jovaní

gaZ

CENTRO POLITECNICO SUPERIOR

DEPARTAMENTO DE INFORMATICA E INGENIERIA DE SISTEMAS

GRUPO DE ARQUITECTURA DE COMPUTADORES DE ZARAGOZA

Prebúsqueda Hardware, Soporte para Reducción, y Almacenamiento de Estado Especulativo en Multiprocesadores de Memoria Compartida

Memoria presentada para optar al grado de
Doctora Ingeniera en Informática por

Doña María Jesús Garzarán Arnau

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
Marzo de 2002

Dirigida por

Dr. D. Víctor Viñals Yúfera
Dr. D. José María Llabería Griñó
Dr. D. Josep Torrellas Jovaní

Departamento de Informática e Ingeniería de Sistemas

Universidad de Zaragoza

A mis padres Santiago y Angeles, y a mis hermanos.

Acknowledgements

First of all, I would like to thank my thesis advisors for their guidance and support. Víctor Viñals introduced me to the interesting world of multiprocessors and encouraged me to work in this area. Jose María Llabería has always been very supportive. He has always been ready to work with me, and to talk with me on the phone whenever I needed to discuss something. I also want to thank Josep Torrellas for hosting me during my stay at Urbana-Champaign, and for suggesting that I work on speculation. Special thanks to all of them, not only for their advising, but also for their attention and friendship.

I am also very grateful to all the members of the I-ACOMA group at Urbana-Champaign and my colleagues at the architecture group in Zaragoza. I especially thank Milos Prvulovic, my work-mate during my stay at Illinois; I had many insightful discussions with him. Special thanks also to José Martínez for all his encouragement. He has always been one of my strongest supports at Illinois. José Luis Briz and Pablo Ibáñez have also been very supportive during all these years. They helped me with the initial work on hardware prefetch and with reading this thesis.

Thanks also to my friends Pablo, Juan, Silvia, and María Angeles. They have been truly good friends, who have always listened to me and offered their help. I also owe special gratitude to Luis, for all the support he has given me.

Finally, I want to thank my parents, my brothers and their wives, and the rest of my family. I dedicate the work of all these long years to my parents and brothers. I could not have done it without their love and support.

This work has been supported in part by the Ministry of Education of Spain under contracts TIC 1998-0511-C02-02 and TIC 2001-0995-C02-02; National Science Foundation under grants CCR-9734471M ACI-9872126, EIA-9975018, CCR-9970488M, EIA-0081307, and EIA-0072102; and by gifts from IBM, Intel, and Hewlett-Packard.

Contents

1 Introduction **1**

1.1 Motivation 1

1.1.1 Hardware Prefetch 4

1.1.2 Parallelizing Reductions 5

1.1.3 State Buffering in Speculative Thread-Level Parallelization 6

1.2 Thesis Contributions 8

1.3 Thesis Organization 9

2 Pattern Characterization **17**

2.1 Introduction 17

2.2 Measured Patterns and Environmental Setup 19

2.2.1 Measured Patterns 19

2.2.2 Workload and Methodology 23

2.3 Pattern Distribution 23

2.4 Sequence Length 27

2.5 Summary 29

3 Hardware Prefetch in Bus-based Multiprocessors **33**

3.1 Introduction 33

3.2 Related Work 34

3.3 Environmental Setup 37

3.3.1 System with prefetch 38

3.3.2 Workload and Methodology 40

3.4 Performance Model 40
3.5 Evaluation 42
3.5.1 Base System 42
3.5.2 Discussion of Prefetching Alternatives 42
3.6 Summary 48

4 Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors 53

4.1 Introduction 53
4.2 Parallelization of Reductions in Software 55
4.2.1 Background Concepts 55
4.2.2 Parallelizing Reductions 56
4.2.3 Drawbacks in Scalable Multiprocessors 58
4.3 Private Cache-Line Reduction (PCLR) 58
4.3.1 Overview of PCLR 59
4.3.2 Implementation of PCLR 60
4.3.3 Summary 66
4.4 Evaluation Methodology 67
4.4.1 Simulation Environment 67
4.4.2 Applications 68
4.5 Evaluation 69
4.5.1 Impact of PCLR 69
4.5.2 Scalability of PCLR 72
4.5.3 Impact of FP-Unit Speed 72
4.6 Additional Use of PCLR 73
4.6.1 Dynamic Last Value Assignment in Software 73
4.6.2 Using PCLR for Last Value Assignment 74
4.6.3 Advanced Support 75
4.7 Related Work 77
4.8 Summary 77

5 Tradeoffs in State Buffering for Speculative Thread-Level Parallelization 83

5.1 Introduction . . . 83

5.2 Buffering State Under Speculative Parallelization . . . 85

5.2.1 Basics of Speculative Parallelization . . . 85

5.2.2 Challenges in Buffering State . . . 88

5.2.3 Application Behaviour . . . 90

5.2.4 Analogy to Register File Management . . . 92

5.3 Taxonomy of Approaches . . . 93

5.3.1 Speculative State Composition . . . 93

5.3.2 Novel Taxonomy of Approaches . . . 94

5.3.3 Mapping Proposed Schemes to the Taxonomy . . . 95

5.4 Implementation Issues . . . 99

5.4.1 Implementing a Distributed MROB . . . 99

5.4.2 Implementing a Distributed MHB . . . 100

5.4.3 Operations To Support in a Distributed MROB and MHB . . . 103

5.5 Tradeoff Analysis . . . 104

5.5.1 Architectural vs Future Main Memory . . . 104

5.5.2 Single vs Multiple Speculative Tasks & Versions per Processor . . . 107

5.5.3 Eager vs Lazy Merging with Main Memory State . . . 109

5.6 Evaluation Methodology . . . 113

5.6.1 Simulation Environment . . . 113

5.6.2 Applications . . . 114

5.7 Evaluation . . . 116

5.7.1 Single vs Multiple Speculative Tasks/Versions per Processor . . . 116

5.7.2 Eager vs Lazy Merging with Main Memory State . . . 117

5.7.3 Architectural vs Future Main Memory System . . . 119

5.7.4 Summary . . . 121

5.8 Conclusion . . . 121

6 Software Buffering Under Speculative Thread-Level Parallelization 127

6.1 Introduction . . . 127
6.2 Speculation Protocol Used . . . 129
6.3 Overview of the History Buffer Operation . . . 130
6.4 Efficient Software Implementation . . . 131
6.4.1 Accessing Local Task-IDs in Software . . . 132
6.4.2 Caching Task-IDs . . . 134
6.4.3 Quantifying Insertion Overhead. . . . 134
6.4.4 Filtering the First Store . . . 136
6.4.5 Exposed Loads that Need to Be Buffered . . . 139
6.4.6 Other Issues . . . 140
6.4.7 Correct Interleaving . . . 141
6.4.8 Hardware-Intensive Solution . . . 143
6.5 Evaluation Methodology . . . 144
6.5.1 Simulation Environment . . . 144
6.5.2 Applications . . . 145
6.6 Evaluation . . . 146
6.6.1 Impact of Software Buffering on Execution Time . . . 147
6.6.2 Implementation Overheads of Advanced Buffering . . . 148
6.6.3 Alternative Designs for Software Logging . . . 150
6.7 Summary . . . 154

7 Conclusions and Future Work **157**
7.1 Conclusions . . . 157
7.1.1 Hardware Prefetch . . . 157
7.1.2 Parallelizing Reductions . . . 159
7.1.3 State Buffering in Speculative Thread-Level Parallelization . . . 160
7.2 Future Work . . . 161

Appendix A: Speculation Protocol for MultiT&MV lazy FMM systems **163**

Appendix B: Description of the Applications **175**

Appendix C: Example of Poor MultiT&SV Performance **177**

References **179**

List of Figures

2.1 Sparse vector. 21

2.2 Sparse vector stored using the LL format. 21

2.3 Breakdown of pattern distribution for 16 processors 24

2.4 Breakdown of pattern distribution for Cholesky and Radix (1-32 processors) 25

2.5 Effect of removing the first three loads of each sequence 27

2.6 Changes in sequence length with the number of processors. 29

3.1 Base system used for comparing prefetching techniques. 38

3.2 Performance mode. Demand and prefetch miss ratios are shown. 41

3.3 Speedup, Ta and BU vs. number of processors, for the Base system 43

3.4 Demand and prefetch miss ratios 45

4.1 Loop with anti and flow dependences. 55

4.2 Loop with a reduction operation. 56

4.3 Code resulting from parallelizing the loop in Figure 4.2. 57

4.4 Representation of how PCLR works. 59

4.5 Parallelized reduction code under PCLR. 60

4.6 Code using a swap instruction to solve the atomicity problem. 65

4.7 Execution time under different schemes for a 16-node multiprocessor 70

4.8 Speedups delivered by the different mechanisms (harmonic mean). 72

4.9 Comparing the performance with floating-point units of different frequencies. 73

4.10 Loop to be parallelized with privatization and dynamic last value assignment. 74

4.11 Code resulting from parallelizing the loop in Figure 4.10 with dynamic last value assignment. 75
4.12 Parallelized reduction code with dynamic last value assignment under PCLR. 76

5.1 Several tasks executing under speculative execution. 86
5.2 Multiple versions of the same variable in the system 89
5.3 Cache keeping versions of different tasks and addresses 89
5.4 Multiple local speculative versions. Last versions and non-last versions. . . 90
5.5 Examples of non-analyzable loops with mostly-privatization access patterns 92
5.6 Speculative State Composition 93
5.7 Taxonomy of approaches to manage multi-version speculative state 94
5.8 Mapping schemes for speculative parallelization onto our taxonomy 96
5.9 Implementing a distributed MROB. 100
5.10 Data structure organization where non-last versions are remapped. 101
5.11 Implementing a Distributed MHB. 102
5.12 Example of tasks executing under SingleT, MultiT&SV, and MultiT&MV . 108
5.13 Progress of the execution and commit wavefronts for a lazy AMM/FMM system. 109
5.14 Progress of the execution and commit wavefronts in different schemes: eager AMM, lazy AMM with stalls, eager AMM with stalls, and eager FMM . . . 110
5.15 Application characteristics that limit performance in each approach. 112
5.16 Effect of supporting single or multiple speculative tasks or versions per processor. 117
5.17 Effect of supporting eager or lazy merging with main memory state. 118
5.18 Effect of limited support for eager or lazy merging with main memory state. 119
5.19 Effect of supporting an architectural or a future main memory system. . . . 120

6.1 Per-processor structures that we use for the history buffer. 130
6.2 Special mapping to access task-IDs from software and Address Translation Module. 133
6.3 Instructions added before a speculative store. 135

6.4 Stores to the array B are non-speculative but need to be buffered. 136
6.5 Filtering first stores 137
6.6 Eliminating the branches with the use of predication 138
6.7 Example that may result in incorrectly-buffered information. 142
6.8 Execution time under different schemes for a 16-node multiprocessor. . . . 147
6.9 Breakdown of dynamic stores. 149
6.10 Effect of eliminating first-store filtering. 151
6.11 Effect of log recycling and L1 cache bypassing for logs. 152
6.12 Comparing filtering with task IDs to filtering with extended loads. 153

7.1 Sliding commit of Tasks 163
7.2 Address Mapping Module. 166
7.3 Per-processor structures required for the Memory History Buffer. 167
7.4 Modifications required in a DSM machine to support the complete speculation protocol. 168
7.5 Local and Home algorithm for the speculation protocol 169
7.6 Recovery in an out-of-order RAW dependence infraction. 173
7.7 Example where MultiT&SV is slower than SingleT. 177

List of Tables

2.1 Access patterns description 22

2.2 Evaluated subset of SPLASH-2 23

2.3 Dominant sequence length for different access patterns 28

3.1 Basic parameters 39

3.2 Selected cache sizes 40

3.3 L1 demand miss ratios of loads for the Base system. 44

3.4 Speedups of LCm and LCms and Ts relative to the Base System 47

4.1 Architectural characteristics of the modelled CC-NUMA. 67

4.2 Application characteristics 68

4.3 Lines flushed or displaced in loops with reduction statements 71

5.1 Application characteristics that illustrate the difficulties of buffering. 91

5.2 Application characteristics 115

5.3 Qualitative characteristics of the Applications. 116

6.1 Applications characteristics 146

6.2 History buffer statistics 149

6.3 Effect of task IDs loads on the L1 miss rate. 150

7.1 How to handle dependences between tasks. 171

Chapter 1

Introduction

The context that embraces all the research presented in this thesis is **Cache-Coherent Shared Memory Multiprocessors**.

"Multis are a new class of computers based on multiple microprocessors. The small size, low cost, and high performance of microprocessors allow design and construction of computer structures that offer significant advantages in manufacture, price-performance ratio, and reliability over traditional computer families ... Multis are likely to be the basis for the next, the fith, generation of computers" [Bel85].

1.1 Motivation

Shared Memory Multiprocessors offer the programmer a single memory address space that all the processors share. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. With shared memory, since processors operate in parallel they need to coordinate when accessing shared data. Thus, when sharing is supported with a single address space, a separate mechanism for synchronization is provided. One approach is to use atomic read-modify instructions operating on lock variables. With locks, only a processor can enter the critical section, and the other processors interested in the shared data must wait until the winner processor unlocks the variable.

Shared memory multiprocessors can have two different organizations. In the first one, processors and memory are connected through an indirect interconnection network, typically a shared bus and main memory is physically centralized. In the second one, memory is physically distributed in nodes containing one or more processors, and these nodes are connected through a direct interconnection network, like a mesh or a hypercube. Dis-

tributed memory multiprocessors are usually called DSM (Distributed Shared Memory) machines [NAB+95] or CC-NUMA(Cache-Coherent Non Uniform Memory Access) machines [SJG92]. In this work, we will use the two terms indistinctly. In both organizations caches are used to reduce the memory latency and increase the processor utilization. They are kept coherent by means of coherence protocols: snoopy protocols in the first case, and directory-based protocols in the second one. In the bus-based organization, the bus limits the bandwidth to access the memory. Thus, this organization does not scale as the number of processors increase, unless we use very expensive networks like multistage networks or crossbars. In the second organization, the memory is distributed, and the memory bandwidth grows as the number of nodes increases, which makes this organization to be scalable.

These multiprocessor systems were designed to reduce the execution time of a single application containing parallel threads or tasks. However, up to now the most important use of the multiprocessors has been the execution of a multiprogrammed workload where independent processes execute in parallel instead of time-sharing a single processor. Several reasons can explain why multiprocessors are not very often used to execute parallel programs. The first one is that it is hard to find parallel programs. On one side, parallel programming is not easy. Thus, only a few applications have been re-written so that they can execute in parallel. On the other side, despite the advances in compiler technology, there are still not many codes that the compiler can fully parallelize. The second reason is that, when multiprocessors execute parallel programs, they usually deliver lower speedups than expected. The main cause for this is the additional main-memory accesses that programs running on a multiprocessor require. In addition, a main-memory access in a multiprocessor is typically slower than in a uniprocessor: interconnection networks have additional delays and the machine is physically bigger.

In this thesis, we address these two problems, and propose several extensions or modifications. Our goal is to reduce the execution time of a single application executing in a shared memory multiprocessor. In particular, we have considered hardware prefetch and reduction support for parallelized codes, and speculative execution support for sections of parallel codes that the compiler could not parallelize.

Prefetch is a well known technique that can speed-up a program by preloading data in cache before the processor needs them. Prefetch is only effective if the prefetcher can learn and reproduce in advance the patterns that the program is following when accessing memory. However, there is no systematic study of the data access patterns that appear in parallel programs. In this thesis, we perform that study for a set of relevant patterns. In addition, we measure if the distribution pattern distribution changes as the

number of processors executing the application changes. We use this study as the starting point for designing a low-cost hardware prefetch mechanism for bus-based multiprocessors. Prefetching in this environment is difficult due to the limited bandwidth resources.

The second technique that we have selected to speed-up the execution of parallel programs is to provide architectural support for parallel reductions. We have observed that reduction operations appear in many scientific codes that the compiler can parallelize. Reductions are important and time-consuming operations. However, finding the most appropriate transformation for a given reduction is usually hard. This is even harder in case of sparse, dynamic applications, where the compiler does not usually has knowledge about the data that each processor needs to access. With DSM machines, things become worse, because the latency to access memory is variable, and depends on the data layout. In this context, we propose new architectural supports that speed-up parallel reduction and make them scalable. These supports require small modifications to the directory of a DSM machine.

Finally we consider speculative thread-level parallelization. Using this approach we can extract tasks of sequential codes that the compiler cannot fully analyze, and speculatively execute them in parallel. The multiprocessor is modified with extra hardware to dynamically detect dependences. If the speculative parallel execution is incorrect a combination of hardware and software repairs the state in memory and resume with the parallel execution. In order to be able to repair the corrupted state in memory, the speculative state produced by these parallel tasks needs to be kept. Given the distributed nature of the memory and caches in a multiprocessor, this buffering is challenging. In this thesis we address the problem of buffering speculative memory state in speculative thread-level parallelization for DSM machines.

Note that this work considers two very different, but important, hardware environments. Prefetch is developed for bus-based multiprocessors, while parallel reduction support and speculative state buffering is developed for scalable DSM machines. Many commercial designs follow nowadays the first trend, like the bus-based multiprocessor Sun Enterprise E450 with up to 4 UltraSPARC processors, or the L2000 of HP with up to 4 PA8500 or PA8600 processors. On the other hand, despite their low commercial presence (SGI Origin is the obliged reference), DSM machines seem to be the straightest way to achieve significant speed-ups in applications requiring access to big amounts of shared memory [BC02].

1.1.1 Hardware Prefetch

One of the mechanisms that we propose to increase the multiprocessor performance is the ability to predict the addresses that the processor will issue in the near future. These addresses are used to prefetch data into the levels of the memory hierarchy closer to the processor. A well-managed data prefetching can hide the latency to access memory, and thus speed-up the application.

Data prefetch can be software [KCPT95, MG91, TE95], when either the programmer or the compiler add extra instructions to prefetch data. It can be hardware [DDS93, CC98], when the hardware of the processor is modified to dynamically predict future addresses. Finally, it can also be hybrid [CB94, ZT95] when hardware and software collaborate in the data prefetch.

In this thesis we are interested in hardware mechanisms. These mechanisms were initially proposed for uni-processors. Simple mechanisms are those able to predict sequential accesses [Smi78], or accesses that differ in a constant stride [CB94, FP92]. More complex hardware mechanisms are able to detect the patterns of single loads that access records chained by pointers and 4-bytes elements of an index list [IVBG98, MH96]. Recent proposals, and also in the uniprocessor context, perform a dependence analysis to find a relation between pairs of loads traversing a linked list [RIVL00, RMS98], or correlate a sequence of misses for a particular miss reference address [JG99].

All previous proposals on hardware prefetch mechanisms track the regularity in the address stream issued by the loads that the processor executed in the past, and based in this regularity try to predict the future addresses. Thus, to guide the research of new mechanisms for multiprocessor systems, it is important to know the data access patterns that appear in parallel programs. It is also important to know how the access patterns and its distribution changes as the number of processors executing the application changes. No systematic study on pattern characterization has been done. The work in [DS96] did some, and measured the burst length of certain patterns. However, no study on pattern characterization that took into account the variation in the number of processors was done. We study all these issues in Chapter 2.

On the other hand, little work has been done on hardware prefetch for bus-based multiprocessors. Sequential and stride-based mechanisms have been evaluated in the context of multiprocessors [CB94, DDS93, DS96, DS98]. But in these works processors are connected through a interconnection network, where latency is assumed to be low. Also, results in [DS95, DS96] show that prefetch can increase the traffic significantly, which can become a serious bottleneck in a bus-based multiprocessor where memory bandwidth is

limited. In the context of bus-based multiprocessors, the most important work has been presented in [TE95, TE93]. This work is based on software prefetch, and the evaluation is done for a fixed number of processors.

In Chapter 3 we focus in hardware prefetch mechanisms for bus-based multiprocessors, that are able to detect simple patterns like the sequential and stride one, and more complex ones like single load linked list and index list traversal [IVBG98, MH96]. One of the goals of this research is to correlate the variation in the number of processors with the effectiveness of the prefetch.

1.1.2 Parallelizing Reductions

Reduction operations are frequently found at the core of many scientific applications. A reduction occurs when an associative and commutative operator $\otimes$ operates on a variable x as in $x = x \otimes expression$. x is called the reduction variable, and it can be a scalar or an array element. If the reduction operation is in a loop and certain conditions are satisfied the loop can be parallelized. However, a loop with a reduction operation can have complex flow dependences across iterations, and therefore special transformations are required. Several transformations have been proposed, but it is usually hard to know which of them is the most appropriate to parallelize the loop. The election depends on many parameters like the degree of contention of the reduction element (number of iterations referencing it) or the sparsity (ratio of referenced elements to the dimension of the array) that can only be known at run-time. In addition, many of these transformations do not scale as the number of processors increases.

For low-contention reductions, a common used mechanism is to enclose the access to the reduction variable in an unordered critical section [EHLP91, Zim91], or with an atomic *fetch-and-op* operation. The main drawback of this method is that it is not scalable, as the contention for the critical section increases with the number of processors.

Another mechanism that has been used to parallelize reductions is based on the use of inspector-executor loops. An approach based on this inspector-executor mechanism is GatherScatter [DUSH94]. With the GatherScatter approach an inspector loop identifies non-local data needed by each processor, and generates a communication schedule. Then, a executor loop uses the communication schedule to gather nonlocal data, perform the computation using local buffers, and scatters non local results to the appropriate processors. Another different approach also based in the inspector-executor is LocalWrite [HH98]. With LocalWrite the computation partitioning is done so that each processor only computes new values for locally owned data. By writing only to local data, LocalWrite avoids the overhead of mapping nonlocal indices into local buffers (i.e. address translations), or

mutual exclusion synchronization; however, LocalWrite must replicate computation when an iteration writes to data placed in different nodes. In addition, both approaches incur the overhead of executing the inspector loop. Usually, this overhead is amortized if the loop is executed several times, but when the loop is adaptive the inspector has to be executed every time that the access pattern changes.

Finally, another approach to parallelize reductions is to exploit the fact that a reduction is a commutative and associative recurrence, and thus it can be parallelized using a recursive doubling algorithm [Kru86, Lei92]. Several implementations of this method have been proposed [Pot97, LP98]. One of them is the Replicated Private Arrays. In this implementation, the reduction variable is privatized using replicated arrays in all the processors, and the partial results are accumulated on each processor. After the parallel loop finishes, results from the replicated arrays are combined in a final merge step. This merge step is usually implemented with a cross-processor algorithm, that suffers from many remote misses. In addition, the execution time of this merge step is proportional to the size of the array, and does not decrease when more processors are used. Some approaches try to reduce the execution time of this merging phase by using a hash table when accumulating partial results; however, then the overhead is displaced to the main computation loop, because an indirect access is needed to access the appropriate entry in the hash table. Some researchers like [YR00] have developed an algorithm to characterize at run-time the reduction access pattern and choose the most appropriate parallelizing mechanism. However, as the inspector-executor mechanism, their mechanism also incurs some extra overheads that can only be amortized if the reduction access pattern does not change across loop invocations.

Thus, since reduction operations are very important and account for a large fraction of the execution time of many scientific applications, in Chapter 4 we investigate on new architectural support to speed-up parallel execution of reduction operations, and to make them scalable. This support should significantly speed-up execution of irregular and adaptive reduction, overcoming the situations where the software transformations did not succeed. I order to use commodity processors, we will always avoid or minimize changes in the processor, and concentrate the changes in the network/directory coherence controller of the DSM multiprocessor.

1.1.3 State Buffering in Speculative Thread-Level Parallelization

Despite the advances in compiler technology [BDE+96, HAA+96, SSC+99], there are still many codes that cannot be successfully parallelized. These are codes that have complex access patterns, like codes with subscripts of subscripts array accesses, pointers, complex

function calls, or input-dependent access patterns. In these cases, the compiler cannot safely parallelize these code sections, and they have to be executed serially. However, if these codes could be executed in parallel a significant degree of parallelism could be extracted.

To solve this problem, some work propose software solutions like the inspector-executor mechanism [CTY94, RP94, ZY87]. In this approach an inspector loop executes first. This loop gathers the data dependence structure of the code. This information will be used by the executor loop to execute the loop with explicit synchronizations to guarantee correct execution. Other software solutions [GN98, RP99] assume that there are not dependences and speculatively execute the code in parallel at the same time that collect the data accessed at run time. When the code ends, there is a test to determine if the parallel execution was correct or incorrect. If it was incorrect, the code must execute again in parallel. While all these techniques are promising they all have some drawbacks that limit their scalability.

Hardware based approaches assume that there are not dependences and let the compiler to extract tasks and execute them in parallel [CMT00, FF01, GVSS98, HWO98, Kni86, KT99, MGT98, OWP+01, PGRT01, SBV95, SCZM00, THA+99, ZRT99]. These schemes extend the invalidation-based cache coherence of the processor and check for dependences at run-time. In case of a violation, a combination of hardware and software solves it. This usually requires the squash of the tasks, the repair of the corrupted memory state and the re-start of the tasks in parallel. These mechanisms have been proposed at the chip-level multiprocessor [GVSS98, HWO98, KT99, MGT98, OWP+01, SBV95, THA+99] or at more scalable levels like [CMT00, FF01, GN98, PGRT01, RP95, RS00, SCZM00, ZRT99].

In Chapter 5, we will focus on the study of mechanisms to buffer the speculative state during speculative thread-level parallelization. Different approaches handle this issue differently. Some times this speculative state is buffered in the write-buffers of the processor [HWO98, THA+99], caches [CMT00, FF01, GVSS98, KT99, OWP+01, SCZM00], or special buffers [FS96, PGRT01] to avoid corrupting main memory. Other proposals take a different approach and generate a log of updates that allows them to backtrack execution [FLA01, Zha99, ZRT99]. Often, there are differences in the way caches, buffers and logs are used in the different schemes. However, there is no study that breaks down the design space and identifies the major design decisions and tradeoffs, and provides a performance and complexity comparison of important design points. We feel that such a study is needed, specially given the high performance that stakes of choosing a particular buffering scheme.

In addition, in this context of speculative thread-level parallelization, we also explore

new buffering mechanisms. In particular, we are interested on the problem of keeping multiple versions of the same variable. This problem appears when several tasks run in a processor and remain speculative. In this case the processor has to buffer the state that all these tasks produced. If these tasks wrote to different variables, then the processor must keep different versions of the same variable. All the current proposals that handle this problem require non-negligible modifications and additional hardware. Thus, in Chapter 6 we explore software algorithms to implement an efficient multi-version buffering scheme. We evaluate the overheads that the software introduced, and which are the performance benefits that this mechanism can deliver.

1.2 Thesis Contributions

This work makes several contributions in the context of shared memory multiprocessors. The first contribution is the characterization of the data access patterns in parallel programs. We perform a systematic characterization of the applications , and we study for first time the change in the program behaviour (pattern distribution and sequence length) as the number of processors increases. Also, in the context of multiprocessors, we propose a new performance model that facilitates the analysis of the demand and prefetch miss-rations, and traffic between caches, and with main memory. We evaluate several hardware prefetch mechanisms for a bus-based multiprocessor system, and we propose a new one. Our proposed mechanism combine a Load Cache with an on-miss insertion policy plus a sequential mechanism. Since bus-based multiprocessors have a limited bandwidth, we propose that the sequential prefetcher can be connected or disconnected. This decision can be taken by either the programmer or the compiler depending on the number of processors executing the application, and the behaviour of the application.

The second contribution of this thesis is the design and evaluation of architectural support for parallel reductions. The idea is to use the caches of the processors as temporary storage where processors accumulate their partial results. As cache lines are displaced, their values are combined with the value in the shared memory location. The required architectural changes are mostly confined to the directory controllers. Our design includes naive hardware modifications to detect reduction accesses, as well as a more advance scheme where the use of extra shadow addresses and the addition of operating system support avoids modifications of the processor. It also identifies the atomicity problems that appears, discuss the main tradeoffs of the different types of solutions, and propose two solutions. Finally we present a scenario of dynamic last value assignment where this architectural support can also be used. The proposed support speeds-ups significantly parallel reductions and makes them scalable when executing in shared-memory multiprocessors.

The third contribution of the thesis is related with the buffering of state in speculative thread-level parallelization. In this context, this thesis presents a novel taxonomy of the different approaches to handle speculative state. Our taxonomy includes a novel application of the concepts of architectural and future state to the memory state. It also classifies the approaches based on the support for multiple tasks and versions, and the main memory update policy. We perform a detailed tradeoff analysis and evaluate all the different approaches under a single architectural framework. We characterize a set of applications that are candidate for speculative thread-level parallelization. Our key insights are useful to understand the main bottlenecks of speculative thread-level systems based on the application characteristics.

Finally, for a particular type of approaches, we propose an effective software scheme to buffer multi-version speculative state. For that, we take a speculative parallelization protocol, and build all the software implementation of the buffering scheme on top of it. We evaluate the performance of our proposed only-software scheme, and compare it to a similar only-hardware one. Our study also includes a detailed evaluation of the major issues of our software proposal like the filtering of first stores.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 performs a characterization of important data access patterns that appear in parallel applications, and show several interesting metrics which can be applied to evaluate the potential of hardware data prefetch. In Chapter 3 we design and evaluate a low-cost Hardware Prefetch Mechanism for Bus-based Multiprocessors. Chapter 4 shows an architectural support added to a Scalable Shared Memory Multiprocessors to speed-up parallel reduction operations and make them scalable. Chapter 5 presents tradeoffs in the Buffering of Speculative State in Speculative Thread-Level Speculation. Chapter 6 presents a software scheme to buffer state from multiple tasks and versions, and finally Chapter 7 summarizes and presents future work.

References

[BC02] G. Bell and J. Cray. What's Next in High-Performance Computing? *Communications of the ACM*, 45(2):91–95, February 2002.

[BDE+96] 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. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[Bel85] C. G. Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–467, April 1985.

[CB94] T.F. Chen and J.L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture (ISCA'94)*, pages 223–232, April 1994.

[CC98] C.H. Chi and C.M. Cheung. Hardware Prefetching for Pointer Data References. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 377–384, 1998.

[CMT00] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 13–24, June 2000.

[CTY94] D. K. Chen, J. Torrellas, and P. C. Yew. An Efficient Algorithm for the Run-Time Parallelization of Do-Across Loops. In *In Proc. of Supercomputing '1994*, pages 518–527, November 1994.

[DDS93] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 Int'l Conference on Parallel Processing (ICPP'93)*, volume 1, pages 56–63, August 1993.

[DS95] F. Dahlgren and P. Stenström. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of the 1st Int'l Symposium on High-Performance Computer Architecture (HPCA'95)*, pages 68–77, January 1995.

[DS96] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[DS98] F. Dahlgren and P. Stenström. Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 47(18):385–398, October 1998.

[DUSH94] R. Das, M. Uysal, J. Saltz, and Y.S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[FF01] R. Figueiredo and J. Fortes. Hardware Support for Extracting Coarse-grain Speculative Parallelism in Distributed Shared-memory Multiprocesors. In *Proc. of the Int'l Conference on Parallel Processing (ICPP'01)*, September 2001.

[FLA01] M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Technical Report MIT/LCS Technical Memo 619, July 2001.

[FP92] J.W.C. Fu and J.H. Patel. Stride Directed Prefetching in Scalar Processors. In *Proc. of the 25th MICRO*, pages 102–110, 1992.

[FS96] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Trans. on Computers*, 45(5):552–571, May 1996.

[GN98] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proc. of Supercomputing '1998*, November 1998.

[GVSS98] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 195–205, February 1998.

[HAA+96] 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.

[HH98] C. Tseng H. Han. Improving Compiler and Run-Time Support for Adaptive Irregular Codes. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[HWO98] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *In Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 58–69, October 1998.

[IVBG98] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 369–376, 1998.

[JG99] D. Joseph and D. Grundwald. Prefetching using Markov Predictors. *IEEE Trans. on Computers*, 48(2):121–133, February 1999.

[KCPT95] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, pages 1250–1264, December 1996. A shorter version appeared in Proc. of the 9th Int'l Conference on Supercomputing (ICS'95), pages 255-264, July 1995.

[Kni86] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[Kru86] C. Kruskal. Efficient Parallel Algorithms for Graph Problems. In *Proc. of the 1986 Int'l Conference on Parallel Processing (ICPP'86)*, pages 869–876, August 1986.

[KT99] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, pages 866–880, September 1999.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[LP98] Y. Lin and D. Padua. On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In *Proc. of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, 1998.

[MG91] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[MGT98] P. Marcuello, A. González, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 77–84, July 1998.

[MH96] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proc. of the 1996 Int'l Conference on Supercomputing (ICS'96)*, pages 133–143, 1996.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. of the 1995 Int'l Conference on Parallel Processing (ICPP'95)*, pages 1–10, August 1995.

[OWP+01] C.L. Ooi, S. Wook, K.I. Park, R. Eigenmann, B. Falsafi, and T.N. Vijaykumar. Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor. In *Proc. of the 2001 Int'l Conference on Supercomputing (ICS'01)*, June 2001.

[PGRT01] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proc. of the 28th Annual Int'l Symposium on Computer Architecture (ISCA'01)*, pages 204–215, July 2001.

[Pot97] W.M. Pottenger. Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs. Technical Report, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Developement, May 1997.

[RIVL00] L. Ramos, P. Ibáñez, V. Viñals, and J.M. Llabería. Modeling Load Address Behaviour Through Recurrences. In *Proc. of the 2000 Int'l Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pages 101–108, April 2000.

[RMS98] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 115–126, October 1998.

[RP94] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. of the 1994 Int'l Conference on Supercomputing (ICS'94)*, pages 33–43, July 1994.

[RP95] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. of the SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 218–232, June 1995.

[RP99] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.

[RS00] P. Rundberg and P. Stenstrom. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[SBV95] G. S. Sohi, S. Breach, and S. Vajapeyam. Multiscalar Processors. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 414–425, June 1995.

[SCZM00] J.G. Steffan, C.B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 1–12, June 2000.

[SJG92] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 80–91, 1992.

[Smi78] Alan Jay Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[SSC+99] H. Saito, N. Stavrakos, S. Carrol, C. Polychronopoulos, and A. Nicolau. The Design of the PROMIS Compiler. In *Proc. of the Int'l Conference on Compiler Construction*, pages 562–573, March 1999.

[TE93] D.M. Tullsen and S.J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 278–288, may 1993.

[TE95] D.M. Tullsen and S.J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Trans. on Computer Systems*, 13(1):57–58, February 1995.

[THA+99] J.Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.C.Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):881–902, September 1999.

[YR00] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization. In *Proc. of the 2000 Int'l Conference on Supercomputing (ICS'00)*, May 2000.

[Zha99] Y. Zhang. Hardware for Speculative Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, May 1999.

[Zim91] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

[ZRT99] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 135–139, January 1999.

[ZT95] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 188–199, June 22–24, 1995.

[ZY87] C.Q. Zhu and P.C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. In *IEEE Trans. on Software Engineering*, pages 726–739, June 1987.

Chapter 2

Pattern Characterization

Data prefetching has been widely studied as a technique to hide memory access latency in multiprocessors. However, the effectiveness of a prefetch mechanism depends on the regularity in the data access pattern of the program. Characterizing the patterns that appear in the applications is very important, because this information can be very helpful to guide the research of new prefetch mechanisms, and to highlight how a prefetcher should behave for a particular application.

In this chapter we analyze the patterns that appear in a subset of the SPLASH-2 applications. We evaluate the percentage of loads that follow a specific pattern (scalar, sequential, stride, linked list and index list), and the number of times in a row that a load follows a particular pattern (sequence length). We also study the persistence of the patterns and the sequence length as the number of processors executing the application changes from 1 to 32. Our results reveal the dominance of sequential traversals and, to a lesser extent, the presence of stride accesses. We also find that for most of the evaluated applications, with the only exception of Cholesky and Radix, the pattern distribution remains constant independently of the number of processors executing the application. Also the sequence length scarcely changes with the number of processors.

2.1 Introduction

As the processor speed continues to increase, the performance of many applications is limited by the high latency to access memory. If we consider multiprocessors, this latency to access memory is even higher. In a bus-based multiprocessor many processors have to compete for a shared bus, which increases the latency to access memory. In a more scalable multiprocessor where processors are connected through a general network, remote accesses to either main memory or a second processor's cache can take hundreds (300 -

800) of cycles. To cope with this problem a solution that many researchers have explored is data prefetch.

Data prefetching has been proposed to hide read latencies in multiprocessors and uniprocessors. Its effectiveness depends on the regularity in the access pattern of the program, and can be done either in software or in hardware. Software approaches can perform well whenever the programmer or the compiler can provide or extract information about the data access pattern of an application [KCPT95, MG91, TE93]. However, they add extra instructions to the code, and their automatic application may not be easy. Hardware approaches add additional hardware to detect the data access patterns. Based on the detected patterns, they are able to predict the next address that the processor will issue in the near future (e.g [DS95, DS96, DS98]). In the next chapter, we focus on hardware prefetching mechanisms. In this chapter we characterize the patterns that appear in parallel programs. We only study load access patterns, since write latencies can be easily hidden by the processor. We analyze the percentage of loads that follow one of these five patterns: scalar, sequential, stride, linked list or index list traversal when the traversal of the list is done with a single load, as described in [IVBG98, MH96].

We also study another metric. Hardware prefetch is hardly useful if once a pattern is detected, the pattern does not appear again in the program. For that reason, we have studied the sequence length. The sequence length is the number of times in a row that a load follows a particular pattern. Hardware prefetchers will perform poorly if this number if low. This is due to the learning time they need to get confidence. Furthermore, access patterns or sequence lengths may change when the number of processors executing the application changes. Previous works do not evaluate how this variation on the number of processors executing the application may affect the distribution of the memory access patterns. The length of memory accesses following a given pattern has been measured in [DS96], but little systematic work on pattern characterization has taken into account the number of processors.

In this chapter we use a subset of applications and kernels belonging to the SPLASH-2 suite [WOT+95]. We study the data access patterns and the sequence lengths that appear in these applications when the number of processors varies from 1 to 32. Our results reveal the dominance of sequential traversals and, to a lesser extent, the presence of stride accesses. Single load linked list or index list traversal almost does not appear in these applications. We observe small differences in the access pattern distribution or sequence length when the number of processors changes. But for Radix and Cholesky, which are the applications with a higher communication-to-computation ratio, there is a significative change on the access patterns as the number of processors increases. In the

other cases, the sequence length changes relatively little with the number of processors.

This chapter is organized as follows: Section 2.2 presents the patterns measured, the applications evaluated and the environmental setup; Section 2.3 shows the patterns distribution; Section 2.4 studies the sequence length for the different access patterns; Finally Section 2.5 summarizes.

2.2 Measured Patterns and Environmental Setup

The prefetching techniques we use in the next chapter strongly rely on the existence of predictable memory access patterns. As far as we know, there is no systematic study of such patterns in parallel applications as the number of processors varies. Therefore, we have analyzed load accesses for a subset of the SPLASH-2 suite [WOT+95], using sequential and non-sequential pattern detectors. We have also analyzed the sequence length of loads. Section 2.2.1 presents these two measured patterns, while Section 2.2.2 presents the experimental setup and the applications we use for our experiments.

2.2.1 Measured Patterns

Access Patterns

We have looked for five load patterns which can potentially be recognized by the prefetching techniques we use later. These patterns, which we explain next, have been recognized only when they appear in the stream of addresses issued by a single load.

- *Stride (STR)* This pattern appears when the distance between the objects accessed is regular and larger than the block size. An example of this pattern appears when traversing the column of a matrix that has been stored in memory by rows.

  To detect this pattern we keep track of the addresses issued by each single load during consecutive executions, and we compute the stride S as the difference between these addresses. Thus, we can define:

  $$S_i = A_i - A_{i-1},$$

  where A_i and A_{i-1} are the addresses issued by a load during its i-th and (i-1)-th execution. A pattern is detected when the stride matches the new stride computed the following times that the load executes:

  $$S_i = S_{i-1} = S_{i-2} = ... = S_{i-k},$$

  where k is a confidence counter. Once a pattern is detected, the adddres to prefetch can be computed as:

$$A_{i+1} = A_i + S_i$$

We have also considered that a load is accessing memory with a stride pattern when the stride S is negative or larger than the the block size.

- *Scalar (SCA)* This pattern appears when accessing the same address. For instance, when accessing the induction variable in a loop. For hardware prefetch, the detection of this pattern is hardly useful. After the first cache miss the data will be usually in the cache, unless it has been displaced because of cache conflicts or capacity problems. This pattern is a particular case of the stride pattern that is detected when the computed stride S is equal to zero.

- *Sequential (SEQ)* This pattern appears when accessing objects that are contiguously placed in memory. For instance, stack accesses or accesses to consecutive elements in an array follow this pattern.

  This pattern can be considered a particular case of the stride pattern, detected when the stride S is larger than zero and smaller or equal to the block size.

- *Linked Data Structure (PTR)* This pattern appears when traversing a list of records chained by pointers, where one of the fields of the record is a *next* pointer to the start address of the next record (e.g. p=p->next).

  For the detection of this pattern the value read by the load needs to be tracked. If we call d_i to the value read by a load during its i-th execution, and A_i to the address issued also during it i-th execution, we can define $Desp_i$ as:

  $$Desp_i = A_i - d_{i-1}$$

  Thus, the linked list pattern is detected when

  $$Desp_i = Desp_{i-1} = Desp_{i-2} = ... = Desp_{i-k}$$

  where k is also a confidence counter. The address to prefetch is computed as:

  $$A_i = d_i + Desp_i$$

- *Index List (IND)* This pattern appears in numeric codes and it is a mechanism used by some methods to store vectors or sparse matrix. This pattern is similar to the pointer list, but in this case we have an index vector that stores the index to the next non-zero element of the index vector (e.g. i=index[i], instead of a pointer to the next record).

  Figure 2.1 shows the data of a sparse vector. In Figure 2.2, the same vector has been stored using the LL (Linked List) format. With this format, only the non-zero elements of the vector in Figure 2.1 need to be stored in the DATA vector. The COLUMN vector keeps the column where each element of the DATA vector was

placed in the initial vector. The in-order traversal of this vector requires the use of the INDEX vector. The list starts with the element 0. The index to the next non-zero element is in INDEX[0], and is 2. The next index to the next non-zero element is in INDEX[2], and so on.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
DATA	0	0	0	71	84	0	86	27	0	0	48	0	0	0	66

Figure 2.1. Sparse vector.

	0	1	2	3	4	5
COLUMN	3	6	4	10	14	7
DATA	71	86	84	48	66	27
INDEX	2	5	1	4	−1	3

Figure 2.2. Sparse vector stored using the LL format.

To detect this pattern we need to compute the initial address of the INDEX vector. We call IV to this address, and $Isize$ to the size of each index (integer) in bytes. We can define IV_i as:

$$IV_i = A_i - d_{i-1} * Isize$$

Thus, this pattern is detected when

$$IV_i = IV_{i-1} = IV_{i-2} = ... = IV_{i-k}$$

and the address to prefetch is computed as:

$$A_{i+1} = IV_i + d_i * Isize$$

The mechanism to detect sequential and stride pattern as described here was first proposed in [BC91]. The mechanism to detect linked list and index list traversal was proposed in [MH96].

Table 2.1 summarizes the recurrence conditions explained above for each particular pattern. Remember that we track the stream of addresses issued by a single load. *Ai* in Table 2.1 is the address generated by a load during its *i-th* execution, *Di* is the value read during its *i-th* execution, and *Bsize* is the block size [GBIV01]. The first three patterns (SCA, SEQ and STR) are particular cases of regular traversing, characterized by address-address recurrences, whereas the next two (PTR, IND) are value-address recurrences that

we classify as Irregular Traversing. When an instance of a load matches several patterns at a time, it is first classified according to the pattern recognized in the previous instance. If a load follows several patterns at the same time we classify it according to the following priorities: SCA, PTR, IND, STR and SEQ.

Pattern		Acro.	Recurrence	Parameter
Regular Traversing	Scalar	SCA	$A_i = A_{i-1}$	
	Sequential	SEQ	$A_i = A_{i-1}$ + s	0 < s <= Bsize
	Stride	STR	$A_i = A_{i-1}$ + S	(S > Bsize) || (S < 0)
Irregular Traversing	Pointer List	PTR	$A_i = D_{i-1}$ + d	record displacement = d
	Index List	IND	$A_i = 4*D_{i-1}$ + K	index integer size = 4 base address of the index array = k
Not Recognized		NR	Not Recognized as any of the previous ones	

Table 2.1. The load patterns we consider and the recurrences defining them. A_i and D_i refer respectively to the address and value of the i-th instance of a given load instruction.

Sequence length

We have also measured sequence lengths. *Sequence length* is defined as the number of consecutive executions of a load following the same pattern. As an example, the address stream

..., 10, 4, **68, 132, 196,** 100, ...

contains a sequence of length 3, since the 3 bolded addresses differ from the previous ones in the constant stride 64.

Sequence lengths are important because hardware prefetching mechanisms will be useful only if they are long. Hardware prefetching detects that a particular load is accessing memory following a certain pattern by recording the address issued by the load during a minimum number of consecutive times. This is called *learning time* and is used to get confidence. If the load does not execute again once the pattern it follows has been detected, or if it only executes a few times, then the detected pattern will be hardly useful.

In our characterization the minimum length we detect is 2. Thus, the hardware prefetchers we evaluate in the next chapter will need a load to execute 3 times before the first prediction can be issued. If we look at the previous address stream, this means that our hardware prefetcher will use addresses 4 and 68 to compute the stride 64; address 132 will confirm that the load is accessing memory with a stride of 64; then, address 196 will be issued to prefetch.

2.2.2 Workload and Methodology

Table 2.2 shows the selected SPLASH-2 applications and some metrics for 16 processors. The applications have been targeted to a MIPS-2 architecture and run until completion by using MINT, an execution driven simulator for multiprocessors [VF94a]. Throughout the chapter the analysis is constrained to the parallel section of the programs.

SPLASH-2 programs have been classified into three classes according to the interaction between their data structures and access patterns and the cache lines. We refer to these classes as: a) *Regular programs*, based on contiguously allocated data structures (Ocean, FFT, LU, Radix and Cholesky), b) *Particle programs*, based on particles, i.e. structures that can share the same cache line, and that are accessed by different processors (FMM, Barnes); and c) *Irregular programs*, with highly irregular data structures (Radiosity, Raytrace). The implementations of Ocean and LU used in [VF94b] corresponds to those specified here as Non Contiguous: main data sets are 2-dimensional arrays where sub-blocks are not contiguous in memory. Alternatively, Ocean and LU Contiguous implementations allocate data as arrays of sub-blocks, in order to reduce or even eliminate false sharing. Moreover, in the case of Ocean contiguous, a multi-grid (instead of a SOR) solver is applied.

	Program	Parameters	Inst. (M)	Reads (M)	Writes (M)
regular	Cholesky	tk15.O	584.4	201.8	27.7
	FFT	64K points	32.9	8.1	5.8
	LU	512x512 matrix, 16x16 blocks	340.6	97.6	47.8
	LU non-	512x512 matrix, 16x16 blocks	340.9	97.5	47.8
	Ocean	258x258, tolerance 10-7, steps 4	282.3	81.6	18.5
	Ocean-non	258x258, tolerance 10-7, steps 4	477.4	101.8	17.8
	Radix	1024 K keys, radix =1024	47.9	11.2	6.6
particle	Barnes	16K particles	2569.0	861.0	575.1
	FMM	16K particles	1035.7	232.8	38.7
irregular	Radiosity	room -ae 5000.0 -en 0.050 -bf 0.10	2878.5	574.8	305.8
	Raytrace	car	994.0	228.6	117.6

Table 2.2. Selected subset of SPLASH-2. Statistics are for 16 processors, compiled with cc -O2 -mips2 -non_shared (MIPS SGI v.7.1 compiler). Ints refers to the total number of executed instructions across all processors. We also show the total number of Reads and Writes.

2.3 Pattern Distribution

Figure 2.3 shows the pattern distribution of all load addresses of the evaluated programs running on a 16-processor system. This experiment has been carried out placing a pattern detector in each simulated processor and then summing up the patterns detected for each

one. The bars in the Figure 2.3 include all sequences of length greater or equal than 2.

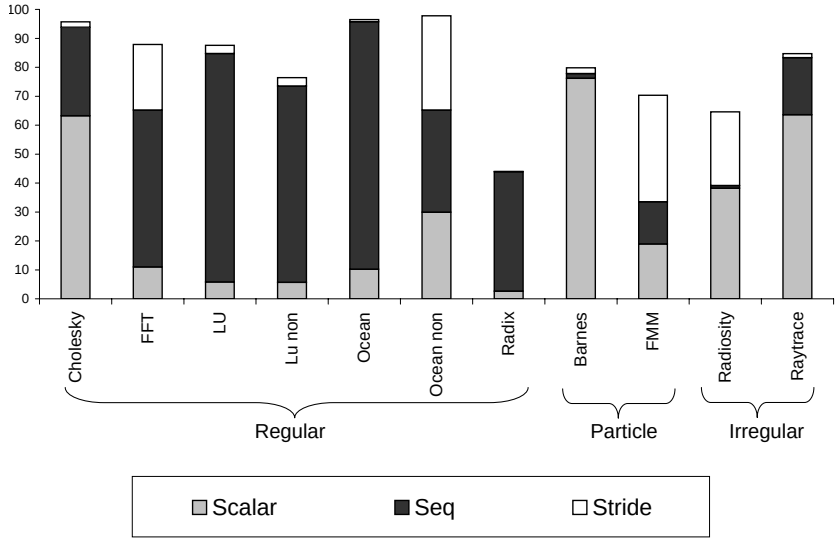

Figure 2.3. Breakdown of pattern distribution for 16 processors and Bsize = 32. Pointer and Index patterns are not plotted due to their negligible contribution.

Figure 2.3 does not show the PTR or IND patterns. The reason is the practical absence of these patterns in the whole set of programs evaluated, with measured percentages always falling under 0.1%. This observation agree with previous characterizations available for uniprocessor applications. For example, only Spice of Spec92 and health of Olden show significant percentages of self-linked patterns (21% of self-index and 27% of self-pointer, respectively), being these patterns irrelevant in the whole set of Spec95fp, Perfect and NAS-serial suites and of little importance in Spec95int and OLDEN (under 3% and 6%, respectively in average [RIVL00]). Consequently, when experimenting with hardware-based prefetchers in the next chapter , we shall not include PTR and IND pattern detectors.

To a large extent, the results mirror the features of the application data structures. Thus, *regular programs* have a large fraction of SEQ pattern and -excluding Radix- a low fraction of NR accesses (9.7% in average). In contrast SEQ is very uncommon in the other two classes of programs. Moreover, contiguous implementations of LU and Ocean prove to increase the percentage of SEQ pattern: in LU this increase adds regularity (the NR group decreases); in Ocean STR disappears and SEQ increases but the overall regularity remains unchanged.

Barnes and FMM are *particle programs* and most structures are linked lists. Nevertheless, a significant amount of the STR, SEQ and SCA patterns appear in FMM, because

of the loops in its code that access 1-dimensional global arrays sequentially, and a 2-dimensional global array following the STR pattern. Moreover, the compiler unrolls loops, converting many sequential accesses to arrays into STR accesses. In Barnes, integer and float global variables are frequently loaded inside a recursive function that performs the core computations, contributing in a remarkable way (76.2%) to the SCA pattern. Finally, NR accesses are significant in particle and irregular programs (25% in average), suggesting the existence of more complex patterns which presumably will not be exploitable from simple hardware-based prefetchers.

Besides 16-processors runs, we have performed simulations for 1-, 2-, 4-, 8-, and 32-processors. We have found that for most applications the sum of the loads executed by all processors remains constant or increases only slightly with the number of processors. In addition, the weights of the patterns observed do not change when we vary the number of processors. The only exceptions are Cholesky and Radix. In these applications, the sum of the loads increases with the number of processors (Figure 2.4). In Cholesky it can be observed a slight transfer from SEQ to STR (ending with a 2% stride for 32 proc.) and a remarkable increase in the number of scalar loads (a factor 20x from 1 to 32 proc.). When adding processors in Radix, the additional loads split equal between SCA and SEC patterns.

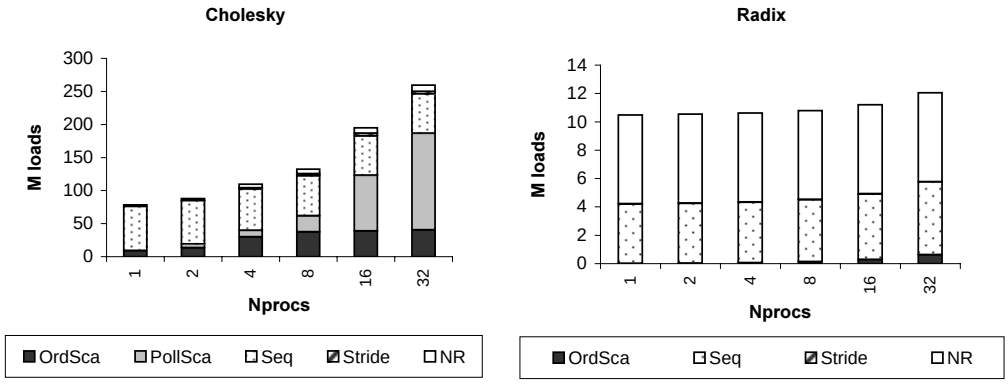

Figure 2.4. Pattern Distribution (million loads) for Cholesky and Radix up to 32 processors. Scalar is split in Ordinary loads (regular and synchronizing) and Polling loads.

These two applications, Cholesky and Radix, have a high communication-to-computation ratio, because each processor communicates some locally-computed data to the rest of the processors:

- In **Cholesky**, in each iteration of the outermost loop, a processor computes the pivot element of the diagonal and forwards it to the rest of processors. It has

a structure and partitioning similar to the LU factorization kernel, but it is not globally synchronized between steps, and the consumer processors wait for the pivot spinning on two scalar variables. Consequently, processors do not synchronize using synchronization variables only: they also synchronize by polling on regular variables. In our simulations we are able to identify the synchronization variables, and any spinning on them appears as a single load in our bars. However, all the loads caused by the polling on regular variables are counted, and appear in the bars as part of the Scalar pattern. To separate these polling loads from ordinary (regular and synchronizing) loads, we break down the Scalar category into Poll-scalar and Ord-scalar, respectively. As can be seen from Figure 2.4 the effect of such polling loads heavily affects the pattern distribution as Nprocs increase. Even though SEQ percentage decreases sharply (from 85.2% for 1 processor to 22.8% for 32), note that the increasing Poll-scalar accesses come from only two scalar loads, so the influence of this scalar stream over a LC-based prefetcher should be negligible.

- The goal of **Radix** algorithm is to sort N integers or keys of B bits using a radix R. The algorithm proceeds in ((B/R) + 1) phases or iterations, and each iteration has several steps. In each iteration, a processor has to sort N/Nprocs keys. In the first step each processor uses a local data structure of R entries to compute an histogram. This step is the main responsible of the high NR pattern, since the access to the entries does not follow any pattern. Next, it traverses this local structure to accumulate the values in order to form the local accumulative histogram. In the second step, the local histograms are accumulated again into a global histogram. The parallel algorithm requires $\log_2$Nprocs stages to compute a global histogram from the local ones. The amount of work in this second step depends on the size of the radix (which is the size of the histograms) and on the number of processors. Finally, the global histograms are used to copy the keys to the sorted position in the output array. Thus, when increasing the processor count, the instructions devoted to do the histogram reduction (which follow either the SCA or SEQ pattern) also increase: a 14.9% load increase from 1 to 32 processors.

Therefore, the relative weight of patterns can vary in applications where a significant overhead is added as the processor count grows (communication and synchronization). In turn this can lead to applications that benefits from a prefetch predictor when executing in a few processors, but which may not benefit when executing with more processors. We feel this consideration is important, since applications obtained from sequential algorithms by automatic parallelism extraction easily could fall in this class of poor-scalability applications.

2.4 Sequence Length

Figure 2.5 shows how the captured pattern distribution changes if we remove the first three loads of each sequence that need to be executed before an address can be predicted (Section 2.2.1). The left bar of each program shows the load pattern distribution when all sequences of length greater or equal than 2 have been included (as in Figure 2.3), while the right bar shows the pattern distribution if we remove the first three loads of each sequence. The overall regularity (SCA+SEQ+STR) decreases for all programs (15.1% loss in average), being noticeable the 45.9% loss in capturing the SEQ pattern of LU-non.

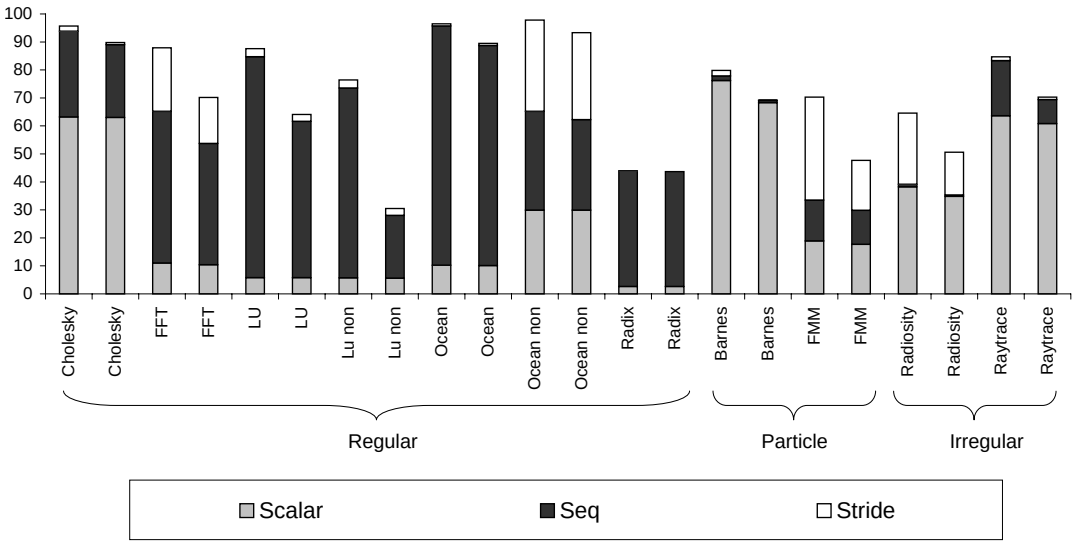

Figure 2.5. Effect of removing the first three loads of each sequence. The left bar shows the pattern distribution when all sequences of length greater or equal than 2 have been included. In the right bar, the first three loads of each sequence are not included.

A finer analysis appears in Table 2.3, where the dominant sequence lengths are recorded for 16 processors, along with the percentage of loads that follow them. We have lumped the more than 7 together. The SEQ pattern has been split up into 4-, 8-, 16- and 32-byte sequences, as if they were different patterns. Sequences with percentages below 2.0 are excluded from the main columns and accumulated in the remaining (REM) column. The length with the maximum percentage is featured in bold.

It can be observed that usually there are a just a few outstanding sequence lengths per application. Moreover, the SCA pattern is nearly always executed in long bursts (>7), and the same holds true for SEQ-4, -8 and -16. By contrast, STR and in a lesser extent SEQ-32 patterns, use to execute in shorter bursts (e.g. length 4 (34.3% STR) in FMM and length 3 (67% SEQ-32) in LU-non). As a rule SEQ patterns perform longer sequences than

	SCAL.	STRIDE	SEQUENTIAL				REM	NR
			4	8	16	32		
Cholesky	**>7 (62.9%)**	-	-	>7 (17.6%)	>7 (7.7%)	-	7.6%	4.3%
FFT	>7 (10.5%)	3 (7.6 %) >7 (14.2%)				2 (5.0%) 6 (7.4%) **>7 (39.9%)**	3.3%	12.1%
LU	>7 (5.8%)	>7 (2.9%)				3(29.5%) 7 (4.7%) **>7 (44.0%)**	0.8%	12.4%
LU Non	7 (5.7%)	>7 (2.9%)	-	-	-	**3(67.1%)**	0.7%	23.6%
Ocean	>7 (10.1%)	-	-	**>7 (31.1%)**	>7 (25.2%)	>7 (28.0%)	2.1%	3.5%
Ocean Non	>7 (29.9%)	**>7 (32.6%)**	-	>7 (4.9%)	>7 (21.3%)	>7 (8.8%)	0.3%	2.3%
Radix	>7 (2.6%)	-	>7 (3.8%)	-	**>7 (37.4%)**	-	0.2%	56.0%
Barnes	6 (2.4%) 7 (9.5%) **>7 (57.5%)**	-	-	-	-	-	10.3%	20.2%
FMM	>7 (17.7%)	**4 (34.3%)**	>7 (3.9%)	-	>7 (4.1%)	>7 (5.0%)	5.3%	29.6%
Radiosity	**>7 (33.2%)**	2 (3.4%) 3 (2.7%) 5 (2.8%) >7 (12.5%)	-	-	-	-	10.0%	35.4%
Raytrace	**>7 (61.2%)**	-	>7 (3.4%)	-	3 (11.8%)	-	8.2%	15.3%

Table 2.3. Dominant sequence lengths in all patterns for 16 processors. Maximum values for each application are shown in bold. Percentages below 2.0 are omitted and accumulated in the REM column.

the STR ones and 32B strides usually dominate over shorter ones. Note that sequential tagged prefetching could yield good results here, because it can capture SEQ patterns coming from small-length sequences which will not be fully profited by another hardware predictor that has larger learning times.

We now consider varying the number of processors. Depending on the problem partition sequence lengths can decrease as the number of processors executing an application increases. Figure 2.6 shows an example. The arrows on top show the traversal of the data structure in the sequential code. The Figure shows two different ways to divide the work among the processors executing the parallel application. The problem partition in Figure 2.6-(a) causes a reduction in the sequence length, while the problem partition in Figure 2.6-(b) keeps it constant. Furthermore, if the problem partition applied is the one in Figure 2.6-(a), the sequence length decreases as the number of processors increases.

We also notice that as the processors increase, the number of loads executed by each processor decreases. This is true even for the communication-intensive applications Cholesky and Radix. Thus, the problem partition when a program is parallelized may cause a reduction in the sequence lengths as the number of processors increases. This would decrease the effectiveness of the hardware-prefetch mechanisms. However, we have done simulations with 1, 4, 8, 16, and 32 processors and we have observed that the sequence

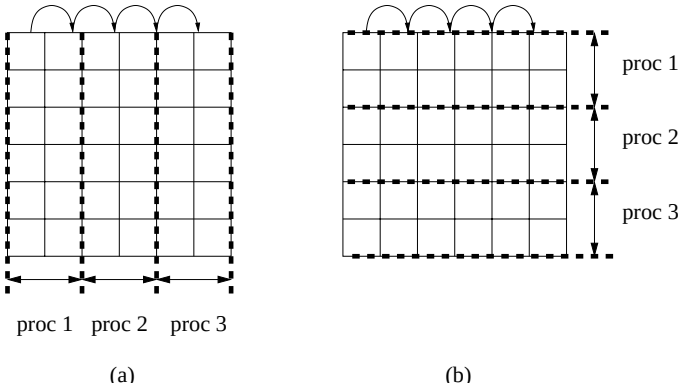

Figure 2.6. Changes in sequence length depending on the problem partition and number of processors executing the application.

length changes relatively little with the number of processors. We believe that SPLASH-2 applications are very well programmed and tuned, but changes in the sequence length will appear in applications not so well programmed, or applications automatically parallelized by the compiler.

2.5 Summary

In this chapter we have used a subset of the SPLASH-2 suite as workload, and we have performed a characterization of the load memory access patterns and sequence lengths. This characterization has been done when the number of processors changes from 1 to 32.

The pattern breakdown reveals a practical absence of list and chained index patterns, a dominance of sequential traversals, and meaningful presence of stride accesses. The fraction of accesses non recognized is big enough (for some applications) to indicate that further research is needed on new cost-effective pattern recognizers. Although there were sparse comments in the literature, we have observed for the first time the persistence of the patterns when ranging from 1 to 32 processors, although with some exceptions in applications with a high communication-to-computation ratio (Cholesky and Radix). This implies that the dynamics of loads scarcely varies as the problem is spawned on more processors. A similar conclusion arises when studying the sequence lengths of memory accesses that follow a pattern: they concentrate on one or at most two values, and this also hold when varying the number of processors.

References

[BC91] J.L. Baer and T.F. Chen. An Effective on-chip Preloading Scheme to Reduce Data Access Penalty. In *Proc. of Supercomputing'91*, pages 176–186, 1991.

[DS95] F. Dahlgren and P. Stenström. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of the 1st Int'l Symposium on High-Performance Computer Architecture (HPCA'95)*, pages 68–77, January 1995.

[DS96] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[DS98] F. Dahlgren and P. Stenström. Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 47(18):385–398, October 1998.

[GBIV01] M.J. Garzarán, J. Briz, P. Ibáñez, and V. Viñals. Hardware Prefetching in Bus-based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In *Proc. of the 2001 Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 345–354, 2001.

[IVBG98] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 369–376, 1998.

[KCPT95] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, pages 1250–1264, December 1996. A shorter version appeared in Proc. of the 9th Int'l Conference on Supercomputing (ICS'95), pages 255-264, July 1995.

[MG91] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[MH96] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proc. of the 1996 Int'l Conference on Supercomputing (ICS'96)*, pages 133–143, 1996.

[RIVL00] L. Ramos, P. Ibáñez, V. Viñals, and J.M. Llabería. Modeling Load Address Behaviour Through Recurrences. In *Proc. of the 2000 Int'l Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pages 101–108, April 2000.

[TE93] D.M. Tullsen and S.J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 278–288, may 1993.

[VF94a] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[VF94b] J.E. Veenstra and R.J. Fowler. The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors. Technical Report number 490, Computer Science Departament. Univ. of Rochester. New York, 1994.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

Chapter 3

Hardware Prefetch in Bus-based Multiprocessors

In the previous chapter we have charaterized the load access patterns of a subset of SPLASH-2 applications. In this chapter, we will focus on hardware prefetch mechanisms. Most recent research on hardware prefetching focuses either on uniprocessors, or on distributed shared memory (DSM) and other non bus-based organizations. Little work has been done in the context of bus-based multiprocessors. In this context prefetching poses a number of problems related to the lack of scalability and limited bus bandwidth of these modest-sized machines.

This chapter considers how the number of processors and the memory access patterns in the program influence the relative performance of sequential and non-sequential hardware prefetching mechanisms in a bus-based multiprocessor. We compare the performance of four inexpensive hardware prefetching techniques, varying the number of processors. After a breakdown of the results based on a novel performance model, we propose a cost-effective hardware prefetching solution for implementing on such modest-sized multiprocessors.

3.1 Introduction

Applying data prefetching on modest-sized bus-based multiprocessors is challenging. These machines are based on a cheap, but fixed and non-scalable organization, and have to cope with the ever growing CPU and memory requirements of new applications. They need to balance processor speed, cache sizes, bandwidth and final cost carefully. In this market so sensitive to cost, yet so performance-driven, cost-effective performance optimizations have a great interest.

Data prefetching has been proposed to hide read latencies in multiprocessors and uniprocessors. Prefetch can be implemented in software or in hardware. Hardware approaches look for regular patterns into the stream of memory references issued by the processor, or into the stream of consecutive references issued individually by each memory instruction (e.g. [DS98]). Lately, proposals based on dependence analysis among memory instructions [CC98, RMS98] and on Markovian predictors [JG99] have been suggested for uniprocessors. Hybrid software/hardware schemes that decrease the instruction overhead and improve the accuracy of predictions have been proposed for multiprocessors [CB94, ZT95]. However, prefetching in shared memory multiprocessors is prone to increasing memory traffic and false sharing. These problems are particularly important in bus-based multiprocessors, due to their limited bandwidth [TE93, TE95]. Little work has been done in recent years to study the impact of prefetching on this kind of machines.

This chapter focuses on how the number of processors and the memory access patterns influence the performance of low-cost prefetching mechanisms in a bus-based multiprocessor. We compare four prefetchers coupled to the on-chip cache against a Base system with no prefetch. We vary the number of processors from 1 to 32. The evaluated system is based on a split-transaction bus, with two levels of cache (on- and off-chip) and with prefetching tied to the first level. A subset of applications and kernels belonging to the SPLASH-2 suite is used as the workload [WOT+95]. We found that the prefetcher based on the use of a Load Cache managed in a non-conventional way (only the loads missing in the data cache are inserted in it), appears to be a suitable prefetcher.

This chapter is organized as follows: Section 3.2 presents related work; Section 3.3 describes the environmental setup; Section 3.4 presents a novel performance model; Section 3.5 evaluates different hardware prefetching alternatives through a detailed simulation model, and proposes a new cost-effective hardware prefetch mechanism; Finally in Section 3.6 conclusions are summarized.

3.2 Related Work

Prefetching is implemented in *hardware* [BC91, CB94, CC98, DDS93, DS95, DS96, DS98, FP92, Iba, IVBG98, JG97, JG99, JT93, MH96, RMS98, Smi78, Smi82], when the hardware alone decides what data to prefetch and when and where to prefetch the data. It can also be *software* [CB94, KCPT95, MG91, TE95], when the hardware supports a prefetching instruction. In this case, either the programmer or the compiler must insert prefetching instructions in the code. Finally, it can be *hybrid* [CB94, ZT95], when a combination of both hardware and software prefetching is used. Next we review past proposals. We focus on hardware prefetching mechanisms, and on the evaluated system (uni-

or multiprocessor).

Sequential prefetching is one of the simplest hardware prefetch mechanisms that has been proposed. In this scheme, cache line *l+1* is considered for prefetching upon the reference of cache line *l*. Three different mechanisms, *always, on-miss*, and *tagged*, have been proposed [Smi78, Smi82]. With *always*, on each access to line *l*, line *l+1* is always prefetched. With *on-miss*, if the access to the line *l* was a miss, then line *l+1* is prefetched. With *tagged* prefetch, in the first access to line *l*, line *l+1* is prefetched. The implementation of tagged prefetching requires an extra bit (tag) per cache line. This bit is used to know if a cache line has been accessed by the processor while it was in the cache. This bit is reset when the cache line is loaded in the cache by a prefetch action, and it is set when the processor access it for first time. A prefetch of the next cache line is issued whenever the bit of a block is set.

In [DDS93], an *adaptive* sequential prefetching mechanism was proposed. This mechanism dynamically changes the number of consecutive cache lines to prefetch (degree of prefetching). This mechanism measures the effectiveness of the prefetch as the ratio between the number of prefetched lines that are used at least once and the number of prefetched lines. If the prefetch is effective, the degree increases; otherwise, it decreases. This mechanism can reduce traffic by removing useless prefetches.

Other proposals try to predict both sequential and non-sequential accesses. A *Load Cache* (LC) mechanism consists of a table relating the PC-address of a load instruction with its individual addressing behaviour. Whenever a load that is not already in the LC is executed, an LC miss arises and the instruction is inserted into the LC. Data addresses issued by that load in successive executions are tracked down, in an attempt to recognize the reference's stride. If a stride is detected, a new prefetch address is computed and issued when the load executes again. With the LC mechanism, sequential and stride patterns can be detected. The original proposal (*Reference Prediction Table*) was introduced in [BC91] and evaluated in [CB94] along with a mechanism based on a *lookahead PC* for issuing just-in-time prefetches. Later proposals elaborate more on the topic, either evaluating, simplifying the prefetch scheduling or adding recognizable patterns [DS96, FP92, JT93]. Note that both, the LC mechanism and [Smi82] can predict sequential accesses. However, the LC mechanism only predicts the sequential accesses issued by a single load, while the mechanism in [Smi82] can predict sequential accesses issued by different loads with spatial locality.

Mehrotra [MH96] also uses a LC table, but introduces the concept of value-address recurrence, enabling identification of the patterns that appear in the traversal of a linked list and/or index list, when the traversal is done using a single load. To recognize these

patterns the LC table must also track the value read by the load instruction [Iba, IVBG98], as explained in the previous chapter (Section 2.2.1).

With the LC approach a load that misses in the LC is always inserted. With this *always insertion* policy, the probability of finding a load in the LC table is proportional to the number of times that the load executes. However, we are interested on prefetching the addresses to be issued by those loads that will miss in the data cache. Therefore, we would prefer that only those missing loads were inserted into the LC table. For that, a LC with *on-miss insertion policy* (LCm) was proposed in [Iba, IVBG98]. Unlike the conventional LC management, LCm only inserts in the table those loads missing in the data cache, preventing loads with high-locality from polluting the LC table. On-miss insertion is shown to be two-fold beneficial: it avoids useless cache lookups and achieves a very good LC space efficiency, usually outperforming the conventional LC. For instance, a LCm with 8 entries can perform even better than a conventional LC with 512 entries. The same work pointed out that activating sequential tagged prefetching in parallel with LCm (called LCms) can achieve the best results in many cases, because the spatial locality coming from an interleaving of several load streams can not be captured by an LC-based prefetcher. This LCm mechanism was proposed in an uniprocessor environment, and its advantages have not been verified in a multiprocessor system.

Recent proposals can detect other more complex patterns. [RIVL00] and [RMS98] extend the work of Mehrotra to recognize the linked list traversals when the traversal of the list is done using a pair of loads. A different approach is taken in [JG97, JG99]. In these proposals, a list with the next misses (successors) is kept for each missed reference. When the processor issues a reference, it looks up for any successors of the referenced line. If there are, it issues the prefetch of one or several of these successor lines.

Sequential and stride prefetching have been extensively studied on scalable DSM [DS95, DS96, DS98, CB94] where processors and memory are connected through a multi-path interconnection network. Results in [DS95, DS96] showed that sequential prefetching can generally outperforms stride prefetching in a DSM environment, however it increases false sharing, and also memory traffic. This additional traffic may not hurt performance in a DSM environment, where memory contention is assumed to be low, but it may become a serious issue in bus-based multiprocessor where memory bandwidth is limited. There are two well-known works on bus-based shared memory multiprocessors [TE93, TE95]. However, these works use software prefetching and only consider a single level of cache. In addition, they do not evaluate the effectiveness of the prefetch as the number of processors increases.

We feel that an LCm is worth trying on a bus-based multiprocessor, because of several

reasons: it could press the cache and the memory far less than sequential prefetchers do, and its implementation cost is low. On the other hand, when adding tagged sequential prefetching, LCms could outperform conventional mechanisms, particularly for those applications requiring moderate bandwidth. Since bandwidth can become a limitation as the number of processors grows, we will consider its impact both on the addressing pattern distribution and the LCm/LCms performance.

3.3 Environmental Setup

The system model we have considered is outlined in Figure 3.1, detailed for a single processor. We simulate a system with 800 MHz RISC processors executing a MIPS-2 ISA where processors are connected through a split transaction bus. Due to the lengthy simulations that our experiments require, we simulate a single-issue processor with blocking read misses. Write latency is hidden by using a Coalescing Write Buffer (CWbuf) operating as described in [VF94b]. According to [PTA97], the performance of a system with a W-issue superscalar can be roughly approximated by simulating a single-issue processor with W times higher frequency. Consequently, our speed results will roughly approximate the speed of a system with 2-issue superscalar processors cycling at 400 MHz.

The bus has split transactions and is based on the DEC 7000 and 10000 AXP Family [AI92]. It runs at 200 MHz, with 32 bits for addresses and 128 bits for data, which yields 3.2 GBps of bandwidth. The pipelined bus protocol consists of four phases: 1) Request & arbitration (4 CPU cycles), 2) Address & command issue (4 CPU cycles), 3) Read bank & snoop (32 CPU cycles), 4) Transfer in chunks of 16B (4x4 = 16 CPU cycles).

Each processor has two levels of cache. The off-chip second-level cache memory (L2) is copy-back and write-allocate; dirty victim blocks are placed into a copy-back Dirty Buffer (Dbuf). L2 supports up to three concurrent misses: a read miss (L1 also missed and processor stalls), a write miss (the top entry of CWbuf is not in L2) and a prefetch miss. The first level (on chip) is composed of the CWbuf and of a write-through, no write-allocate cache (L1). Table 3.1 shows sizing, timing and structure for all the hierarchy components. Cache sizes will be detailed in Section 3.3.2. Coherence in L2 and Dbuf is kept using an Illinois-based MESI invalidation protocol [PP84], which uses an additional shared control line in the bus. Associativity, block size and number of sets of L2 and L1, along with the replacement rule of L2, guarantee the inclusion property between cache levels. The CWbuf can delay and aggregate writes belonging to the same L1 cache blocks, and therefore other processors are not able to immediately observe those writes. This buffer is emptied before releasing a lock or acquiring a barrier, and the processor stalls while the CWbuf empties. This behaviour is based on the support the Alpha AXP Architecture

provides for implementing release consistency [Cor96].

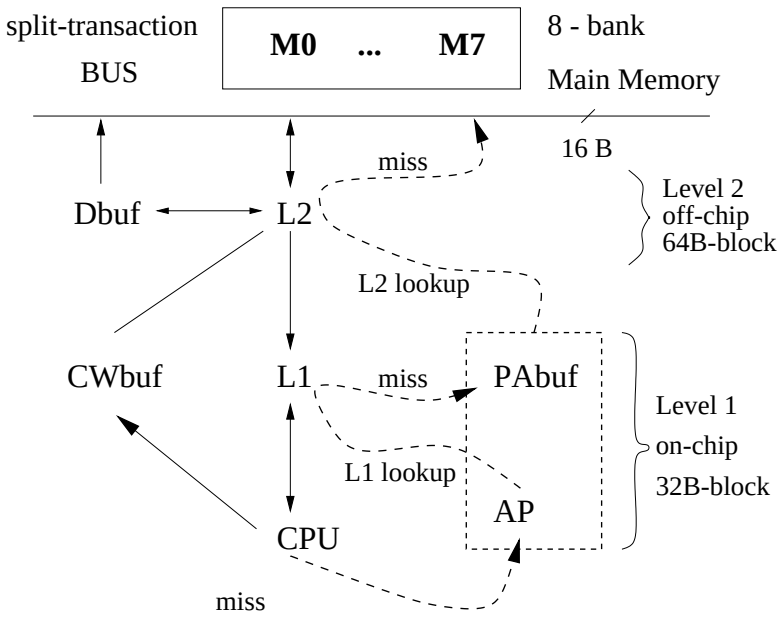

Figure 3.1. Multiprocessor system used for comparing prefetching techniques. The Bases System (no prefetch) excludes the dashed components. (Address predictor and Prefetching Address Buffer).

3.3.1 System with prefetch

The Address Predictor (AP) and the Prefetch Address Buffer (PAbuf) make up the prefetch subsystem. We consider the following four Address Predictors, presented in Section 3.2:

- *1) Sequential Tagged (Ts)*

  This is the tagged sequential prefetching approach as proposed in [Smi78]

- *2) Load Cache (LC)*

  This predictor is based on a conventional Load Cache [BC91]. The Load Cache can record the needed information to recognize sequential, stride, single load linked list and index list traversal accesses. However, as suggested in the previous chapter we do not add any list prediction ability, due to the practical inexistence of the right patterns in the applications that we use for our experimentation.

Component	Timing (CPU cycles)	# Entries or size	Structure
Main Memory	32	8 banks	64B block, interleaved
Bus	4: 4: 32: 4x4	16B width	split-transaction
L2 cache	4: rd/wr hit from L1 8: fill 64B from bus	see Table 5	4-way, 64B block Data Only
Dbuf (Dirty Buffer)	L2+1: hit from L1	6 entr 64B/entry	FIFO
CWbuf (Coalescing Write Buff.)	1: rd hit from L1 6 entr	6 entr 32B/entry	FIFO
L1 cache	1: rd/wr hit; 2: fill from L2	see Table 5	1-way, 32B block Data Only
PAbuf (Prefetch Address Buff.)	1	6 entr	FIFO, filter addresses already present
Load Caches: LC, LCm, LCms	1	16 entr	1-way + data path for computing address predictions

Table 3.1. Default sizings, basic timing and structure for memory, bus, caches and buffers.

- *3) Load Cache with on-miss insertion (LCm)*

  This mechanism is based on a conventional Load Cache but uses an on-miss insertion policy [Iba, IVBG98].

- *4) Load Cache with on-miss insertion plus sequential tagged (LCms)*

  This mechanism was also proposed in [Iba, IVBG98]. It combines Ts and LCm to operate in parallel. The advantage of combining both mechanisms is that we are able to detect not only sequential and stride prefetches followed by a single load, but also sequential accesses coming from different load streams.

The Prefetch Subsystem of this model is described in [GBIV01], and works as follows. According to the internals of each address predictor, a particular stream of prefetch lookups is issued to the first level. Dedicated ports allow CPU demands and prefetches to proceed in parallel. A hit ends up the prefetching activity, whereas a miss (both in CWBuf and L1) pushes the predicted address in the PAbuf, which will contend (with the lowest priority) for the only port of the second level cache. No other prefetch request is issued to the second level until the previous one is serviced. A block prefetched from main memory is loaded in both cache levels.

3.3.2 Workload and Methodology

We have selected a subset of the applications evaluated in the previous chapter (Section 2.3 and 2.4). We have focused on two kernels (FFT and Radix) and four applications (FMM, Radiosity, Ocean and Ocean-non). The applications have been selected to have a mixed of applications with different characteristics. For instance, Ocean has been chosen because more than 80% of the executed loads follow a sequential pattern. In Ocean-non the percentage of sequential decreases to a 30%, which is also the percentage of scalar and stride patterns for this application. Radix and FMM are two of the applications with the lower percentage of recognizable patterns (when considered the learning time, this percentage is in both cases below 50%). Radix and Radiosity are the two remaining applications with the larger percentage of stride pattern.

Since we concentrate on the effects of data prefetching, we disregard the time or bandwidth invested in instruction processing, as in other related works (e.g. [DS96, TE93]). We have arranged the experiments into two sets: Set 1 models a system that strongly presses the memory system, while Set 2 features cache sizes closer to current organizations (see Table 3.2), WS1 and WS2 are respectively the primary and secondary working sets stated in [WOT+95] for 32 processors. The size ratio between levels is 1:16 for Set 1 and 1:32 for Set 2.

	SET1		SET2	
	L1	L2	L1	L2
FMM	2 KB < WS1	32 KB > WS1	8 KB < WS1	256 KB < WS2
FFT Ocean Ocean-non	8 KB < WS1	128 KB > WS1	32 KB < WS2	1MB > WS2
Radiosity	2 KB < WS1	32 KB > WS1	8 KB < WS1	256 KB > WS1
Radix	8KB < WS1	128 KB < WS1	32 KB < WS1	1MB > WS1

Table 3.2. Selected cache sizes for the two experimental sets and their relation with the working sets.

3.4 Performance Model

In order to evaluate a prefetching activity, the most important measure is the execution time. However, to better understand the results, the study of other metrics is needed. In this thesis we study the miss-ratio and we extend the performance model for uniprocessors presented in [Iba, IVBG98] to a multiprocessor system. This resulting model is useful to analyze the behaviour of a prefetch mechanism in a multiprocessor system with a memory hierarchy of two cache levels. The performance model is shown in Figure 3.2. All ratios in the Figure are relative to the number of load instructions.

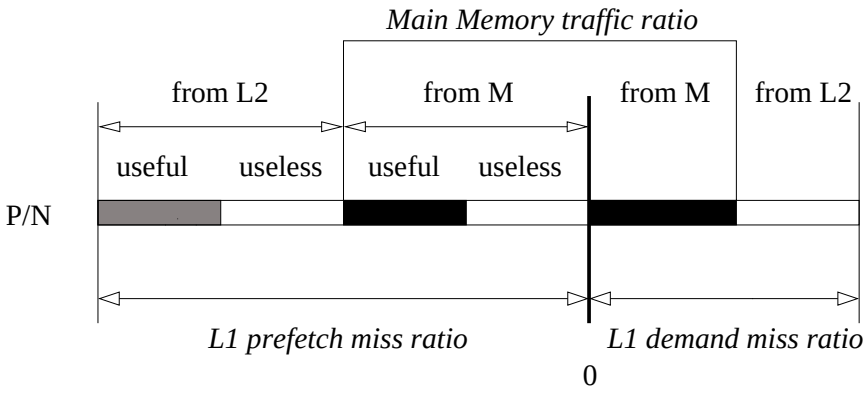

Figure 3.2. Performance model showing the demand and prefetch miss ratios of interest.

The bar on the right of the zero axis is the L1 load miss ratio, split into the misses serviced either from Main Memory or from L2 (*L1 demand miss ratio = from M + from L2*).

The bar on the left is the L1 miss ratio for prefetch requests (*L1 prefetch miss ratio*). In this bar misses are first classified according to the supply source (*Main Memory or L2*) and further divided by their utility (*useful* and *useless*): *L1 prefetch miss ratio = from M* (useful + useless) + *from L2* (useful + useless). A *useless* prefetch means data prefetched but a) never demanded by the processor, or b) replaced or invalidated before being requested.

Note that by adding the two *from M* miss ratios (demand and prefetch) we obtain the main memory requests per read reference, or *Main Memory traffic* ratio from now on. On the other hand, the total bar length (L1 demand + L1 prefetch miss ratio) equals the L2 requests per read reference, or *L2 traffic ratio*. Finally, the number P/N on the left is the number of prefetch requests per every 100 loads, a measure of the lookup pressure on L1.

The ultimate goal of any prefetch system is to reduce the execution time. This goal can be achieved by:

- Decreasing the L1 miss ratio due to CPU demands (*L1 demand miss ratio*).
- Keeping the *L2 traffic ratio* low.
- Keeping the *Main Memory traffic ratio* low.

In a real system with bandwidth restrictions, these three parameters must be balanced. Prefetch mechanisms can be helpful at reducing the L1 demand miss ratio, but this may not reduce the execution time of a program if they increase the L2 or main memory traffic ratio

due to useless prefetches. In our model, an active prefetcher will appear as a prefetcher with a long left-hand bar, in which case the useless areas must be considered carefully. For a particular prefetcher, if the right bar appears shifted to the left when compared to the bar without prefetching, it means that the prefetchers reduces L1 demand miss ratio. All these three parameters can easily be identified with our miss-ratio performance model. Therefore, we consider our model to be useful to highlight what we can expect from a particular prefetcher.

3.5 Evaluation

3.5.1 Base System

Before evaluating the prefetching mechanisms, we show results for the system without prefetch. Table 3.3 shows the (*L1 demand miss ratio*), while Figure 3.3 shows (*Speedup, average read access time (Ta) in CPU cycles* and *bus utilization (BU)*). The number of processors ranges from 1 to 32.

The data displayed in the Table 3.3 and Figure 3.3 show the scenario where we are going to apply prefetch. It can be observed that the operating point in Set 1 has been selected in order to get substantial miss ratios. When rising the number of processors, BU almost saturates in several cases, particularly in FMM and Ocean. On the contrary, miss ratios and BU are much lower in Set 2. Ta increases with the number of processors in most cases, because more processors compete for the shared bus and same number of memory banks. The progressive saturation of the bus dominates the effect of having a greater global cache size when we have more processors. We have also calculated the invalidation rates with respect to the total number of memory references, for each program and varying the number of processors. Differences are negligible when prefetching is applied, and we will not devote further attention to them.

The Base system miss ratios (Table 3.3), shows a marked stability with independence of the processor count. This allow us to concentrate in a 16 processor system to analyze the miss ratios of the whole set of prefetching alternatives.

3.5.2 Discussion of Prefetching Alternatives

Miss Ratios

Firstly, we will analyze the miss ratios. Figure 3.4 shows the miss-ratios of the applications in Section 3.3.2, using the performance model described in Section 3.4. We show results for the four considered prefetching alternatives against the Base system, and for 16 processors.

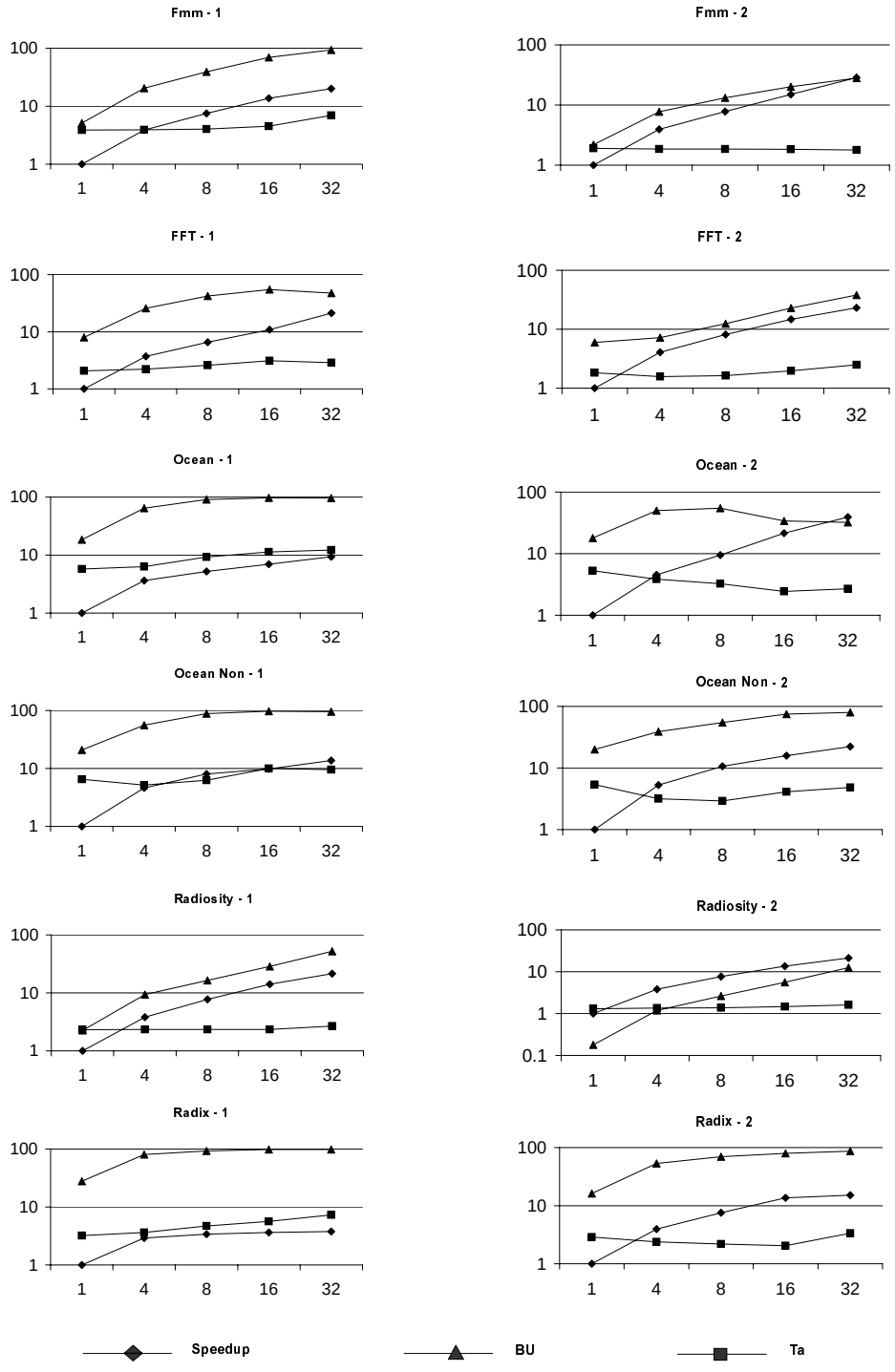

Figure 3.3. Speedup, Ta (cycles) and BU (in %) vs. number of processors, for the Base system.

		Number of processors				
		32	16	8	4	1
FMM	Set 1	21.8	22.3	22.4	22.7	23.8
	Set 2	8.0	8.1	8.2	8.2	8.7
FFT	Set 1	4.8	4.8	4.8	4.8	4.0
	Set 2	2.9	2.9	2.9	2.9	2.9
Ocean	Set 1	13.5	13.4	13.7	14.0	13.5
	Set 2	7.6	9.9	12.3	11.7	12.4
Ocean non	Set 1	11.0	11.4	11.2	11.8	14.5
	Set 2	8.4	9.5	9.9	10.5	13.2
Radiosity	Set 1	12.2	12.1	12.6	11.6	12.0
	Set 2	5.4	5.3	4.9	4.8	4.3
Radix	Set 1	8.5	7.8	7.4	7.1	7.1
	Set 2	5.6	5.6	5.6	5.5	5.5

Table 3.3. L1 demand miss ratios of loads for the Base system.

Our observations are:

- When compared against the Base system, the Main Memory traffic ratio shifts to the left. Tagged- Sequential prefetchers (Ts and LCms) increase this traffic in FFT and Ocean Non.
- With respect to L1 demand miss ratio, the prefetch mechanisms can be divided into two groups: a) LC and LCm and b) Ts and LCms. The second group achieves stronger reductions, except for FMM and Ocean Non, the two programs with higher percentages of STR pattern.
- LC and LCm generate a small number of useless prefetch requests. On the other hand, the Ts and LCms can trigger many useless prefetches, specially for those applications with a small fraction of sequential pattern (FMM and Radiosity in Figure 2.3 in previous chapter), where traffic from L2 to L1 due to prefetching is similar and high for these prefetchers.
- LCms, which combines requests from two predictors, usually makes the highest pressure on L1 (P/N). On the other hand, P/N in LC only depends on the recognized patterns and on the number of entries. Thus, note that P/N is the same in both Sets for this prefetcher, even though cache sizes are very different. Remember, however, that in our system prefetch requests do not stall the processor, since we have a dedicated prefetch port in L1.

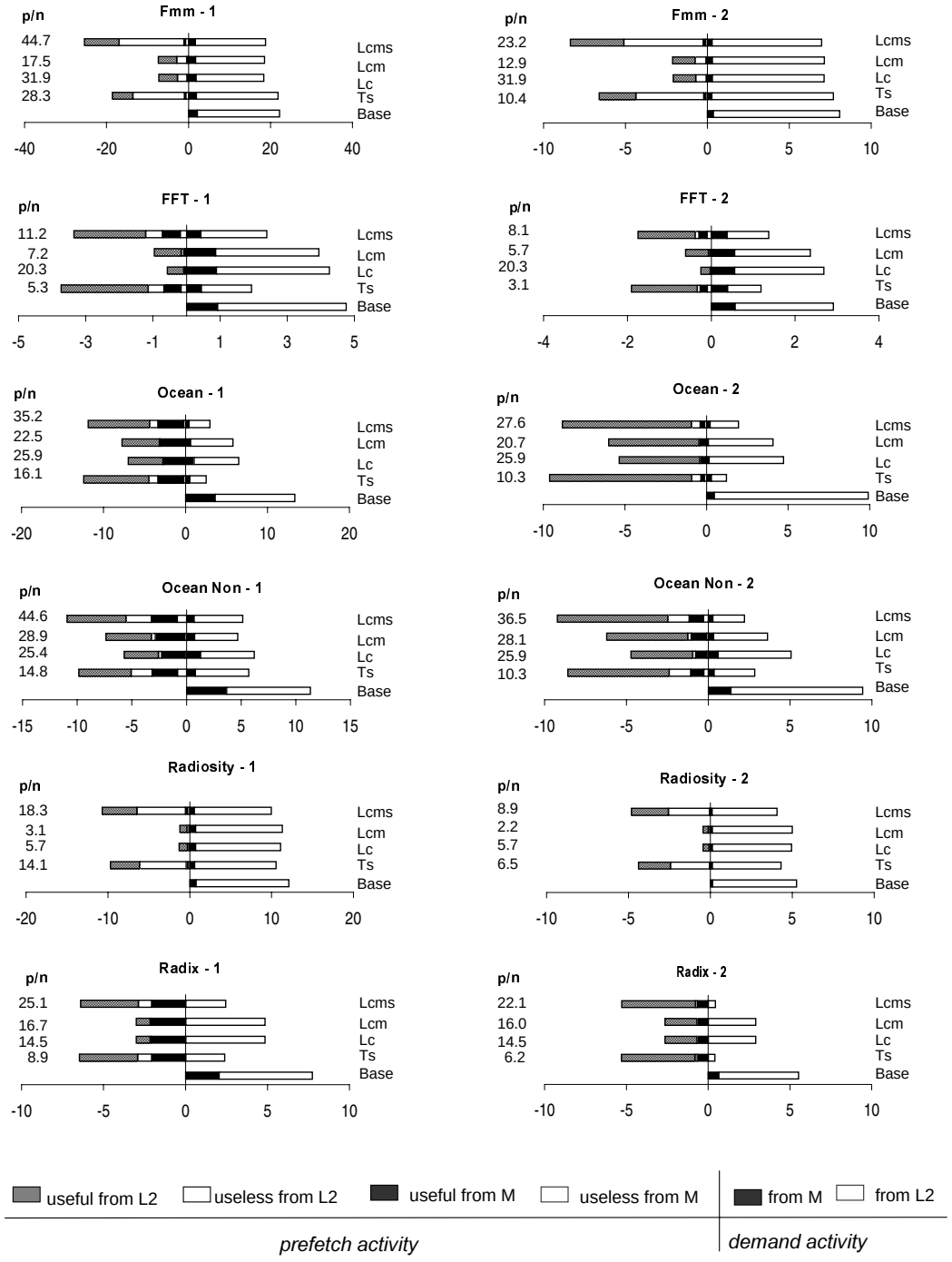

Figure 3.4. Demand and prefetch miss ratios, showing the behaviour of the Base system and Ts, LC, LCm, LCms Prefetchers for 16 processors. On the left, we report the number of prefetch request per every 100 loads (p/n)

Execution Time

Now, let us focus on the performance of prefetchers against the Base System. Table 3.4 shows the speedups (percentage) relative to the Base System achieved by LCm, LCms and Ts for 1, 4, 8, 16 and 32 processors respectively. Note that the speedup numbers in Table 3.4 show the execution time improvement of the system with prefetch with respect to the same system without prefetch and with the same number of processors. Thus, speedups for a given number of processors are not comparable with those for other number of processors. Numbers in bold indicate the best speedup for a given number of processors. However, we do not show results for LC, since it is the best prefetcher only in Radiosity (16 processors, Set 1) and it degrades performance less than the others only in Radix (32 processors, Set 1).

Relations between pattern distribution and performance can be clearly observed only in certain cases. Thus, in the two programs with the higher percentage of stride pattern (FMM and Ocean Non), either LCm, LCms or both achieve stronger reductions in demand miss ratio and perform always better than Ts (Table 3.3). In Ocean (the application with a higher percentage of sequential patterns) Ts eliminates more demand misses, but outperforms LCm and LCms only in 4 over 10 cases, yet differences in the speedups relative to the Base System (Table 3.4) are often negligible. For the other three applications, these relations are not so clear. Looking at LCm and LCms, one of the two is the best prefetcher in 83% of cases, improving performance with respect to the Base System in 70% of cases. LCm, LCms and Ts yield negative speedups respectively in 8.3%, 9% and 12% of cases. There are 5 cases (3%) where all prefetchers degrade performance respect to the Base System.

Summing up, if we take the best one from LCm and LCms in each case, we obtain in average relative speedups of 9.97%, 5.03%, 2.54% and 0.43% for 4, 8, 16, and 32 processors respectively, where Ts yields 9.25%, 4%, 0.17% and -5.22%. The implementation cost of LCms is less than 24 B per LC entry [IVBG98], totalizing less than 400 B, plus the control logic, and a bit per L1 line for supporting Ts. This represents a little cost regarding the cache sizes we have dealt with in Set 1, and a negligible one considering the (more realistic) sizes used in Set 2. Consequently, we believe it is worth incorporating a LCms mechanism, where the sequential tagged predictor could be disconnected by software, either by the programmer or the compiler. This decision could be taken based in the number of processors executing the application, or the characteristics of the application. In those cases where the sequential prefetcher gets disengaged there will be some unused hardware, but of very low cost.

		SET 1			SET2		
		LCm	LCms	Ts	LCm	LCms	Ts
FMM	1	**3.82**	1.92	-1.47	**1.51**	1.74	0.97
	4	**4.90**	0.95	-1.19	**1.47**	1.45	0.62
	8	**4.34**	0.33	-2.12	**1.31**	0.62	-0.09
	16	**4.35**	-4.53	-7.04	**1.07**	-0.68	-0.62
	32	**1.19**	-17.10	-17.38	**0.93**	-2.04	-1.94
FFT	1	7.00	**8.99**	8.29	6.80	**7.88**	6.86
	4	1.42	**5.91**	5.76	1.81	**5.25**	4.82
	8	2.51	3.70	**4.10**	0.51	**1.71**	**1.71**
	16	1.10	1.10	**1.76**	**0.31**	-4.35	-3.11
	32	-0.43	-17.45	-17.02	**0.98**	-14.15	-14.63
OCEAN	1	16.65	22.97	**23.02**	18.35	24.07	**24.31**
	4	12.23	**15.47**	15.28	16.29	19.57	**20.58**
	8	2.90	4.85	**6.13**	8.42	**11.26**	11.03
	16	8.58	**9.02**	6.41	**5.07**	2.63	2.63
	32	-1.27	-3.68	-2.11	**2.41**	-2.58	-1.85
OC NON	1	**28.58**	25.07	19.29	24.96	**28.27**	24.53
	4	20.32	**22.77**	21.35	17.56	**21.63**	20.85
	8	**6.84**	6.36	6.54	10.86	**12.57**	12.24
	16	**2.00**	-2.41	-2.56	**2.23**	1.12	1.61
	32	-0.09	-4.78	-3.84	-1.44	-3.80	-2.34
RADIOSITY	1	0.79	**3.33**	2.41	0.39	**1.41**	1.05
	4	3.54	**3.65**	3.58	**2.10**	1.26	0.90
	8	1.04	**2.80**	1.83	0.90	**1.47**	0.88
	16	-0.54	0.24	-0.17	0.65	1.06	**1.61**
	32	**0.77**	-0.31	-1.40	0.63	**1.02**	-0.25
RADIX	1	8.67	**12.74**	12.39	13.65	20.10	**20.15**
	4	4.03	**6.49**	6.09	7.86	**12.46**	12.32
	8	2.01	**3.07**	1.39	0.18	**6.42**	4.31
	16	-0.63	-0.69	-0.85	1.66	**4.66**	2.33
	32	-1.28	-2.29	-2.22	0.55	**2.39**	**2.39**

Table 3.4. Speedups of LCm and LCms and Ts relative to the Base System

3.6 Summary

In this chapter, we have analyzed four cost-effective hardware prefetching techniques that can succeed in a bus-based multiprocessor. These four hardware prefetching approaches have been tested against a system with no prefetch, varying the number of processors from 1 to 32. For the experiments a subset of the SPLASH-2 applications have been used, and two experimental sets have been arranged for a two-level cache system. Our results show that the the performance of the different prefetchers depend more on the characteristics of the applications than on the cache sizes. LCms appears to be a suitable prefetcher, even though it degrades performance in some cases where bus utilization is near saturation and any moderate increase in the traffic negatively affects global performance.

Observing results from LCm and LCms together, we suggest considering an LCms mechanism where sequential prefetching can be activated or disconnected by software when compiling or by the programmer. Taking the best from the two prefetchers, we obtain in average relative speedups of 9.97%, 5.03%, 2.54% and 0.43% for 4, 8, 16, and 32 processors respectively, at a truly low cost, considering a bus with a reasonable bandwidth for a modern modest-sized bus-based multiprocessor, where a traditional sequential prefetcher yields 9.25%, 4%, 0.17% and -5.22%.

.

References

[AI92] B. R. Allison and C. Van Ingen. Technical Description of the DEC 7000 and DEC 10000 AXP Family. *Digital Technical Journal*, 4(4):1–1, Special Issue 1992.

[BC91] J.L. Baer and T.F. Chen. An Effective on-chip Preloading Scheme to Reduce Data Access Penalty. In *Proc. of Supercomputing'91*, pages 176–186, 1991.

[CB94] T.F. Chen and J.L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture (ISCA'94)*, pages 223–232, April 1994.

[CC98] C.H. Chi and C.M. Cheung. Hardware Prefetching for Pointer Data References. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 377–384, 1998.

[Cor96] Digital Equipment Corporation. Alpha Architecture Handbook. Version 3. October 1996.

[DDS93] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 Int'l Conference on Parallel Processing (ICPP'93)*, volume 1, pages 56–63, August 1993.

[DS95] F. Dahlgren and P. Stenström. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of the 1st Int'l Symposium on High-Performance Computer Architecture (HPCA'95)*, pages 68–77, January 1995.

[DS96] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[DS98] F. Dahlgren and P. Stenström. Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 47(18):385–398, October 1998.

[FP92] J.W.C. Fu and J.H. Patel. Stride Directed Prefetching in Scalar Processors. In *Proc. of the 25th MICRO*, pages 102–110, 1992.

[GBIV01] M.J. Garzarán, J. Briz, P. Ibáñez, and V. Viñals. Hardware Prefetching in Bus-based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In *Proc. of the 2001 Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 345–354, 2001.

[Iba] Pablo Ibáñez. Gestión Multinivel y Prebúsqueda Hardware en Memorias Cache Integradas. Ph.D. Thesis, Universidad de Zaragoza, Departamento de Informática e Ingeniería de Sistemas., July 1998.

[IVBG98] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 369–376, 1998.

[JG97] D. Josephand and D. Grunwald. Prefetching Using Markow Predictors. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture (ISCA'97)*, pages 252–263, June 1997.

[JG99] D. Joseph and D. Grundwald. Prefetching using Markov Predictors. *IEEE Trans. on Computers*, 48(2):121–133, February 1999.

[JT93] Y. Jegou and O. Temam. Speculative Prefetching. In *Proc. of the 1993 Int'l Conference on Supercomputing (ICS'93)*, pages 1–11, 1993.

[KCPT95] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, pages 1250–1264, December 1996. A shorter version appeared in Proc. of the 9th Int'l Conference on Supercomputing (ICS'95), pages 255-264, July 1995.

[MG91] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[MH96] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proc. of the 1996 Int'l Conference on Supercomputing (ICS'96)*, pages 133–143, 1996.

[PP84] M. S. Papamarcos and J. H. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. of the 11th Annual Int'l Symposium on Computer Architecture (ISCA'84)*, pages 348–354, June 1984.

[PTA97] V.S. Pai, P. Tanganathan, and S.V. Adve. The Impact of Instruction-Level Parallelism onn Multiprocessor Performance and Simulation Methodology. In *3rd Int'l Symposium on High-Performance Computer Architecture (HPCA'97)*, pages 72–83, February 1997.

[RIVL00] L. Ramos, P. Ibáñez, V. Viñals, and J.M. Llabería. Modeling Load Address Behaviour Through Recurrences. In *Proc. of the 2000 Int'l Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pages 101–108, April 2000.

[RMS98] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 115–126, October 1998.

[Smi78] Alan Jay Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[Smi82] A. J. Smith. Cache Memories. In *Computing Surveys*, pages 473–530, September 1982.

[TE93] D.M. Tullsen and S.J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 278–288, may 1993.

[TE95] D.M. Tullsen and S.J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Trans. on Computer Systems*, 13(1):57–58, February 1995.

[VF94] J.E. Veenstra and R.J. Fowler. The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors. Technical Report number 490, Computer Science Departament. Univ. of Rochester. New York, 1994.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[ZT95] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 188–199, June 22–24, 1995.

Chapter 4

Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors

Reductions are important and time-consuming operations in many scientific codes. Effective parallelization of reductions is a critical transformation for loop parallelization, especially for sparse, dynamic applications. Unfortunately, conventional reduction parallelization algorithms are not scalable.

In this chapter, we present new architectural support that significantly speeds-up parallel reduction and makes it scalable in shared-memory multiprocessors. The required architectural changes are mostly confined to the directory controllers. Experimental results based on simulations show that the proposed support is very effective. While conventional software-only reduction parallelization delivers average speedups of only 2.7 for 16 processors, our scheme delivers average speedups of 7.6.

4.1 Introduction

During the last decade, programmers have obtained increasing help from parallelizing compilers. Such compilers help detect and exploit parallelism in sequential programs. They also perform other transformations to reduce or hide memory latency, which is crucial in modern parallel machines.

In scientific codes, an important class of operations that compilers have attempted to parallelize is *reduction* operations. A reduction operation occurs when an associative and

commutative operator $\otimes$ operates on a variable x as in $x = x \otimes expression$, where x does not occur in *expression* or in any other place in the loop.

Parallelization of reductions is crucial to the overall performance of many parallel codes. Transforming reductions for parallel execution requires two steps. First, data dependence or equivalent analysis is needed to prove that the operation is indeed a reduction. Second, the sequential computation of the reduction must be replaced with a parallel algorithm.

In parallel machines of medium to large size, the reduction algorithm is often replaced by a parallel prefix or recursive doubling computation [Kru86, Lei92]. For reductions on array elements, a typical implementation is to have each processor accumulate partial reduction results in a private array. Then, after the loop is executed, a cross-processor merging phase combines the partial results of all the processors into the original, shared array.

Unfortunately, such an algorithm can be very inefficient in scalable shared-memory machines when the reduction array is large and sparsely accessed. Indeed, the merging phase of the algorithm induces many remote memory accesses and its work does not decrease with the number of processors. As a result, parallel reduction is slow and not scalable.

In this chapter, we propose new architectural support to speed-up parallel reductions in scalable shared-memory multiprocessors. Our support eliminates the need for the costly merging phase, and effectively realizes truly-scalable parallel reduction. The proposed support consists of architectural modifications that are mostly confined to the directory controllers.

Results based on simulations show that the proposed support is very effective. While conventional software-only parallelization delivers an average speedup of 2.7 for 16 processors, the proposed scheme delivers an average speedup of 7.6.

This chapter is organized as follows: Section 4.2 discusses the parallelization of reductions in software, Section 4.3 presents our new architectural support, Section 4.4 describes our evaluation methodology, Section 4.5 evaluates our proposed support, Section 4.6 outlines how the support can also be used for another problem, Section 4.7 presents related work, and Section 4.8 concludes.

4.2 Parallelization of Reductions in Software

4.2.1 Background Concepts

A loop can be executed in parallel without synchronization only if its outcome does not depend upon the execution order of different iterations. To determine whether or not the order of iterations affects the semantics of the loop, we need to analyze the data dependences across iterations (or cross-iteration dependences) [Ban88]. There are three types of data dependences: *flow* (read after write), *anti* (write after read), and *output* (write after write).

If there are no dependences across iterations, the loop can be executed in parallel. Such a loop is called a *doall* loop. If there are cross-iteration dependences, we must insert synchronization or eliminate the dependences before we can execute the loop in parallel.

If there are only anti or output dependences in the loop, we can eliminate them by applying *privatization*. With privatization, each processor creates a private copy of the variables that cause anti or output dependences. During the parallel execution, each processor operates on its own private copy.

Figure 4.1(a) shows an example of a loop that can be parallelized through privatization. There is an anti dependence between the read to variable *Temp* in line 4 and the write to *Temp* in line 2 in the next iteration. Furthermore, there is an output dependence between the write to *Temp* in line 2 in one iteration and the next one. By privatizing *Temp*, these dependences are removed and the loop can be executed in parallel.

```
1    for(i=0;i<n;i+=2){
2        Temp=a[i+1];
3        a[i+1]=a[i];
4        a[i]=Temp;
5    }
```

(a)

```
1    for(i=1;i<n;i++)
2        A[i]+=A[i-1];
```

(b)

Figure 4.1. Loops with anti and output dependences (a) and flow dependences (b).

If there are flow dependences across iterations, the loop cannot generally be executed in parallel. For example, the loop in Figure 4.1(b) has a flow dependence in line 2 between consecutive iterations. In this case, iteration i needs the value that is produced in iteration $i-1$. As a result, the loop cannot be executed in parallel.

4.2.2 Parallelizing Reductions

A special and frequent case of flow dependence occurs in loops that perform reduction operations. A reduction operation occurs when an associative and commutative operator $\otimes$ operates on a variable x as in $x = x \otimes expression$, where x does not occur in $expression$ or in any other place in the loop. In such a case, x is a reduction variable.

A simplified example of reduction is shown in Figure 4.2. In the figure, array w is a reduction variable. Note that the pattern of access to a reduction variable is a read followed by a write. Therefore, there may be flow dependences across iterations. As a result, the loop cannot be run in parallel.

```
1    for(i=0;i<Nodes;i++)
2        w[x[i]]+=expression;
```

Figure 4.2. Loop with a reduction operation.

Parallelizing loops with reductions involves two steps: recognizing the reduction variable and transforming the loop for parallelism. Recognizing the reduction variable involves several steps [Zim91]. First, the compiler syntactically pattern-matches the loop statements with the template of a general reduction ($x = x \otimes expression$). In our example, the statement in line 2 matches the pattern. Then, the operator (+ in our example) is checked to determine if it is commutative and associative. Finally, data dependence or equivalent analysis is performed to verify that the suspected reduction variable is not accessed anywhere else in the loop. In our example, all of these conditions are satisfied for w.

Once the reduction variable is recognized, the loop is transformed by replacing the reduction statement with an equivalent parallel algorithm. For this, there are several known methods. The two most common ones are as follows:

- Enclose the access to the reduction variable in an unordered critical section [EHLP91, Zim91]. Alternatively, we can access the variable with an atomic *fetch-and-op* operation. The main drawback of this method is that it is not scalable, as the contention for the critical section increases with the number of processors. Thus, it is recommended only for low-contention reductions.

- Exploit the fact that a reduction operation is an associative and commutative recurrence. Therefore, it can be parallelized using a parallel prefix or a recursive doubling algorithm [Kru86, Lei92]. This approach is more scalable.

For reductions on array elements, a commonly-used implementation of the second method is to create, for each processor, a private version of the reduction array initialized with the neutral element of the reduction operator. During the execution of the parallelized loop, each processor accumulates partial results in its private array. Then, after the loop is executed, a cross-processor merging phase combines the partial results of all the processors into the shared array.

Using this approach, our example loop from Figure 4.2 gets transformed into the parallel loop of Figure 4.3. For simplicity, static scheduling is used and the code for forking and joining is omitted. Thus, we show only the code executed by each processor.

In the parallelized loop, each processor has its own array *w_priv[PID]*, where PID is the processor ID. First, each processor initializes its array with the neutral element of the reduction operator (lines 1-2). In our example, the neutral element is 0 because the reduction operator is addition. Next, each processor gets a portion of the loop and executes it, performing the reduction operation on its array *w_priv[PID]* (lines 3-4). After that, all processors synchronize (line 5). Then, they all perform a *Merging* step, where the partial results accumulated in the different *w_priv[PID]* arrays are merged into the shared array *w*.

```
// Initialize the private reduction array
1     for(i=0;i<NumCols;i++)
2         w_priv[PID][i]=0;
// The range 0..Nodes is split among the processors
3     for(i=MyNodesBegin;i<MyNodesEnd;i++)
4         w_priv[PID][x[i]]+=expression;
5     barrier();
//The range of indices of w is split among processors
6     for(i=MyColsBegin;i<MyColsEnd;i++)
7         for(p=0;p<NumProcessors;p++)
8             w[i]+=w_priv[p][i];
9     barrier();
```

Figure 4.3. Code resulting from parallelizing the loop in Figure 4.2.

In the case of scalars, this merging step can be parallelized through recursive doubling. In the case of arrays, however, it is more efficient to parallelize it by having each processor perform merging for a sub-range of the shared array. Thus, in our example each processor processes a portion of *w* element by element (line 6). For each element, the processor in charge of processing it takes the partial result of each processor (line 7) and combines it

into its shared counterpart (line 8). Finally, another global synchronization is performed (line 9), to guarantee that the subsequent code accesses see only the fully merged array `w`.

4.2.3 Drawbacks in Scalable Multiprocessors

This implementation of reduction parallelization has two important drawbacks in scalable shared-memory multiprocessors with large memory access latencies: many remote misses in the merging phase and cache sweeping in the initialization and merging phases.

The merging phase necessarily suffers many remote misses. Indeed, for each shared array element that a processor accesses in line 8, all but one of the corresponding private array elements are in remote memory locations. Because of this, the merging operation (also called *merge-out*) can be very time consuming.

Note that the time needed to perform the merging does not decrease when more processors are used. With more processors, each processor has to perform combining for fewer elements of the shared array. However, each element requires more work because more partial results need to be combined. Specifically, consider an array of size s and p processors. The merging step requires that each processor combine p sub-arrays of size s/p. As a result, the total merging time is proportional to $p * s/p = s$, which does not depend on the number of processors.

The problem gets worse when the access pattern of the reduction is sparse. In this case, the merging operation performs a lot of unnecessary work, since it operates on many elements that still contain the neutral element. To improve this case, each processor could use a compact private data structure such as a hash table instead of a full private array. With this approach, however, improving the merging phase comes at the cost of slowing down the main computation phase. The reason is that addressing this compact structure requires indirection, which is more expensive than the simple addressing of array elements.

The second problem, namely cache sweeping, occurs in the initialization (lines 1-2) and merging (lines 6-8) phases. Cache sweeping in the initialization may cause additional cache misses in the main computation phase (lines 3-4). Cache sweeping in the merging phase may cause additional misses in the code that follows the reduction loop.

4.3 Private Cache-Line Reduction (PCLR)

To address the problems discussed in Section 4.2.3, we propose to add new architectural support to scalable shared-memory multiprocessors [DGP+02, GPZ+01]. We call the new scheme *Private Cache-Line Reduction (PCLR)*. In this section, we give an overview of the

scheme and then propose an implementation.

4.3.1 Overview of PCLR

The essence of PCLR is that each processor participating in the reduction uses *non-coherent lines in its cache* as temporary private storage to accumulate its partial results of the reduction. Moreover, if these lines are displaced from the cache, their value is *automatically accumulated* onto the shared reduction variable in memory. Finally, since the cache lines are non-coherent, cache misses are satisfied from within the local node by returning a line *filled with neutral elements*. Figure 4.4 shows a representation of the scheme.

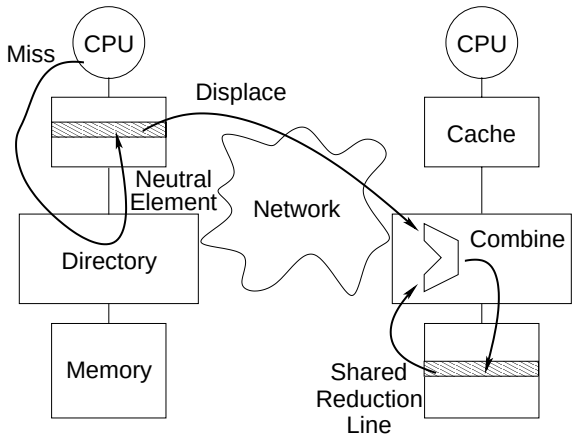

Figure 4.4. Representation of how PCLR works.

With this approach, the processors are relieved of the initialization and merge-out work, therefore eliminating the two problems pointed out in Section 4.2.3. Also, since the approach is still based on computing partial results and combining them, the reduction is performed with no critical sections.

The initialization phase is avoided by initializing the reduction lines on demand, as they are brought into the cache on cache misses. Since the cache is used as private storage to accumulate the partial results, there is no need to allocate any private array in memory. On a cache miss to a reduction line, the local directory controller intercepts the request and services it by supplying a line of neutral elements.

The merging phase is avoided by combining the reduction cache lines in the background, as they are displaced from the cache during parallel loop execution. As each displaced reduction line reaches the home of the shared reduction variable, the directory controller combines its contents with the shared reduction variable in memory. Meanwhile, the processors continue processing the loop without any interruption.

When the parallel loop ends, some partial results may still remain in the caches. They must be explicitly flushed so that they are correctly combined with the shared data before any further code is executed. This flush step takes much less time than the ordinary merging phase of Figure 4.3. There are two reasons for it. First, it has less combining to perform, as most of it has already been performed through displacements during the loop execution. In fact, the work to do is at worst proportional to the size of the cache, rather than to the size of the shared array. The second reason is that the processor issues no remote loads. Instead, it simply sends all the partial results to their homes, where the directory controller combines the data.

With PCLR support, the code in Figure 4.2 becomes the one in Figure 4.5. Note that we have added a call to a function that configures the machine for PCLR before the loop execution. As in Figure 4.3, this example is also simplified by using static scheduling and omitting the forking and joining code. In the rest of this section, we present an implementation of PCLR.

```
1     ConfigHardware(arguments);
// The range 0..Nodes is split among the processors
2     for(i=MyNodesBegin;i<MyNodesEnd;i++)
3           w[x[i]]+=expression;
4     CacheFlush();
5     barrier();
```

Figure 4.5. Parallelized reduction code under PCLR.

4.3.2 Implementation of PCLR

Any implementation of PCLR has to consider the following issues: differentiation of reduction data (Sections 4.3.2 and 4.3.2), support for on-demand initialization (Section 4.3.2) and combining (Section 4.3.2) of lines, configuration of the hardware (Section 4.3.2), and atomicity guarantees (Section 4.3.2). We discuss these issues in this section.

In the following discussion, we assume a CC-NUMA architecture such as the one in Figure 4.4. Each node in the machine has a directory controller that snoops and potentially intervenes on all requests and write-backs issued by the local cache, even if they are directed to remote nodes.

Differentiating Reduction Data

While the data used in reduction operations remain in the cache, they are read and written just like regular, non-reduction data. However cache misses and displacements of reduction data require special treatment. Consequently, any implementation of PCLR has to provide a way to distinguish reduction data from regular data.

A simple way of doing so is to use special load and store instructions for "reduction" accesses. Cache lines accessed by these special instructions are marked as containing reduction data by putting them into a special "reduction" state. In this state, a processor can read and write the line without sending invalidations, even though other processors may be caching the same memory line. Misses by reduction loads and displacements of lines in the reduction state cause special transactions that are recognized by the local and home directories, respectively.

Note that we assume that reduction and regular data never share a cache line. Although it would be possible to enhance our scheme to support line sharing, alignment of reduction data on cache line boundaries is beneficial even without PCLR. Consequently, we assume that the compiler guarantees no line sharing.

In the following, we explain the rest of PCLR assuming this simple approach to differentiating reduction data. In Section 4.3.2, we propose a more advanced scheme for reduction data differentiation that allows using unmodified or slightly modified processors and caches.

On-Demand Initialization of Reduction Lines

When a reduction load misses in the cache, a specially-marked cache line read transaction is issued to the memory system. The local directory controller intercepts the request and satisfies it by returning a line initialized with *neutral elements* for the particular reduction operation. The line is loaded into the cache in the reduction state.

A reduction load may hit in the cache on a line that is not in the reduction state. This may occur if the line had been accessed prior to the reduction loop with plain accesses and happened to linger in the cache. In this case, if the line is in state dirty, it is written back to memory in a plain write-back. Irrespective of its state, the line is then invalidated. Finally, the cache issues a reduction read miss as indicated above.

On-Demand Combining of Partial Results

When a line in the reduction state is displaced from the cache, a specially-marked write-back transaction is issued to the memory system. Once the write-back arrives at its home, the directory controller reads the previous contents of the line from memory, combines it with the newly-arrived partial result, and stores the updated line back to memory. The combining of the lines is done according to the reduction operator in the code, and is performed for every single element in the line. Note that those elements of the displaced line that were not accessed by the processor still contain the neutral element, so the effect of merging them with memory content is that the memory content is unchanged.

To combine the lines, the directory controller has to be enhanced with execution units that support the required reduction operators. Since a cache line contains several individual data elements, such execution units may become a bottleneck if their performance is too low. Luckily, all the elements of a line can be processed in parallel or in a pipelined fashion. Consequently, it is not too difficult to improve the performance by pipelining these execution units or adding more units.

These execution units should include an integer ALU for integer operations. For floating-point operations, having a full floating-point unit would be more general, but would also increase the complexity of the directory controller significantly. Our experience with the applications in Section 4.4.2 suggests that multiplication is rarely used as a reduction operator. Thus, for floating-point operations, having a floating-point adder and comparator is sufficient.

Finally, it is possible that the reduction data had been accessed prior to the reduction loop with plain accesses, and still lingers in several caches when the reduction loop starts. To handle this case, when the home directory controller receives a write-back for the line, it always checks the list of sharer processors for the line in the directory. Note that misses due to the reduction accesses do not go to the home. Thus, the home only has sharing information about non-reduction sharers. If the line is in a (non-reduction) dirty state in a cache, the controller recalls the line and writes it back to shared memory before performing any combining. The controller also sends invalidations to all (non-reduction) sharer processors. After the first reduction write-back of a line, the list of sharers at its home is empty for the remainder of the reduction loop and causes no further invalidation or recall messages.

Configuring the Hardware

Before executing a reduction loop, each processor issues a system call to inform the directory controller in its node about the data type and the operation of the reduction. This is shown in line 1 of Figure 4.5. With this simple approach, we can only support one type of reduction operation per parallel section. In our example of Figure 4.5, the controller must be configured to perform double-precision floating-point addition when it receives a reduction write-back.

Any loop that performs several types of reduction operation must be distributed into multiple loops, so that each loop performs only one type of reduction operation. Fortunately, loops with multiple types of reduction operation are rare.

Finally, the operating system knows if different, time-shared processes want to use different types of reduction operations. If this is the case, the operating system flushes the reduction data from the caches when a process is preempted, and reprograms the directory controller when the process is re-scheduled.

Advanced Differentiation of Reduction Data

In Section 4.3.2 we explained a simple mechanism to distinguish reduction data from regular data and then explained the rest of PCLR using that simple mechanism. Now we propose a more advanced, but equivalent, mechanism that eliminates the need to modify the processor, the caches, or the coherence protocol.

In this scheme, instead of using special instructions, cache states, and protocol transactions to identify reduction data, such data are identified by using *Shadow Addresses* [CHS+99]. The scheme works as follows. In the reduction code, we use a *Shadow Array* instead of the original reduction array. For example, in Figure 4.5, we would use array *w_redu* instead of *w*. This shadow array is mapped to physical addresses that do not contain physical memory. However, such addresses differ from the corresponding physical addresses of the original array in a known manner. For example, they can have their most significant bit flipped. As a result, when a directory controller sees an access that addresses nonexistent memory, it will know two things. First, it will know that it is a reduction access. Second, from the physical address, it will know what location of the original array it refers to.

With this approach, we do not need to modify the hardware of the processor, caches, or coherence protocol. The only requirement is that the machine must be able to address more memory than physically installed. Then, when a directory controller sees a read miss from the local processor to nonexistent memory, it simply returns a line of neutral

elements to the processor. Furthermore, when a directory controller sees the write-back of a line from the local processor to nonexistent memory, it will forward it to the home of the corresponding element of the original array. Finally, when a directory controller receives the write-back of a line from a remote processor, it translates its address to the address of the corresponding element in the original array and combines the incoming data with the data in memory.

This approach requires modest compiler and operating system support. The compiler modifies the reduction code to access a shadow array instead of the original array. It also declares the shadow array and inserts a system call to tell the operating system which array is shadow of which. The operating system has to support the mapping of pages for the shadow array. Specifically, on a page fault in the shadow array, it assigns a nonexistent physical page whose number bears the expected relation to the number assigned to the corresponding original array page. Moreover, if the latter does not exist yet, it is allocated at this time.

Atomicity Concerns and Solutions

In PCLR, a problem occurs if a line with reduction data is displaced from the cache between a read and the corresponding write of the reduction operation. As an example, assume that the value of a variable in the line is X. This value is read into a register and updated to $X+Y$. However, before the result register is written to the cache, the line gets displaced from the cache. In this case, the partial result X will be sent to memory and accumulated onto the shared data. Then, the cache miss will be serviced with the neutral element and the variable in the line will be updated to $X+Y$. Later, a displacement of this line will cause $X+Y$ to be accumulated onto the shared data in memory. Thus, the partial result X will be accumulated onto the shared data twice.

We can solve this problem through *recovery* or *prevention*. Recovery solutions attempt to recover the correct state of computation after the problem has already occurred. Unfortunately, the problem can generally be detected only when the store misses in the cache. In our example, the recovery would involve subtracting X from either the register involved in the miss or the shared location. However, X is unknown at the time the problem is detected, as the shared location contains the combined result of other computations and X, while the offending store has $X+Y$. Instead of attempting to checkpoint the partial results or the shared value in order to enable recovery, we choose to prevent the occurrence of the problem.

Note that reduction lines in a cache do not receive external invalidations or downgrade requests that force them to write-back. Therefore, a miss on a store to a reduction line

can only occur because, between the load and the store, the local processor has brought in a second line that has displaced the one with reduction data. If we ensure that the processor does not perform any access between the load and the store to the reduction variable, the displacement problem should not happen. Unfortunately, modern processors reorder independent memory accesses like those to different words of the same line and, therefore, may induce the problem. Preventing this reordering involves putting a memory fence before the load and after the corresponding store to the reduction variable. This approach is unacceptable because it would limit the performance of PCLR on modern processors.

We propose two different solutions. The first one uses a swap instruction and does not modify the processor hardware. The second one, instead, does not require the ISA of the processor to have a swap instruction, but requires small modifications to pin line caches. We explain each of them in turn.

1) Swap Approach

The idea of this approach is to atomically exchange the neutral element with the memory contents as Figure 4.6 shows. The compiler can replace the load to the reduction variable from the original code in Figure 4.6-(a) with the two instructions of Figure 4.6-(b). The first instruction charges into the register the neutral element, and the *swap*, instruction atomically exchanges the register contents with the memory contents. With this simple modification, if the cache line containing the reduction variable gets displaced before the store executes, the data in the home will not suffer any modification, since it will be operated with the neutral element. Of course, for this mechanism to perform well, the swap instruction should not be implemented as a fence.

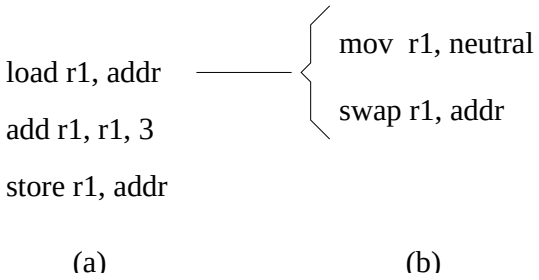

Figure 4.6. Original reduction code (a), and instructions that replace the load to solve the atomicity problem (b).

2) Pin Approach

The second approach that we propose is the *pinning* of a line in the cache between a

reduction load to the line and the corresponding store. We introduce two new instructions, namely *load&pin* and *store&unpin*, and add a small number of *Cache Pin Registers* (CPRs) to the processor. Each CPR has two fields: the tag field which holds the tag of the pinned line, and a *pin count* counter.

When a *load&pin* instruction is executed, a read from the cache is performed. At the same time, a CPR is allocated, its tag is set to the tag of the cache line, and the pin count is set to one. If one of the CPRs already has the tag of the line, its pin count is incremented. When a *store&unpin* instruction is executed, a store to the cache is performed. At the same time, the pin count for the matching CPR is decremented. If after this the pin count is zero, the CPR is freed. Before a displacement of a cache line is allowed, the tags of the CPRs are checked. If any of the active CPRs has a matching tag, the displacement is prevented until the line is no longer pinned in the cache.

With this support, all micro-architectural features found in modern microprocessors can still be used, including out-of-order instruction issue, speculative execution, instruction squashing, and memory renaming. However, care must be taken to keep the CPRs up-to-date. For example, if a speculatively executed *load&pin* has to be squashed, the hardware needs to decrement the corresponding pin count and possibly free the CPR. Similarly, consider memory renaming from a *store&unpin* to a *load&pin* of the same address. In this case, even though the load is transformed into a register-to-register transfer, the CPR for the *load&pin* still needs to be operated on.

Finally, if all CPRs are in use when one more is needed, or a pin counter saturates, the instruction is delayed until a CPR is free or the counter is decremented. Because CPRs are needed to allow instruction reordering by the processor, this delay cannot cause deadlocks. In fact, even if a processor has only one CPR, it can correctly execute any code. With more CPRs, the compiler can be more aggressive about instruction scheduling. In practice, we have found that a small number of CPRs (8) is sufficient to maintain good performance.

4.3.3 Summary

The PCLR scheme addresses the problems of parallel reductions in scalable shared-memory multiprocessors as discussed in Section 4.2.3. PCLR has two main advantages. First, it uses cache lines as the only private storage and initializes them on demand. As a result, there is no need to allocate private data structures or to perform a cache-sweeping initialization loop. Second, it performs the combining of the partial results with their shared counterparts on demand, as the reduction loop executes. As a result, there is no need for a costly merging step that involves sweeping the cache and many remote misses.

All that is needed is to flush the reduction data from the caches at the end of the loop. These two advantages are particularly important when the reduction access patterns are sparse.

Most PCLR modifications are in the directory controllers, which perform special actions on read misses and write backs. With the use of shadow addresses, if the ISA of the processor has a swap or similar instruction no modification is needed to the hardware of the processor. If not, the only modification modification that we introduced to the processor and caches is the ability to pin and unpin lines in the caches through the *load&pin* and *store&unpin* instructions. It can be argued that these instructions could also be useful for other functions in modern processors.

4.4 Evaluation Methodology

We evaluate the PCLR scheme using simulations driven by several applications. In this section, we describe the simulation environment and the applications.

4.4.1 Simulation Environment

We use an execution-driven simulation environment based on an extension to MINT [VF94a] that includes a dynamic superscalar processor model [KT98]. The architecture modelled is a CC-NUMA multiprocessor with up to 16 nodes. Each node contains a fraction of the shared memory and the directory, as well as a processor and a two-level cache hierarchy with a write-back policy. The processor is a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. Table 4.1 lists the main characteristics of the architecture. Contention is accurately modelled in the entire system, except in the network, where it is modelled only at the source and destination ports.

Processor Parameters	Memory Parameters
4-issue dynamic, 1 GHz	L1, L2 size: 32 KB, 512 KB
Int, fp, ld/st FU: 4, 2, 2	L1, L2 assoc: 2 way, 4 way
Inst. window: 64	L1, L2 size: 64 B, 64 B
Pending ld, st: 8, 16	L1, L2 latency: 2, 10 cycles
Branch penalty: 4 cycles	Local memory latency: 104 cycles
Int, fp rename regs: 64, 64	2-hop memory latency: 297 cycles

Table 4.1. Architectural characteristics of the modelled CC-NUMA. The latencies shown measure contention-free round trips from the processor in processor cycles.

The system uses a directory-based cache coherence protocol along the lines of DASH [LLG+92]. Each directory controller has been enhanced with a single double-precision floating-point add unit. Both the directory controller and the floating point-unit are clocked at 1/3 of the processor's frequency. The floating-point unit is fully pipelined, so it can start a new addition every three processor cycles. Its latency is 2 cycles (6 processor cycles). Floating-point addition is the only reduction operation that appears in our applications (Section 4.4.2).

Private data are allocated locally. Pages of shared data are allocated in the memory module of the first processor that accesses them. Our experiments show that this allocation policy for shared data achieves the best performance results for both the baseline and the PCLR system.

4.4.2 Applications

Appl.	Names of Loops	% of Tseq	# of Invoc	#Iters Invoc	#Instr Iter	Red. Ops. per Iter	Red. Array Size (KB)
Euler	*dflux_do[100,200]* *psmoo_do20* *eflux_do[100,200,300]*	84.7	120	59863	118	14	686.6
Equake	*smvp*	50.0	3855	30169	550	22	707.1
Vml	*VecMult_CAB*	89.4	1	4929	135	6	40.0
Charmm	*dynamc_do*	82.8	1	82944	420	54	1947.0
Nbf	*nbf_do50*	99.1	1	128000	1880	200	1000.0
Average		81.2	795	61181	620	59	871.0

Table 4.2. Application characteristics. In *Euler*, we only simulate dflux_do100, and all the numbers except Tseq correspond to this loop.

To evaluate the PCLR system, we use a set of C and Fortran scientific codes. Two of them are applications: *Euler* from HPF-2 [DSH94] and *Equake* from SPECfp2000 [Hen00]. The three other codes are kernels: *Vml* from Sparse BLAS [DMRV95], *Charmm* from [BBO+83], and *Nbf* from the GROMOS molecular dynamics benchmark [GB88]. Out of them, *Equake* and *Vml* are written in C, while the rest are written in Fortran.

All of these codes have loops with reduction operations. Table 4.2 lists the loops that we simulate in each application and their weight relative to the total *sequential* execution time of the application (%Tseq). This value is obtained by profiling the applications on a single-processor Sun Ultra 5 workstation.The table also shows the number of loop invocations during program execution, the average number of iterations per invocation, the average number of instructions per iteration, the average dynamic number of reduction operations per iteration, and the size of the reduction array. Next, a brief description of

each application:

- *Euler* is a 3-D hydrodynamics code that models transient flows involving multiple immiscible fluids.
- *Equake* models earthquake-induced ground motions using an unstructured 3-D finite element model of the area under test. The loop performs a sparse matrix-vector multiplication using subscripted subscripts.
- *Vml* is a kernel for a multiplication of a dense vector and a sparse matrix.
- *Charmm* is a N-body simulation to model macromolecular dynamics, including energy minimization, and Monte Carlo simulations. In this code, each node interacts only with its neighbours. The list of neighbours changes at various times because of the forces applied to them.
- *Nbf* is also a N-body simulation quite similar to Charmm in the sense that each node only interacts with its neighbours. At every step forces are evaluated at each node and applied through a reduction operation.

The loops in Table 4.2 are analyzed by the Polaris parallelizing compiler [BDE+96] or by hand to identify the reduction statements. Then, we modify the code to implement the parallel reduction code for the software and PCLR algorithms, as shown in Sections 4.2.2 and 4.3, respectively. For PCLR, reduction accesses are also marked with special load and store instructions to trigger special PCLR operations (Section 4.3.2) in our simulator.

The data shown in the next section, including speedups, refer only the sections of code described in Table 4.2. Also, since there is a significant variation in speedup figures across applications, average results are reported using the *harmonic mean*.

4.5 Evaluation

4.5.1 Impact of PCLR

We evaluate two different implementations of our PCLR scheme. The first one is an implementation where the directory controller is hardwired. The second implementation utilizes a programmable directory controller, similar to the MAGIC micro-controller in the FLASH multiprocessor [KOH+94]. A programmable controller can provide the functionality required by PCLR without requiring hardware changes. These two implementations of PCLR are compared against a baseline system, which uses a software-only approach

to parallelize reductions. The software-only approach utilizes an algorithm that accumulates partial results in private arrays and merges the data out when the loop is done, as described in Section 4.2.2.

Figure 4.7 compares the execution time of these three systems. The baseline software-only system is *Sw*. The PCLR implementation with a hardwired directory controller is *Hw*, and the implementation with a flexible programmable directory controller is *Flex*. The simulated system is a 16-node multiprocessor. For each application, the bars are normalized to *Sw*, and broken down into time spent in the initialization phase of the *Sw* scheme (*Init*), loop body execution (*Loop*), and time spent merging the partial results at the end of the loop in *Sw* or flushing the caches in *Hw* and *Flex* (*Merge*). The numbers above each bar show the speedup relative to the sequential execution of the code. In the sequential execution, all data were placed on the local memory of the single active processor.

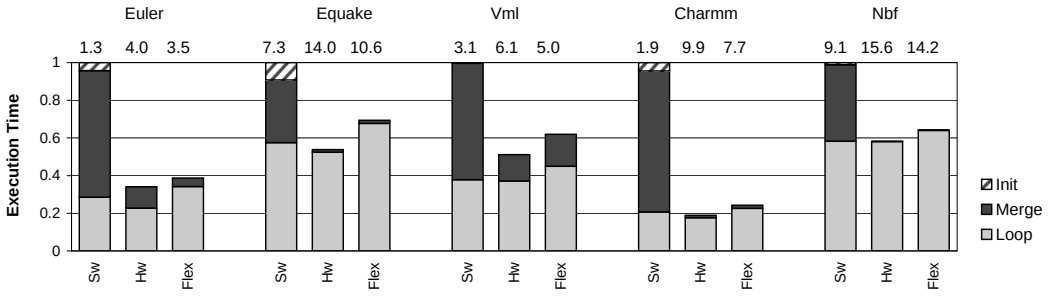

Figure 4.7. Execution time under different schemes for a 16-node multiprocessor. The numbers above the bars are speedups relative to the sequential execution.

The performance of *Hw* and *Flex* improves significantly over *Sw*. This improvement is mainly due to the elimination of the final cross-processor merging step that is required in the software-only implementation. In *Sw*, the work in this merging step is proportional to the size of the reduction array and does not decrease when more processors are available. When this time is significant relative to the time spent in the execution of the main loop, the benefits of PCLR become substantial. For example, in *Charmm* the main parallel loop alone executes with *Sw* 9 times faster than the sequential loop. However, the merging step is responsible for the poor final speedup of *Sw* (1.9 on 16 processors). As mentioned in Section 4.3, a second benefit of PCLR is that the initialization phase is removed. In general, this phase accounts for a relatively small fraction of the *Sw* execution time.

The *Hw* and *Flex* systems always spend less time in *Merge* than the *Sw* system. In

PCLR, *Merge* only accounts for the time spent flushing the caches after the last processor has finished the execution of the parallel loop. When using PCLR, processors do not have to synchronize after the parallel loop. They can start flushing their caches as soon as they finish, and overlap this flush with the execution of the loop on other processors.

In our experiments, it is not not assumed special support to flush only the reduction data. Thus, the L2 cache is traversed and *all* the dirty lines (reduction or not) are written back to memory. In Table 4.3, the column *Lines Flushed* shows the average number of cache lines flushed by each processor. Most of these lines contain reduction data. The column *Lines Displaced* shows the average number of lines with reduction data displaced by each processor during the execution of the main reduction loop.

Appl.	Names of Loops	Lines Flushed	Lines Displaced
Euler	*dflux_do100*	3261	2117
Equake	*smvp*	742	580
Vml	*VecMult_CAB*	168	0
Charmm	*dynamc_do*	1849	330
Nbf	*nbf_do50*	238	1774
Average		1251	960

Table 4.3. Cache lines flushed at the loop's end or displaced during loop execution. The data correspond to a single loop and are collected through simulation of a 16-processor system.

The differences between the *Hw* and the *Flex* schemes are mainly due to two reasons. First, with a software directory controller, all the transactions in the node have to go through the node controller, increasing the contention. Second, the software directory controller takes longer to process individual transactions. To accurately simulate these two effects, in our simulations of *Flex* we have used the cycle counts for response time and occupancy reported for the FLASH directory controller [HKD+94]. For example, a clean read miss is serviced in 11 cycles of the directory controller. Since we assume that the directory controller is clocked at 1/3 of the processor's frequency, this corresponds to 33 cycles of the main processor. The directory controller is occupied during that time.

The figure shows that the speedups in *Flex* are, on the average, only 16% lower than in *Hw* and 136% higher than in *Sw*. Therefore, implementing PCLR using a programmable directory controller is a good trade-off.

Overall, for a 16-node multiprocessor, the *Hw* PCLR scheme achieves an average speedup of 7.6, while the software-only system delivers an average speedup of only 2.7. If PCLR is implemented with a programmable directory controller the average speedup is

6.4.

4.5.2 Scalability of PCLR

To evaluate the scalability of PCLR, a multiprocessor system with 4, 8, and 16 processors is simulated. Figure 4.8 shows the harmonic mean of the speedups delivered by the different mechanisms. It can be seen that PCLR (both *Hw* and *Flex*) scale well. However, the *Sw* scheme scales poorly. As explained in Section 4.2.2, the time of the merging step in *Sw* does not decrease when more processors are available. If the main loop scales well, the merging step limits the achievable speedups according to Amdahl's law.

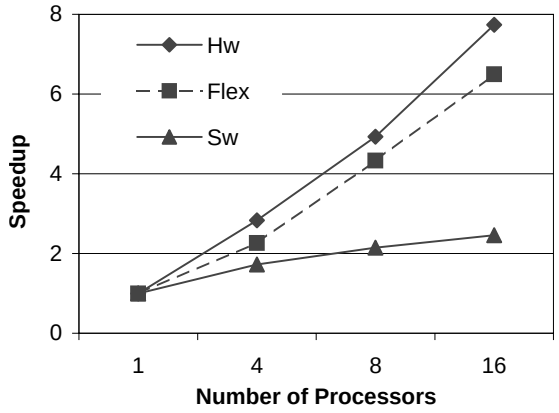

Figure 4.8. Speedups delivered by the different mechanisms (harmonic mean).

4.5.3 Impact of FP-Unit Speed

In previous sections it is assumed that the floating-point unit in the directory controller was clocked at 1/3 of the processor's frequency. To determine whether this unit is a point of contention in our PCLR system, a system with a faster unit is evaluated in this section. Thus, Figure 4.9 compares the previous system (*Hw*) with a system where the floating-point unit in the directory is clocked at the full frequency of the processor (*Fast*). It can be seen from the Figure that, although the execution times of some applications improve, the improvements are not significant. Therefore, even the relatively slow floating-point unit is not a bottleneck in this system.

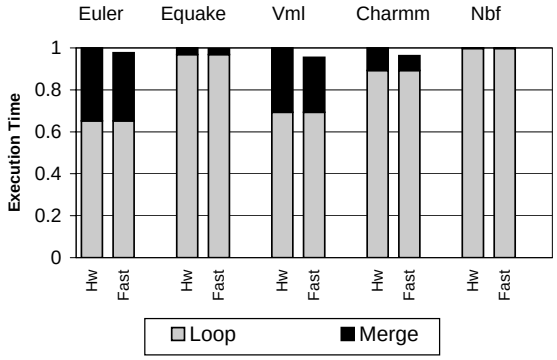

Figure 4.9. Comparing the performance with floating-point units of different frequencies.

4.6 Additional Use of PCLR

The PCLR scheme can also be used to speed-up another algorithm, namely the dynamic last value assignment. In this section, we explain the dynamic last value assignment problem (Section 4.6.1) and show how PCLR can be used (Section 4.6.2).

4.6.1 Dynamic Last Value Assignment in Software

As explained in Section 4.2, privatization is a common technique used to parallelize loops with anti and output dependences. When privatization is used, if the value of the privatized variable is needed after the parallel loop, a *last value assignment* has to be performed. Specifically, after the loop execution is complete, the shared counterpart of the privatized variable has to be updated with the value produced by the highest writing iteration. If the compiler can determine which iteration is that, it generates the code that puts this value in the shared variable. In the common case when all iterations write to the privatized variable, the last writing iteration is the last iteration of the loop. In this case, the compiler can simply peel off this last iteration and make it write directly to the shared variable.

However, when the last writing iteration cannot be determined at compile time, *dynamic last value assignment* has to be performed. In this case, the main parallel loop is followed by a *copy-out* phase. This phase identifies, for each element of the shared variable, the processor that ran the highest iteration that wrote to that element.

Figure 4.10 shows an example of a loop that is parallelized through the privatization of array A. It also needs dynamic last value assignment if the elements of array A are read after the loop, or the compiler cannot prove they are not read.

Figure 4.11 shows the parallel version of the loop, with the copy-out phase. The figure assumes that array A contains *NumElement* elements. As usual, static scheduling is used

```
1    for(i=0;i<N;i++)
2        if(f[i]){
3            A[g[i]]= . . .;
4            . . .=A[g[i]];
5        }
```

Figure 4.10. Loop to be parallelized with privatization and dynamic last value assignment.

for simplicity. As can be seen, together with the private array *A_priv[PID]*, each processor also has a private time-stamp array *A_ts[PID]*. Whenever a processor writes to an element of the private array, it also updates the corresponding element in the private time-stamp array with the number of the iteration it is executing. When the loop is done, the copy-out phase compares the private time-stamps of all the processors. Each element of the shared array is updated with the private copy of the processor where the maximum time-stamp is found.

Note that the private time-stamps *A_ts[PID]* have to be initialized to a number that is smaller that any possible iteration number (-1 in the example). Also, note that when static scheduling is used, as in the example, the processor ID can be used to update the time-stamps (*A_ts[PID][g[i]]* in line 6) instead of the iteration number, assuming that scheduling is done so that processors with increasing PIDs get iteration ranges with increasing indices.

4.6.2 Using PCLR for Last Value Assignment

Note that Figure 4.11 uses the private time-stamp arrays as reduction data, where the reduction operation is *maximum*. During the execution of the main loop, the updates to the private time-stamps compute the partial results, while the search for the maximum during the copy-out phase corresponds to the merge-out. Thus, the PCLR mechanism as explained in Section 4.3, can be used to speed-up the dynamic last value assignment operation. Figure 4.12 shows the code that results when PCLR support is applied to the code in Figure 4.11. Since PCLR performs reductions without declaring private reduction arrays, we only need to declare one shared time-stamp array (line 1). Its elements have to be initialized to a number lower than any possible iteration or processor ID (line 2).

During the execution of the main loop, the time-stamps are updated using the *maximum* operator. After the parallel loop is finished and the caches are flushed, each element of the shared time-stamp array will contain the maximum time-stamp for that element. Note that in the example, since static scheduling is used, the time-stamp array is updated with the PID of the process instead of the maximum (line 8). However, when these time-

```
// Initialize the private time-stamp array
1     for(i=0;i<NumElement;i++)
2         A_ts[PID][i]=-1;
// The range 0..N is split among the processors
3     for(i=MyNBegin;i<MyNEnd;i++)
4         if(f[i]){
5             A_priv[PID][g[i]]= . . .;
6             A_ts[PID][g[i]]=PID;
8             . . .=A_priv[PID][g[i]];
9         }
10    barrier();
// Copy-out. Range 0..NumElement is split among procs
11    for(i=MyNumElemBegin;i<MyNumElemEnd;i++){
12        x=-1;
13        for(p=0;p<NumProcessors;p++)
14            x=max(x,A_ts[p][i]);
15        if(x>-1)
16            A[i]=A_priv[x][i];
17    }
18    barrier();
```

Figure 4.11. Code resulting from parallelizing the loop in Figure 4.10 with dynamic last value assignment.

stamps are displaced or flushed from the cache, the shared counterpart in main memory will be updated with the maximum value, as the hardware directory controller was configured in line 4. Finally, each of these maxima will be used to identify the correct private version to copy into the corresponding element of the shared array (line 15).

Thus, using PCLR speeds-up dynamic last value assignment and makes it scalable with the number of processors.

4.6.3 Advanced Support

A further speedup could be obtained if the final *copy-out* phase was eliminated. To do so, we could extend PCLR with additional support. Currently, PCLR speeds-up the computation of the maximum time-stamp for each data element. However, we still have to explicitly perform the copy-out.

```
// Declare and initialize the shared time-stamp array
1     int A_ts[NumElement];
2     for(i=0;i<MyNumEl;i++)
3           A_ts[i]=-1;
// Configure the hardware
4     ConfigHardware(maximum_operation, int_type, A_ts,NumElement);
// The range 0..N is split among the processors
5     for(i=MyNBegin;i<MyNEnd;i++)
6           if(f[i]){
7                 A_priv[PID][g[i]]= . . .;
8                 A_ts[g[i]]=PID;
9                 . . .=A_priv[PID][g[i]];
10          }
11    CacheFlush();
12    barrier();
// Copy-out
13    for(i=MyNumElemBegin;i<MyNumElemEnd;i++){
14          if(A_ts[i]>-1)
15                A[i]=A_priv[A_ts[i]][i];
16    }
17    barrier();
```

Figure 4.12. Parallelized reduction code with dynamic last value assignment under PCLR.

With advanced support, caches and directory controllers could help eliminate the copy-out as follows. Every time that a line from the privatized array is displaced from the cache, the cache could also force the displacement of the corresponding time-stamps. When the displaced privatized line and time-stamps arrived at the home directory controller, a comparison would take place. An element in the privatized line would update the shared data in memory only if its time-stamp was higher than the one in shared-memory. At the end of the loop execution, a cache flush step would flush the remaining private array lines and time-stamps. Again, the home directory controller would only conditionally accept the incoming private data based on a time-stamp comparison. Overall, with this support, at the end of execution, the shared array would have the last values and no copy-out would have been necessary.

4.7 Related Work

Nearly all of the past work on reduction parallelization has been based on software-only transformations [EHLP91, YR00]. The most related architectural work that we are aware of is the work of Larus *et al.* [LRV94], Zhang *et al.* [ZPG+99], and the work on advanced synchronization mechanisms [BMW85, GVW89, GGK+83, KRS88, KDLS87, KDL+93, PBG+85, PN85, Smi81, ZY87].

Larus *et al.* briefly mention an idea similar to PCLR as one application of their Reconcilable Shared Memory (RSM) [LRV94]. RSM is a family of memory systems whose behaviour can be controlled by the compiler. They use RSM to support programming language constructs. The paper only mentions the applicability to reduction very briefly and provides no evaluation.

Zhang *et al.* propose a modified shared-memory architecture that combines both speculative parallelization and reduction optimization [ZPG+99]. In contrast to that work, which relies on a significantly modified multiprocessor architecture, we have presented relatively simple architectural support to optimize reduction parallelization. In addition, unlike in [ZPG+99], our scheme assumes that the compiler has already proved that our transformation is legal.

Finally, the combining support that we propose for the directory controller is related to the existing body of work on hardware support for synchronization and combining trees. Synchronizations mechanisms are relevant to reduction optimizations because, when implemented with critical sections, they can improve performance by allowing very efficient atomic Read-Modify-Write access to a shared variable. Such synchronization work includes the full/empty bit of the HEP multiprocessor [Smi81], Fetch&Add primitive of the NYU Ultracomputer [GGK+83], the Fetch&Op synchronization primitives of the IBM RP3 [BMW85, PBG+85], support for combining trees [KRS88, PN85], the memory-based synchronization primitives in Cedar [KDLS87, KDL+93, ZY87], and the set of synchronization primitives proposed by Goodman *et al.* [GVW89].

4.8 Summary

In this chapter, a new architectural support to speed-up parallel reductions in scalable shared-memory multiprocessors has been proposed. The support consists of architectural modifications that are mostly confined to the directory controllers. With this support, we eliminate the final merging phase that typically appears in conventional algorithms for parallel reduction. This phase takes time that is proportional to the data size in the

dense case, or to the data structure dimensions in the sparse case. With the proposed support, parallel reduction only needs a final cache flush step that takes time proportional only to the cache size. Overall, this scheme realizes truly scalable parallel reduction. While conventional software-only parallelization delivers average speedups of 2.7 for 16 processors, the proposed scheme delivers average speedups of 7.6.

References

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[BBO+83] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[BDE+96] 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. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 782–789, 1985.

[CHS+99] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture (HPCA'99)*, pages 70–79, January 1999.

[DGP+02] F. Dang, M.J. Garzarán, M. Prvulovic, A. Jula, H. Yu, N. Amato, L. Rauchwerger, and J.Torrellas. SmartApps, an Application Centric Approach to High Performance Computing: Compiler-Assisted Software and Hardware Support for Reduction Operations. In *Workshop on Performance Engineering Technology and Research Sponsored Under the NSF Next Generation Software Program*, February 2002.

[DMRV95] I. Duff, M. Marrone, G. Radiacti, and C. Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for Sparse Matrices. Technical Report RAL-TR-95-049, Rutherford Appleton Laboratory, 1995.

[DSH94] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[GB88] W. Gunsteren and H. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical Report, Laboratory of Physical Chemistry, University of Groningen, 1988.

[GGK+83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. In *IEEE Trans. on Computers*, pages 175–189, February 1983.

[GPZ+01] M.J. Garzarán, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pages 243–254, September 2001.

[GVW89] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. of the 3rd Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 64–73, April 1989.

[Hen00] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[HKD+94] M. Heinrich, J. Kuskin, D.Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operationg Systems (ASPLOS'94)*, pages 274–285, October 1994.

[KDL+93] D.J. Kuck, E.S. Davidson, D.H. Lawrie, A.H. Sameh, and C.Q. Zhu. The Cedar System and an Initial Performance Study. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 213–224, May 1993.

[KDLS87] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. *Experimental Parallel Computing Architectures: Volume 1 - Special Topics in Supercomputing*, chapter Parallel Supercomputing Today and the Cedar Approach, pages 1–23. J. J. Dongarra editor, North-Holland, New York, 1987.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[KRS88] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization on Multiprocessors with Shared Memory. *ACM Trans. on Programming Languages and Systems*, 10(4):579–601, October 1988.

[Kru86] C. Kruskal. Efficient Parallel Algorithms for Graph Problems. In *Proc. of the 1986 Int'l Conference on Parallel Processing (ICPP'86)*, pages 869–876, August 1986.

[KT98] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[LRV94] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 208–218, October 1994.

[PBG+85] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor

Prototype (RP3): Introduction and Architecture. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 764–771, 1985.

[PN85] G. Pfister and A. Norton. 'Hot Spot' Contention and Combining in Multistage Interconnection Networks. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 790–797, August 1985.

[Smi81] B.J. Smith. Architecture and Aplications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[VF94] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[YR00] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization. In *Proc. of the 2000 Int'l Conference on Supercomputing (ICS'00)*, May 2000.

[Zim91] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

[ZPG+99] Y. Zhang, M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas. A Framework for Speculative Parallelization in Distributed Shared-Memory Multiprocessors. Technical Report CSRD-1582, University of Illinois at Urbana-Champaign, July 1999.

[ZY87] C.Q. Zhu and P.C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. In *IEEE Trans. on Software Engineering*, pages 726–739, June 1987.

Chapter 5

Tradeoffs in State Buffering for Speculative Thread-Level Parallelization

Speculative thread-level parallelization aggressively runs hard-to-analyze codes in parallel. As speculative tasks run concurrently, they generate unsafe or speculative *memory state* that needs to be separately buffered and carefully managed. Such state may even contain multiple versions of the same variable. A multiprocessor must buffer and automatically manage this state despite its distributed caches, buffers, and memories.

This chapter presents a novel taxonomy of approaches to buffer and manage such state. The taxonomy includes a novel application of the concepts of architectural and future register state to memory state. We perform a tradeoff analysis and a detailed performance evaluation of the different approaches under a single DSM architectural framework. Our key insights are that support for buffering the state of multiple speculative tasks and versions in a processor is more cost-effective than support for merging the state of tasks with memory lazily. Moreover, both supports have orthogonal effectiveness and can be gainfully combined. Finally, a system with future state in main memory is more robust than one with architectural state, but more expensive to implement.

5.1 Introduction

Speculative thread-level parallelization attempts to extract parallelism from hard-to-analyze codes like those with pointers, indirectly-indexed structures, interprocedural dependences, or input-dependent patterns. The approach is to build tasks from the code and speculatively run them in parallel, hoping not to violate sequential seman-

tics. As tasks execute, special software or hardware supports check that no dependence across tasks is being eluded. If a infraction occurs, the state polluted by the offending tasks is repaired and the tasks are resumed. Many different schemes have been proposed, ranging from hardware-based [CMT00, FS96, FF01, GVSS98, HWO98, Kni86, KT99, MG99, OWP+01, PGRT01, SBV95, SCM97, SCZM00, THA+99, Zha99, ZRT99] to software-only [FLA01, GN98, RP95, RS00] schemes, and targeting small machines [FS96, GVSS98, HWO98, KT99, MG99, OWP+01, SBV95, THA+99] or scalable ones [CMT00, FF01, GN98, PGRT01, RP95, RS00, SCM97, SCZM00, Zha99, ZRT99].

As a speculative task executes, it generates speculative memory state. Such state is unsafe and, typically, is temporarily kept separated from the safe state of the memory system and the state of other speculative tasks. Moreover, in programs with dependences between tasks, different speculative tasks running concurrently may generate different versions of the same variable. In this case, reader tasks must have a way to identify the correct version, and the merging of versions with main memory state must be done in the correct order. Finally, the cache of a processor that has been running multiple tasks may end up holding speculative state from multiple tasks, and maybe even multiple versions of the same variable. Clearly, given the distribution of caches, buffers, and memories in shared-memory multiprocessors, buffering and automatically managing all this state in these machines is challenging.

Buffering schemes proposed in the literature manage this memory state in different ways. In some proposals, speculative tasks buffer unsafe state dynamically in *caches* [CMT00, FF01, GVSS98, KT99, OWP+01, SCZM00], *write buffers* [HWO98, THA+99] or *special buffers* [FS96, PGRT01] to avoid corrupting main memory. In other proposals, speculative tasks generate a *log of updates* that allows backtracking execution in case of an infraction [FLA01, GPL+01, Zha99, ZRT99]. Often, there are noticeable differences in the way caches, buffers, and logs are used in the different schemes.

Unfortunately, despite the popularization of speculative parallelization, there is no systematic classification and tradeoff analysis of possible buffering approaches. Understanding the involved design choices and issues is crucial, given the high fraction of performance typically lost in the memory hierarchy of multiprocessors.

This chapter addresses this issue and makes two contributions. First, it introduces a novel taxonomy of approaches to buffer and manage multi-version speculative memory state in multiprocessors. The taxonomy includes a novel application of the well-known concepts of architectural and future register state from Smith and Pleszkun [SP88] to memory state. On this taxonomy, we map the buffering schemes proposed in the literature.

The second contribution is a tradeoff analysis and a detailed performance evaluation

of the different buffering approaches under a single baseline architectural framework. Our key insights are that support for buffering the state of multiple speculative tasks and versions per processor is more cost-effective than support for merging the state of tasks with main memory lazily. Moreover, both supports have largely orthogonal effectiveness and can be gainfully combined. Finally, a buffering system with architectural state in main memory is not as robust as one with future state there. The latter handles the buffering of demanding applications better, although it is more expensive to implement.

This chapter is organized as follows: Section 5.2 introduces the challenges of buffering; Section 5.3 presents our taxonomy of approaches; Section 5.4 explains the main implementation issues: Section 5.5 performs a tradeoff analysis; Section 5.6 describes our evaluation methodology; Section 5.7 evaluates the different buffering approaches; and Section 5.8 concludes. Three appendices A, B, and C complement the chapter with background material, and extra explanations. These appendices can be skipped if the reader is familiar with thread-level speculation topics.

5.2 Buffering State Under Speculative Parallelization

5.2.1 Basics of Speculative Parallelization

Speculative thread-level parallelization extracts tasks from sequential codes and executes them in parallel hoping not to violate any sequential semantics. Under speculative thread-level parallelization, potentially dependent tasks execute in parallel. At run time, data references from a task may generate cross-task data dependences with references from other tasks. Thus, special software or hardware must ensure that those data references are handled properly and enforce the expected sequential semantics of the original code. At any time, tasks have a relative order imposed by the sequential code they come from. Consequently, we use the terms predecessor and successor tasks. If we give increasing IDs to successor tasks, the lowest-ID task still running is called the *non-speculative*, while the others are called *speculative*.

When a speculative task finishes, it cannot commit until all its predecessors have finished and committed. However, to better tolerate load imbalance, when a speculative task finishes, the processor that ran it can start to execute another speculative task. At any time, the system contains many speculative tasks, either finished or unfinished, and one unfinished non-speculative task. The set of all these tasks is called the *window of uncommitted tasks*.

If the task that finishes execution is the non-speculative one, it commits. Committing at least involves updating a variable called the *Commit Point* (*CP*), which identifies the

committed task with the highest ID, and making another task non-speculative. Figure 5.1 shows an example of several tasks running on four processors. In this example, when task T3 executing in processor 4 finishes the execution, it cannot commit. It has to wait until the predecessors tasks T2 and T3 also finish and commit. However, processor 4 can start to execute task T5. The example also shows how the non-speculative task status changes as tasks finish and commit (non-spec task timeline). In the example we assume that the commit is instantaneous.

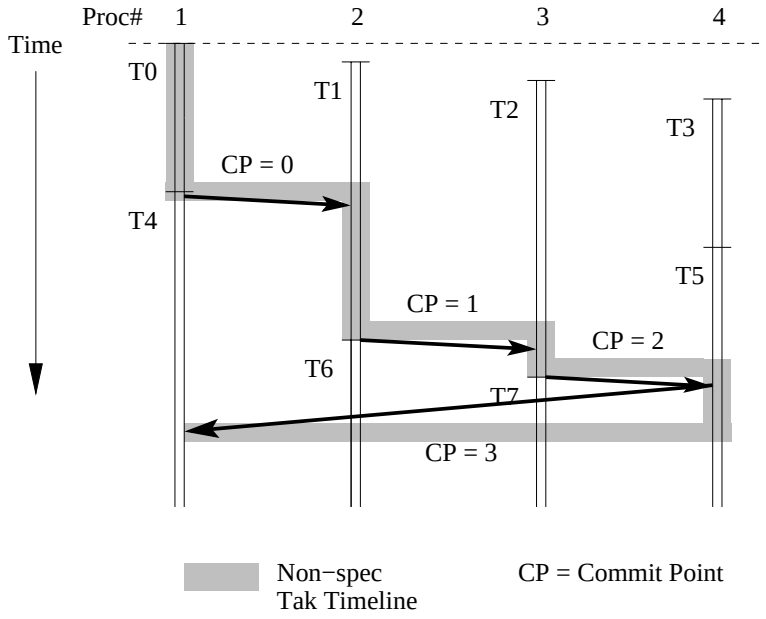

Figure 5.1. A set of tasks executing in four processors. The commit point (CP) advance is shown inside the non-speculative task timeline.

As speculative tasks execute in parallel, special software or a speculative coherence protocol checks for dependence infractions. Its function is to track memory references to identify any cross-task data dependence infractions. The possible data dependences are WAR (Write-After-Read), WAW (Write-After-Write) and RAW (Read-After-Write), and they can execute in-order or out-of-order. Typically, an error will occur if these dependences execute out-of-order. However, out-of-order WAR and WAW dependences can be handled at run-time and not induce errors in systems with support for multiple versions of the same datum. In these systems, as tasks execute they generate versions that are usually kept separate from the main memory, and tagged with the ID of the task that produced them. WAR and WAW dependences in these systems are handled by carefully merging the versions from the different tasks.

RAW dependences are more problematic. In an in-order RAW the speculation protocol must find the correct version and supply it to the reader task. However, in an out-of-order

RAW, a task reads a value that is subsequently modified by a predecessor task. A task has prematurely loaded a value. The speculation protocol must detect this error and the reader task needs to be squashed. Ordinarily, all the successors tasks are also squashed because they may have consumed versions produced by the squashed task. While it is possible to resolve reference chains and selectively squash only tasks at fault, it involves extra complexity. In any case, when a task is squashed, all its speculatively produced data must be purged. Then, tasks can resume the parallel execution.

Many different schemes have been proposed for speculative thread-level parallelization. They range from hardware-based to software-only. Some of these schemes work at the chip level, while others target larger Distributed Shared Memory machines. In hardware-based approaches, a special table records task memory references and checks for data dependences. For that, memory references that can cause dependences are marked with special codes of operation. When these specially marked load and store instructions execute, an extra coherence message is sent. This message must at least contain the type of operation, the ID of the task that executed the operation, and the referenced memory address. The message needs to reach the special table where all this information is recorded. In addition, the table is looked-up to find out if other tasks have accessed the same memory address before. If they have, the speculation protocol must check whether special actions must be undertaken. If for example the message was a write and the table has recorded that a successor task had previously read the same data (out-of-order RAW) the reader task and its successor ones need to be squashed. However, if the message was due to a read and the information in the table says that this data was previously written by another task (in-order RAW), the speculation protocol must find the correct version. In this case, the table can point-out to the processor that keeps the version produced by the writer task [CMT00], or it may send a message to all the sharers to find out which one has the correct version [PGRT01]. Finally, notice that for a large DSM machine, this table should be distributed and located close to the directory controller, while in the case of a small machine like an on-chip multiprocessor, it can be located between the private L1-caches and the shared L2-cache. The extra coherence transactions need to be designed such that they get correctly seriallized, as the baseline coherence messages get seriallized in the directory controller. This helps minimizing races in the protocol. As an example, Appendix A presents in detail the speculation protocol proposed in [Zha99]. It supports multiple versions an falls into the schemes named later in Section 5.3.3 as MultiT&MV lazy FMM schemes.

Next we describe the difficulties involved on buffering state under the previous conditions (Section 5.2.2), present some supporting application data (Section 5.2.3), and discuss an analogy to gain insight into how to improve buffering (Section 5.2.4).

5.2.2 Challenges in Buffering State

We identify three main difficulties in buffering state in a multiprocessor memory system.

Separation and Merging of Task State

Since a speculative task may be squashed, its updates are unsafe. Therefore, they are typically kept in caches [CMT00, FF01, GVSS98, KT99, OWP+01, SCZM00] or buffers [FS96, HWO98, PGRT01, THA+99], and not merged with the safe state of the memory system or the state of other tasks. Typically, when the task finally becomes non-speculative, its updates are merged with the safe memory state. Consequently, the multiprocessor must use its distributed caches, buffers, and memories to keep the state of the different speculative tasks separate, and merge them at the right times so that the system can always revert to a safe state.

Multiple Versions of the Same Variable in the System

A *new version* of a variable appears in the system when a task writes for the first time to that variable. Thus, when two speculative tasks running on different processors create two different versions of the same variable [CMT00, GVSS98], there is at least a WAW dependence between them. In addition to WAW dependences, WAR and RAW dependences may also exist between tasks that have created different versions of the same variable. For example, in Figure 5.2 a RAW dependence appears between task T1 and T2 when task T2 reads the variable x that previously was written by task T1. Since, later task T2 also writes to the variable x, T1 and T2 have two different versions of the same variable.

WAW dependences among tasks appear in applications that exhibit mostly-privatization patterns. Under such patterns, each task creates a new version of the variable without first reading an older version. Of course, this pattern should not be fully compiler-analyzable or the variable would have been privatized. Figure 5.2 shows an example of WAW dependences where tasks T0 - T3 write the same variable x.

Notice that to the rest of the system, a task appears as having at most a single version of any given variable at a time. Therefore, a task can write several times to the same variable, but the processor only needs to keep the last version produced by that task. The reason is that on a dependence infraction we undo the whole task, we never undo half a task. In the example in Figure 5.2, task T3 writes twice to the same variable. However, the processor only buffers the value written by the last write (4 in the example).

Buffering state under these conditions is hard. Specifically, versions from different processors must be merged with the safe memory state in correct task order. Furthermore, when a task executes a load involved in an in-order RAW dependence, obtaining the correct version is harder. The speculation protocol must find the correct version out of the several

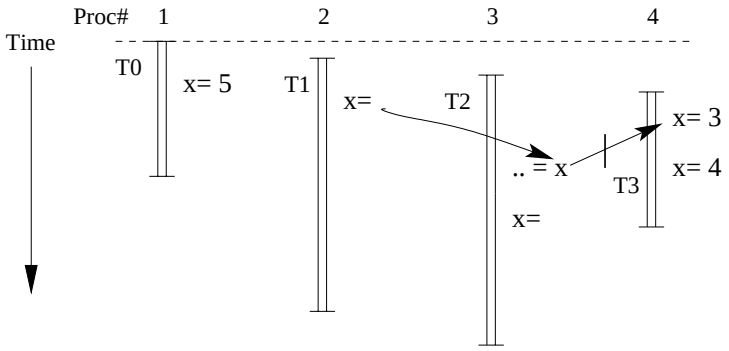

Figure 5.2. Multiple versions of the same variable in the system. Examples of in-order RAW, out-of-order WAR, and WAW dependences.

co-existing in the system and provide it to the reader task. An example of this problem appears in Figure 5.2 when task T2 reads the variable x.

Multiple Versions of the Same Variable in a Cache

When a processor finishes a task that cannot commit because it is still speculative, the processor may be forced to stall. Alternatively, and to better tolerate task load imbalance, it may be allowed to start a new task. In this case, the processor's cache may have to hold speculative versions from multiple tasks. Such versions are tagged with the owner task ID (Figure 5.3-(a)). Moreover, it is possible that the cache may even have to hold multiple versions of the same variable (Figure 5.3-(b)). This may occur in load-imbalanced applications that exhibit the mostly-privatization patterns discussed above.

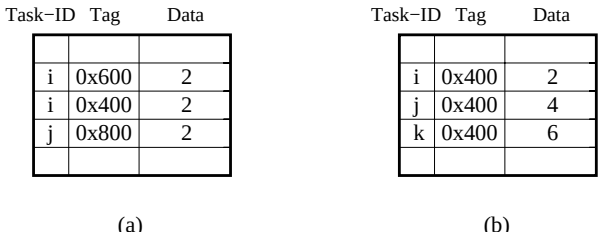

Figure 5.3. Example of a cache keeping versions from different tasks and different addresses (a) or different tasks and the same address (b).

Among all the speculative versions that a processor needs to buffer, we can distinguish the *last* version from the *non-last* versions. The *last* version is the one generated by the youngest task that ran (or is still running) in a processor and wrote the variable. It is the version that will be used to satisfy any subsequent load from the task that generated it, and possibly younger ones. All the previous overwritten versions are *non-last* versions. In the example in Figure 5.4, at time t the last version in processor 4 is the one with value 5

produced by task T5. Any subsequent load of this variable by task T5 will read this last version. A non-last version in processor 4 is the one with value 4.

Access to non-last versions should occur less frequently. They may occur when a task executes a read involved in an in-order RAW dependence, and the version produced by the writing task has been overwritten. For example, when task T4 in Figure 5.4 executes a read, the version supplied is the one produced by task T3, buffered in processor 4. This version is a non-last version. Note, however, that in-order RAW may equally be satisfied with last versions. Remember the example in Figure 5.2 where task 2 also executed a read involved in an in-order RAW dependence. In this case, the version provided was a last version.

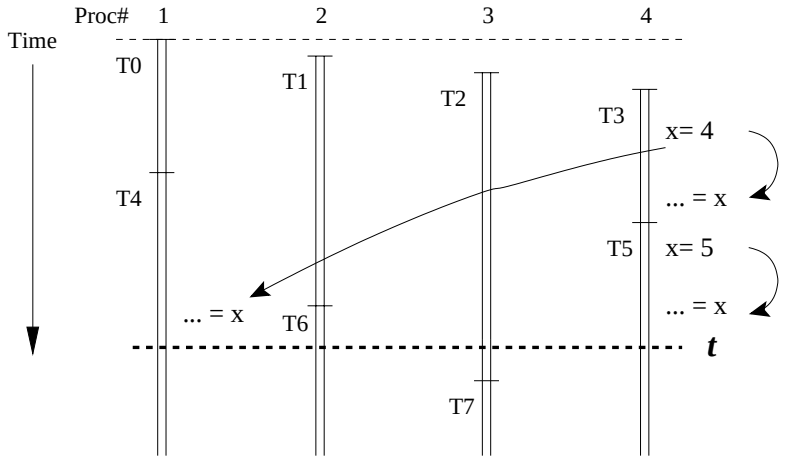

Figure 5.4. Multiple local speculative versions. Last versions and non-last versions.

Managing state under these conditions, where versions from multiple tasks can coexist in a processor, is even harder. On a read, several entries can match the address requested. Consider first a read from the processor. The cache will provide the data only if address tag and task-ID match with the requested address and ID of the requester task, respectively. On a external read to a cache, the cache provides the data and task ID of all the entries that match the address requested. The correct version is the one with the highest task ID that is still lower than the requester's task ID.

5.2.3 Application Behaviour

Table 5.1 illustrates the three difficulties described in Section 5.2.2. The applications (discussed in Section 5.6.2) execute speculatively parallelized loops on a simulated 16-processor CC-NUMA (discussed in Section 5.6.1). Column 2 in the table shows that,

in most applications, there are around 18-29 speculative tasks in the system at a time, although for very load-imbalanced loops as in *P3m*, the number is higher. Moreover, Column 3 shows that each processor may need to buffer the state of more than 1 speculative task at a time.

Application	Average # Spec Tasks		Average Written Footprint per Spec Task	
	In System	Per Proc	Total (KB)	Priv %
P3m	800.0	50.0	1.7	87.9
Tree	32.0	2.0	0.9	99.5
Bdna	25.6	1.6	23.7	99.4
Apsi	28.8	1.8	20.0	60.0
Track	20.8	1.3	2.3	0.6
Dsmc3d	17.6	1.1	0.8	0.5

Table 5.1. Application characteristics that illustrate the difficulties of buffering.

Figure 5.5 shows simplified loops taken from the applications in Table 5.1, where each task generates a new version of the same variable without first reading older versions. In these examples, the arrays *xdt()*, *work()*, and *stack()* are accessed with mostly-privatization patterns. For instance, in *Apsi*, each task creates its own *work(k)* elements without first reading any previous version of them. The elements of *work(f(i,j,j))* that a task reads without first writing are never written by a task. Thus, this loop is found to be fully parallel at run-time. However, since the compiler is unable to guarantee that there are no intersections between the values of k and the values of *f(i,j,k)*, this loop has to be speculatively executed in parallel. Similar reasonings could be given for the loops of the other applications.

In Table 5.1, columns 4 and 5 show the size of the written footprint per speculative task and the percentage of it that is accessed with mostly-privatization patterns, respectively. The percentage in the last column (*Priv*) has been obtained by measuring how much data generated by task *i-1* was also written by task i without first reading it. For *P3m*, *Tree*, *Bdna*, and *Apsi* this percentage is high. Thus, for these applications the buffer system must support the two multi-version issues of Section 5.2.2: multiple versions of the same variable in the system, and in case of imbalance (Column 3 of Table 5.1) multiple versions of the same variable in the same processor.

```
speculative_parallel do i
  do j = 1, i-1
    xdt(j) = ...
  enddo
  Compute L
  do j = 1, L
  enddo
enddo
```

(a) Bdna

```
speculative_parallel do i
  do j
    do k
      work(k) = work (f(i,j,k))
    enddo
    call foo(work(j))
enddo
```

(b) Apsi

```
speculative_parallel do i
  while (sptr > 0) do
    ... = stack(sptr)
    do j
      if (cond)
        stack(sptr) = ...
    enddo
  enddo
enddo
```

(c) Tree

Figure 5.5. Examples of non-analyzable loops that exhibit mostly-privatization access patterns. The outermost loop is speculatively parallelized.

5.2.4 Analogy to Register File Management

To better understand multi-version buffering for speculative parallelization, we propose to use the concepts proposed by Smith and Pleszkun for register file management in pipelined processors with precise exceptions [SP88]: architectural file, reorder buffer, future file, and history buffer.

The *Architectural File* (AF) refers to the *safe* contents of the register file. The AF is updated with the result of an instruction only when the instruction has completed and all previous instructions have already updated the AF. Exceptions do not affect the AF state.

The *Reorder Buffer* (ROB) allows instructions to execute speculatively without modifying the AF. The ROB keeps the register updates generated by instructions that have finished but are still speculative. When an instruction commits, its result is moved into the AF. If an exception occurs, some ROB entries may be invalidated.

The *Future File* (FF) refers to the *most recent* contents of the register file. A FF entry is updated by the youngest instruction in program order updating that register. The FF is used as the working file by later instructions. When an instruction commits, no copying is needed. However, the FF is unsafe because it has been updated by uncommitted instructions. Therefore, in an exception, we have to revert the FF to the AF.

The *History Buffer* (HB) allows the FF to be speculatively updated. The HB stores the *previous* contents of registers updated by speculative instructions. When an instruction commits, its HB entry can be freed up. In an exception, the HB is used to revert the FF to the AF.

We see an analogy between managing register file state for speculative instructions

and managing memory system state for speculative parallelization. Instruction completion, commit, and squash are now task completion, commit, and squash. Exceptions are now dependence violations. Finally, multiple instructions creating multiple versions of a register are now multiple tasks creating multiple versions of a variable.

5.3 Taxonomy of Approaches

5.3.1 Speculative State Composition

To optimize the buffering of multi-version speculative state, we examine its composition. Figure 5.6 shows a snapshot of the memory state of a program using the ROB or HB analogies.

Using the ROB analogy (Figure 5.6-(a)), we see the *architectural* memory system state of the program as being composed of unmodified variables (black region) and the committed versions of modified variables (white region). The remaining memory system state is comprised of speculative versions that have not yet committed. These versions form a *Memory-System Reorder Buffer (MROB)* distributed across the memory system. Among these versions, we distinguish between the *last* version of each variable (striped region) and the other, *non-last* speculative versions that have been overwritten (shaded region).

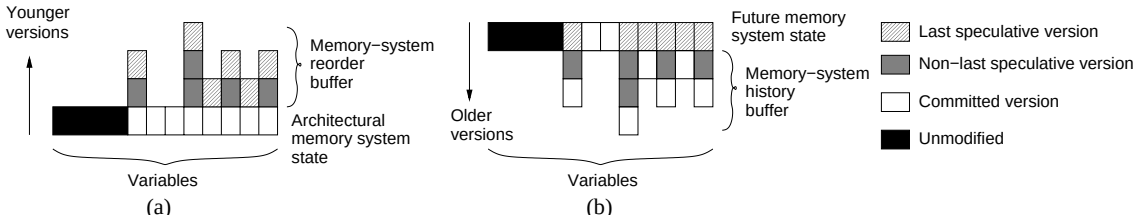

Figure 5.6. Snapshot of the memory system state of a program during speculative parallel execution. We use the ROB and HB analogies in (a) and (b), respectively.

Using the HB analogy (Figure 5.6-(b)), we see the *future* memory system state of the program as being composed of: unmodified variables, *last* speculative versions, and committed versions of modified variables that have no speculative version. The remaining *non-last* speculative versions and committed versions form a *Memory-System History Buffer (MHB)* distributed across the memory system.

Depending on how these versions are managed, we can classify the approaches to buffer multi-version speculative state. In the following, we present a novel taxonomy of approaches (Section 5.3.2), and map existing schemes to the taxonomy (Section 5.3.3).

5.3.2 Novel Taxonomy of Approaches

We propose two axes to classify the different approaches to buffer and manage speculative state in the memory system of multiprocessors (Figure 5.7-(a)). The first axis applies the concepts of Section 5.2.4 to main memory update and classifies the approaches based on whether main memory keeps the *architectural* (safe) state or the *future* (unsafe) state of the program.

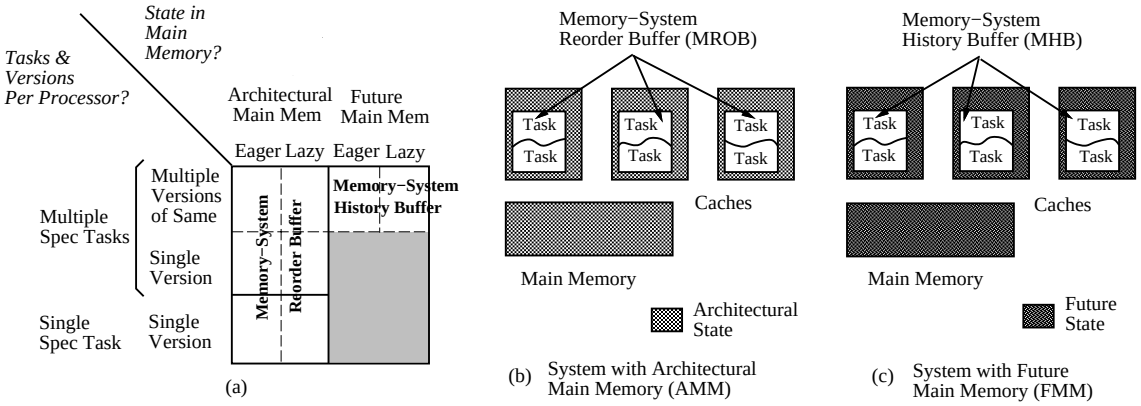

Figure 5.7. Different approaches to manage multi-version speculative state: taxonomy (a), keeping the architectural state in main memory (b), and keeping the future state in main memory (c).

In systems with Architectural Main Memory (AMM), all speculative versions remain in caches or buffers that are kept separate from the coherent main memory state. Only when a task becomes non-speculative can its updates in cache or buffer be merged with the main memory state. In this approach, caches or buffers become a distributed MROB (Figure 5.7-(b)).

In systems with Future Main Memory (FMM), versions from speculative tasks can be merged with the coherent main memory state. However, to enable recovery from task squashes, before a task generates a speculative version of a variable, the previous version of the variable is saved in the cache or a buffer. Such state is kept separate from the main memory state. In this approach, caches or buffers become a distributed MHB (Figure 5.7-(c)).

Note that in both AMM and FMM systems, part of the coherent main memory state can temporarily reside in caches (Figures 5.7-(b) and (c)). This is because caches also function in their traditional role as extensions of main memory.

AMM and FMM systems can each be further subdivided into *Eager* and *Lazy* depending on how fast eligible versions are merged with the coherent state of main memory. In eager schemes, a task eagerly merges all its versions with main memory when it commits

(in AMM systems) or eagerly merges every speculative version as it generates it (in FMM systems). Merging may involve a write back to memory [CMT00] or an ownership request to obtain coherence with main memory [SCZM00]. Lazy schemes, instead, merge lazily, usually on line replacement from the cache or on external request. As a result, several different committed versions of the same variable may temporarily co-exist in different locations. However, a variable only has a single committed version at any time. Therefore, the remaining previous versions can be thought as garbage versions whose future is certainly to disappear as soon as the space they occupy is needed [GVSS98].

The second axis in Figure 5.7-(a) classifies the approaches based on what type of speculative state can the private cache hierarchy of a processor hold at a time. One approach is to hold only the state of a single speculative task (*SingleT*) while the other is to support multiple speculative tasks (*MultiT*).

In SingleT systems, when a processor finishes a speculative task, it has to stall until the task becomes non-speculative and can commit. Only then can the processor start a new speculative task. In MultiT systems, when a processor finishes a speculative task, it can immediately start a new one.

One issue in MultiT systems is whether the private cache hierarchy of a processor is designed to hold multiple speculative versions of the *same* variable (*MultiT&MV*) or only a single one (*MultiT&SV*). If multiple, each of these versions necessarily belongs to a *different* task. This division is shown in Figure 5.7-(a). Under *MultiT&SV*, a processor stalls when a task is about to create a second local speculative version of a variable. Intuitively, *MultiT&MV* support is analogous to supporting multiple versions of the same logical register in the processor.

Finally, we shade an area in Figure 5.7-(a) to indicate that these schemes are relatively less interesting. We explain why in Section 5.4.

5.3.3 Mapping Proposed Schemes to the Taxonomy

Figure 5.8 maps existing schemes for speculative parallelization in multiprocessors onto our taxonomy.

1 AMM schemes

1.1 SingleT AMM schemes

Schemes such as Multiscalar with hierarchical ARB [FS96], Superthreaded [THA+99], MDT [KT99], Marcuello99 [MG99], SVC [GVSS98], and DDSM [FF01] belong to this category. These schemes are SingleT because before a processor starts to run a new speculative task, it waits until the task that it has just run has become non-speculative

and committed.

Out of all these schemes, Multiscalar with hierarchical ARB, Superthreaded, MDT, and Marcuello99 are eager because versions are merged with the main memory state eagerly at task commit. The buffering in these schemes differs on what part of the cache hierarchy holds the speculative state of a task: one stage in the global ARB of a hierarchical ARB in Multiscalar, the Memory Buffer in Superthreaded, the L1 in MDT, and the register file (plus a shared Multi-Value cache) in Marcuello99.

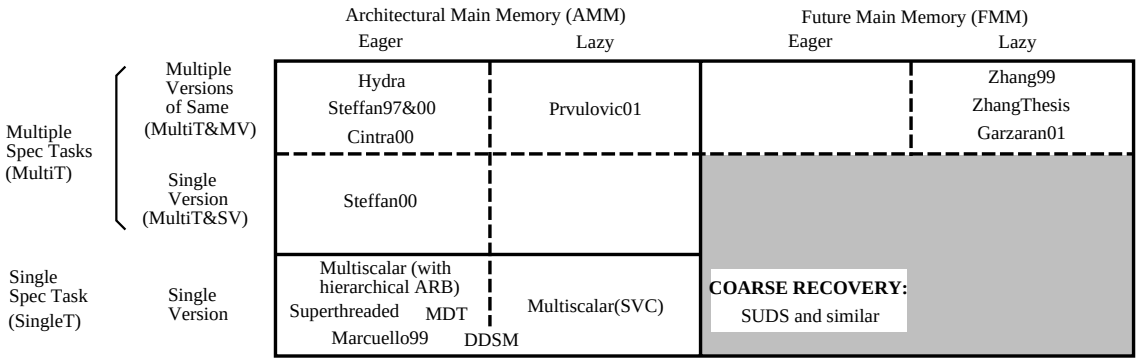

Figure 5.8. Mapping schemes for speculative parallelization in multiprocessors onto our taxonomy.

Multiscalar with SVC is lazy because committed versions linger in the cache after the owner task commits. Several different committed versions of the same variable can concurrently exist in different caches. They are only written back or invalidated when a new task somewhere accesses the variable. For example, on a load, a Version Ordering List (VOL) is used to find the last committed version, which is supplied to the requester. All the previous versions no need to be written back and are invalidated.

Finally, in DDSM, speculative versions are also kept in caches. However, work is partitioned so that each processor only executes a single task in the speculative section. For this reason, the distinction between eager and lazy does not apply.

1.2 MultiT AMM schemes

We start with the more aggressive **MultiT&MV** schemes: Hydra [HWO98], Steffan97&00 [SCM97, SCZM00], Cintra00 [CMT00], and Prvulovic01 [PGRT01]. Hydra stores speculative state in buffers between L1 and L2, while the others store it in L1, and possibly L2 and local memory. In all these schemes, a processor can start a new speculative task without waiting for the task that it has just run to become non-speculative[1].

[1]While this statement is true for Hydra in concept, the evaluation in [HWO98] assumes the hardware limitation of only as many buffers as processors, which makes the system SingleT.

The private cache or memory hierarchy of a processor may contain state from multiple speculative tasks, including multiple speculative versions of the same variable. This requires appropriate cache design in Steffan97&00, Cintra00, and Prvulovic01. In Hydra, the implementation is easier because the state of each speculative task goes to a different buffer. For example, a processor may be filling a buffer while the buffer that it filled with the previous task is not fully merged with memory. In this case, any two versions of the same variable will appear in different buffers.

In Steffan97&00, Cintra00 and Prvulovic01 versions in cache or overflow area are tagged with the owner task ID. In Steffan97&00 all these versions are in L1. In Cintra00, where each node of the Distributed Shared Memory Multiprocessor is an on-chip multiprocessor, speculative state produced by the task executing in each processor is hold in its L1 cache. Only when the task becomes the non-speculative task of the chip, can its speculative state be merged with the speculative state in L2 cache. Both, Steffan97&00 and Cintra00 use a victim cache to avoid or reduce stalls due to speculative state overflows.

In Prvulovic01, versions are kept in caches and overflow areas in the local memory of each processor. In absence of conflicts or capacity problems, two versions can be kept in L1 and L2. L1 always keeps a younger version (a last one), while L2 keeps an older one (a non-last version). This is an intelligent design choice since the oldest one (a non-last version) will never be required by the local processor. External requests can require last or non-last versions. However, the processor is already prepared to access L1 on an external request, since a normal coherent request of a data dirty in L1 also requires a L1 access. Note that this approach uses the inclusion property of the multi-level caches to keep two versions of the same variable (or cache line), which avoids cache pollution. Finally, an overflow area in the local memory of each processor is provided to buffer speculative versions in case of conflicts, capacity problems, or if more than two speculative versions need to be buffered. This overflow area is organized and accessed as a cache.

Of these schemes, Hydra, Steffan97&00, and Cintra00 eagerly merge versions with the coherent main memory state. Merging involves writing the versions to main memory in Hydra and Cintra00, or asking the directory for owner state in Steffan97&00. Prvulovic01 is lazy: committed versions remain in caches and are merged when they are displaced or when caches receive an external request. For lazy commit, they use a Version Combining Register (VCR) in the directory controller of each node.

One of the designs in Steffan00 [SCZM00] is **MultiT&SV**. Such a design is like Steffan97&00 except that the L1 is not designed to hold multiple speculative versions of the same variable. When a task is about to create a second local speculative version of a variable, it stalls. However, since the applications that they evaluate do not have

privatization pattern, this issues does not seem to affect their performance results. This schemes eagerly merges versions by asking the directory for ownership at commit time.

2 FMM schemes

2.1 SingleT FMM schemes

There is a class of schemes labeled *Coarse Recovery* in Figure 5.8. These schemes only support *coarse-grain* recovery. The MHB can only contain the state that existed before the speculative section. Such state is not overwritten with any other version throughout the section. In these schemes, if a violation occurs, the state reverts to the beginning of the entire speculative section.

These schemes typically use no hardware support for *buffering* beyond plain caches. They rely on software copies. They are *effectively* SingleT due to their coarse recovery, and are usually eager because it is the most straight-forward approach in software. Examples of such schemes are [FLA01, GN98, RP95, RS00, ZRT98][2].

The work in SUDS [FLA01] has two unusual characteristics. One is that no caches are used and, therefore, all versioning is done at the shared memory. All writes proceed directly to the shared memory. The second characteristic is that at run-time they build a MHB. Indeed, the MHB is a software array called log where each memory location can at most save one older version, namely the one existing before speculation started.

2.2 MultiT FMM schemes

All the schemes proposed, such as Zhang99 [ZRT99], ZhangThesis [Zha99], and Garzaran01 [GPL+01] are lazy. Their laziness comes from keeping in caches speculative versions that belong to the future state; they are merged with main memory only if they are displaced.

Zhang99 and ZhangThesis keep the MHB in hardware structures called logs. In ZhangThesis, a multi-version speculative protocol is able to handle all the dependences but the out-of-order RAW ones. Logs are kept in the local memory of each processor and are managed from a hardware controller. They are accessed in an in-order RAW dependence requiring a non-last version, or in an out-of-order RAW dependence to revert the system to a safe state. Logs can be recycled when the task that created it commits. This system also has a page-based overflow area in the local memory of each processor. This is explained with more detail in Appendix A.

In Zhang99, two different speculation protocols are proposed. The choice of the protocol will depend on the appearance of privatization access patterns in the executed application. For the applications without privatization access patterns, a Non-Privatization

[2]The work in [RS00] has some differences but can be shown to fall into this class.

protocol is proposed. In this Non-Privatization protocol, for each variable, its last version in the system is kept in main memory or cache of the last processor that wrote to it. All previous speculative versions of that variable in the system are kept in a hardware-managed log which is placed in the home memory of the variable. In this system, logs are only used to revert the system to a safe state in case of a dependence infraction. This system is very similar to SUDS01, with two main differences. First, in Zhang99 speculation is supported in hardware, and the logs can keep multiple versions of the same variable. Thus, Zhang99 does not need to synchronize after each processor executes a task. Second, the system in Zhang99 is a cache-based Distributed Shared Memory, while SUDS does not use caches.

In Garzaran01, the MHB is a set of software log structures, which can stay in caches or be displaced to memory. This system is based on the speculation protocol proposed in ZhangThesis. This scheme is a contribution of this thesis, and will be described in detail in the next chapter.

5.4 Implementation Issues

Before we examine the trade-offs associated with the different axes described, we discuss how to implement a Distributed MROB and MHB, and the main issues that appear. These issues will clarify why we shaded the single version area for FMM in the taxonomy.

5.4.1 Implementing a Distributed MROB

Consider a trivial example where every task creates a new version of variable X at address 0x400 without first reading an older version of it. Figure 5.9-(a) shows the code for 2 tasks that run on the same processor. Building a distributed MROB using cache or write buffer space is easy. Each task generates its version locally and keeps it in the cache or write buffer. Multiple speculative tasks running on a processor simply tag their versions with their task IDs and stack them up in the cache or write buffers. Versions will eventually be merged with the architectural state in main memory when the task that produced them commits. Figure 5.9-(b) shows an example where the MROB is built in the cache of the processor.

If the execution of a task causes the displacement of a speculative version, the task has to stall since main memory in AMM systems is designed only to contain architectural state. The stalled task will be able to resume when the task that has created the version that needs to be displaced becomes non-speculative. Since tasks commit in order, stopping a task may force its successors to remain stopped for a longer time. In this case, cache or

Task	writes	Value	to	Address
i		2		0x400
i + j		10		0x400

(a) Code

Cache

Task ID	Tag	Data
i	0x400	2
i + j	0x400	10

(b) MROB in a processor

Figure 5.9. Implementing a distributed MROB.

speculative buffers accumulate more speculative state, which can cause further stalls. In systems with many processors, these stalls may be a serious bottleneck.

In order to avoid task stalls we need to have an unlimited-sized area where the cache can safely overflow into. The design of such an overflow area will largely depend on the requirements of the underlying speculation protocol. A possible design is the one proposed by Prvulovic01 [PGRT01]. In this case, the overflow area is organized and accessed as a cache, but uses the node main memory as a storage media. Thus, it can be considered as a slow unlimited-size cache, where versions also need to be tagged with task-IDs. This overflow area is used only in case of overflow of the speculative state in cache.

An alternative solution to reduce the amount of speculative state in caches, and thus reduce the possibility of task stalls, is to remap non-last versions. This solution works as follows. When a processor generates a new version of a variable, the cache line that contains the older version is remapped or copied to a structure at a new address. The remapped non-last versions can be displaced from the cache because such write-backs do not affect the versions in the architectural state. As before, these remapped versions can be organized as a cache. They can also be organized by grouping records according to the ID of the task that produced the version. Figure 5.10-(b) shows an example where non-last versions are remapped, when tasks *i*, *j*, and *k* execute the code in Figure 5.10-(a). In the example, versions are contiguously placed in memory, and grouped under the *Producer Task-ID*. Note that this approach by itself does not solve the speculative state overflow problem since last versions are still kept in cache, and if one of these last versions needs to be displaced, the processor has to stall. This is the approach used in [GPL+02]

5.4.2 Implementing a Distributed MHB

A distributed MHB should ideally work as follows. Task *i+1* saves task *i*'s version of *X* in its MHB and makes its own version of *X* visible to the rest of the memory system. Then, task *i+2*, possibly running on another processor, saves *i+1's* version in its own MHB and makes its own version visible to the memory system. Unfortunately, this approach

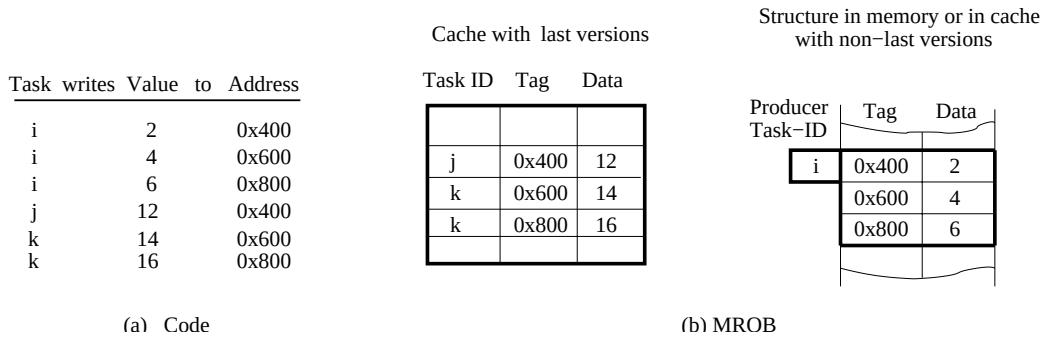

Figure 5.10. Data structure organization where non-last versions are remapped. Speculative tasks i, j, and k execute the events in (a) while running on the same processor and using a reorder buffer approach (b) where cache keeps the last versions, and the non-last versions are remapped in a data structure in cache or memory. When task i commits, the versions it produced will be sent to main memory and the data structure space released.

requires much communication between processors. Furthermore, since a task must save its immediate predecessor's version, this scheme disables concurrency, even in the absence of true dependences as in the example.

To mitigate this problem, the MHB is made to work differently. Specifically, when a task is about to generate a new version, the task saves in its MHB the most recent version of the variable generated by a previous *local* task. With this approach, when there are no true dependences as in the example, processors do not communicate, tasks run concurrently, and the distributed MHB works in parallel. Of course, true dependences require communication between processors.

The resulting MHB in a node is shown in Figure 5.11-(b) when tasks execute the same code than before (Figure 5.11-(a)). The MHB itself is a structure in memory or in the cache. A consequence of this MHB implementation is that the MHB entry created by a task does not store the most recent version that globally precedes that task. Thus, each MHB entry must be tagged with the ID of the task that generated the version (task ID i in the figure). To see why, consider the following example. Assume that two speculative tasks p and $p+q$ running on different processors have each pushed into their MHB two different committed versions of the same variable. Since each task pushes *local* versions, the version pushed by $p+q$ may be younger or older than the one pushed by p. Thus, we need to tag versions with task ID in order to maintain a total order of all the versions of a variable in the distributed buffers. This information will be needed in case a infraction occurs and recovery is necessary.

Unfortunately, such per-entry tagging is needed even in SingleT schemes. This is unlike

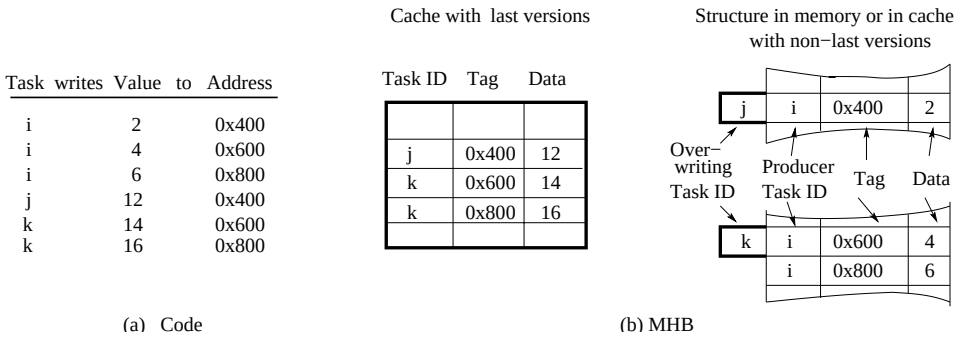

Figure 5.11. Distributed MHB when tasks i, j, and k execute in the same node.

in the MROB, where SingleT schemes only need a single, per-processor task ID. Therefore, SingleT FMM schemes have nearly as much hardware as MultiT FMM schemes without the latter's performance advantages (as we will see in Section 5.7). A similar argument can be made for MultiT&SV FMM relative to MultiT&MV FMM. For this reason, we claim that the shaded area in Figure 5.7-(a) is uninteresting (except for coarse recovery).

Finally, note that in the MHB versions are also grouped under the the *Overwriting* task-ID (task-ID i and k in the example). As in a register HB, when the overwriting task commits, its overwritten versions become obsolete and can be deallocated. Thus, grouping together all the versions that a task overwrote is very useful, because they all can be deallocated at the same time once the task has committed. Note that this MHB implementation is very similar to the previous MROB one, where non-last versions are remapped. The difference comes from the different organization of the remapped versions.

Another implementation issue that appears in FMM systems comes from the fact that versions of a given variable can be displaced from different caches to main memory out of task-ID order. Main memory always keeps the most advanced future state possible. As a result, it must reject write-backs of older versions that try to overwrite younger versions of variables already present in main memory. To do so, FMM systems associate a task-ID tag with each variable under speculation present in main memory. This tag identifies the version currently in memory. In addition, main memory needs support to compare the task-IDs of incoming and existing versions.

Finally, another source of complexity in FMM systems appears if speculative state can overflow caches. As we have just explained, in an FMM system, any version can be evicted to main memory, which always will grab the youngest evicted one. However, speculative systems that only cause squashes under out-of-order RAW dependences must always keep all speculative versions of a variable (last and non-last ones), and the committed version. These versions can be needed in an in-order RAW dependence, to supply the correct

version. In addition, the committed version of a variable will be needed on an out-of-order RAW, to revert the system to a safe state. Keeping all this state in a FMM system needs additional support to the one already described. Suppose that a task wants to evict the speculative version i, and the home memory already has a younger version j. Since version j is younger than version i, the home memory will reject version i. However, by doing so we can loose the speculative version i. The processor has evicted it, and since version i was not overwritten (it is a last version), it did not save a copy in the MHB. The main memory does not keep it because it is not the youngest version. Next, we explain two different solutions.

One solution to this problem was proposed in ZhangThesis [Zha99], and consists of a page-based *overflow area*, where versions in the cache can flow into. Under this scheme, the first time a node displaces a line of a cache that has been speculatively modified, a page is placed in the local memory node and the displaced line is saved in it. This overflow page is allocated only if at least one cache line of a shared page has been displaced in dirty state from a local cache. Only the dirty lines need to be valid in the overflow page. This mechanism guarantees that we are always able to revert the system to a safe state, and all the speculative and the committed versions are kept. This solution requires an additional per-node Address Mapping Module that is somewhat similar to the one in Prism [ELPS98], WildFire [HK99], and S3.MP [NAB+95]. See Appendix A for a more detailed explanation.

An alternative approach to the overflow area would be to build the MHB in the home memory. With this approach, the memory will keep the youngest displaced version, as before. The MHB will keep older displaced versions. The MHB can keep the last commit version plus all the displaced speculative versions that are older than the one in memory. However, no proposal has addressed this alternative, probably due to the high traffic it may require.

5.4.3 Operations To Support in a Distributed MROB and MHB

All buffering schemes must support five important operations, namely *Accesses to Own Versions*, *Accesses to Non-Last Versions*, *Version Commit*, *Version Recovery*, and *Obsolete Version Elimination*.

Accesses to the Versions of the Current Task are satisfied from the cache or buffer. Identifying such versions is easy: their task ID should match the ID of the requester. Since such accesses are frequent, they should be satisfied efficiently. Note that these local versions are always the last ones.

Accesses to Versions Produced by a Task Previous to the Current One are not intercepted by the cache. They need to reach a directory or similar structure that can point to the correct version. This version can be found in the cache or buffer, overflow area, MHB, or MROB. Identifying such correct version may involve non-trivial comparisons involving several task ID tags. In most cases, a processor only really executes these accesses in RAW dependences, which are hopefully not very frequent.

Version commit is the merging of the version with the architectural state of the program. Versions commit when the task that generated them commits. Committing may involve explicit data movement or state change like in eager AMM systems. Since version commit is a frequent operation, it should be efficiently supported.

Version recovery is needed to revert the memory system back to a correct state after a dependence infraction (an out-of order RAW). Recovery may involve discarding versions from the MROB or copying back versions from the MHB. Hopefully, recovery is not very frequent.

Obsolete version elimination involves expunging from the system committed versions that have become useless. They are useless because other copies exist in the system or because a different, younger version has also committed. It is hardly a time-critical operation and can usually be done lazily or in the background.

5.5 Tradeoff Analysis

We analyze the impact of the three degrees of freedom in Figure 5.7-(a) in the following subsections: Architectural vs Future Main Memory System, Single vs Multiple Speculative Tasks & Versions per Processor, and Eager vs Lazy Merging with Main Memory State.

5.5.1 Architectural vs Future Main Memory

The tradeoff is that FMM systems handles version commit better, while AMM systems make version recovery easier. Moreover, FMM systems have a higher implementation cost.

Version Commit and Recovery

In FMM systems, *Version commit* does not require any special operation. When a task generates a new version, the older version is saved in the MHB and the new one takes its place. The new version can be written back to main memory either eagerly or lazily. The older versions can have been remapped to the MHB and, therefore, they cannot overwrite memory. In either case, when the task commits, the version simply commits.

Consider now an AMM system. Versions must be kept buffered in the MROB until the task that produced them becomes non-speculative to prevent overwriting the architectural state in main memory. Versions commit after the task that generated them commits. Version commit requires an additional access to each one of the versions generated by the task. This access takes place at task commit time in eager protocols, or it can be delayed in lazy protocols. Since cache access is fast, committing versions in cache should also be fast. However, committing versions in an overflow area in the local memory may be slower. First, we need to find the version, and then access it.

On the other hand, *Version Recovery* is more natural under AMM. Recovery under AMM simply involves discarding from the distributed MROB the speculative versions generated by the offending task and successors. In contrast, recovery under FMM involves copying all the versions over-written by the offending task and successors from the distributed MHB to main memory. During this recovery procedure, different versions of the same variable may be sent to the home. The home must be selective, and choose the version with the largest task-ID which is smaller than the ID of the Commit Point. This process may also require selective access to the overflow areas, where versions not present anywhere else can be kept. Note that bringing the distributed memory system of a multiprocessor to a consistent state is harder under FMM because main memory may have a lot of non-architectural state.

Version commit and version recovery are the two operations that are intrinsically different in AMM and FMM schemes. Next we will discuss about the other operations.

Obsolete version elimination

It is equally cheap in AMM or FMM schemes. In AMM schemes, the entries in the MROB are eliminated when the corresponding task commits and its versions are merged with memory. In FMM schemes, the entries in the MHB become obsolete when the task that produced them (the overwriting task) commits. Grouping all these entries together allows to deallocate all of them at a time. These version can be freed either immediately or lazily at regular intervals.

Accesses to the Versions of the Current Task

It is similar in both cases. In the case of AMM systems, the cache can have multiple versions of the same variable tagged with a different task-ID; while in the FMM systems, the cache only has the last version. In any case, under a request from the processor, the ID of the requester task and the task-ID in cache have to match.

Accesses to Versions Produced by a Task Previous to the Current One

This type of access is required in an in-order RAW dependence. In AMM schemes, all

the speculative versions (last and non-last ones) are kept in the MROB, and it has to be accessed. If the MROB, is only built in the cache, this access is relatively fast. However, if the MROB is also in the overflow area in the local memory, its access will be slower. Since the request asks for an specific address, a cache-organized overflow area will speed up this access. In a FMM system, last versions are in the cache and overflow areas, while the non-last versions are in the MHB, which has to be searched. If data in the MHB are not organized by address, access to a particular version requires a serial traversal and may be slow. Fortunately, in practice non-last versions are rarely accessed.

Finally, recall from Section 5.4.1 that we propose an approach to build a MROB where non-last versions in the cache are remapped to a different address, like in the MHB. If non-last versions are remapped, the cache only needs to keep last versions, which are the only ones that will be requested by the local processor. In addition, removing non-last versions from cache may have an advantage, since the effective capacity of the cache increases, which may increase the hit ratio. While this remapping or copying could be expensive if implemented in hardware, it can also be done inexpensively in software. The disadvantages of this approach, is that on an in-order RAW access, the data structure where non-last versions are remapped may need to be accessed. At version commit, AMM systems also need to access this data structure.

Implementation Complexity

FMM systems are more complex than AMM systems. One source of complexity of FMM systems comes from the need of tagging main memory with the task-ID to allow out of task-ID order displacement.

Such support in main memory is unnecessary in AMM systems because version merging with main memory state can always be forced to occur in task order. Indeed, in eager systems (Section 5.3.2), versions are trivially combined with main memory state in order as they commit. In lazy systems, a scheme as proposed in [PGRT01] can be used to write back versions in order.

Another source of complexity under FMM systems comes from the difficulties of building a distributed MHB. One solution to this problem that keeps traffic low, requires an overflow area in the local memory of the processor where displaced versions can be saved. An alternative solution is to build the MHB in the home memory node. The main drawback of this alternative is that the traffic may increase significantly. No solution has been provided so far for FMM systems with limited capacity to keep the speculative state. The study of such solutions that would probably require task stalls and squashes are out of the scope of this thesis. On the other hand, AMM systems also need an overflow area to prevent tasks from stalling if speculative state overflow occurs. In AMM systems without

an overflow area, tasks have to stall until the task that produced the version that needs to be displaced commits.

There are other complexities in FMM and AMM systems. In FMM systems, when a new version is created, the old one has to be copied to the MHB. In reality, while special hardware for copying could be provided, low-overhead copying can be done completely in software (next Chapter). In AMM systems, a cache may contain several versions with the same address tag. Thus, to find the correct version we require some serial comparisons that may increase cache occupancy, or more hardware for parallel comparisons.

5.5.2 Single vs Multiple Speculative Tasks & Versions per Processor

The tradeoff is higher performance and higher implementation cost as we go from single-task (SingleT) to multiple-task (MultiT) schemes and, within the latter, from single-version (MultiT&SV) to multiple-version support (MultiT&MV). MultiT schemes are typically faster in the presence of task load imbalance. The exception is MultiT&SV which may loose performance with load imbalance if mostly-privatization patterns are dominant.

Task Load Imbalance and Mostly-Privatization Patterns

SingleT schemes perform poorly if tasks have load imbalance. In this case, a processor that has completed a short speculative task may have to wait for the completion of a long, non-speculative task running elsewhere. Only when the short task becomes non-speculative and, typically, commits can the processor start a new task. For example, consider Figure 5.12-(a), where T_i and c_i represent task execution and commit, respectively. The figure corresponds to an eager AMM system. In the figure, processor *1* completes task *T1* and waits. When it receives the commit token, it commits *T1* and can start *T3*. Summarizing, in SingleT schemes, version commit needs to be done before a new task can start to execute. This implies that version commit cannot take place in the background while another task is executing.

MultiT schemes can better tolerate load imbalance because processors that complete a speculative task can immediately start a new one. However, MultiT&SV schemes can run slowly if tasks have load imbalance and create multiple versions per variable. The latter occurs, for example, under mostly-privatization patterns (Section 5.2.2).

Under such circumstances, a processor executing a second speculative task stalls when the task is about to create a second *local* speculative version of a variable. The processor remains stalled until the first task becomes non-speculative and, as a result, the first version of the variable can merge with memory. For example, in Figure 5.12-(b) processor

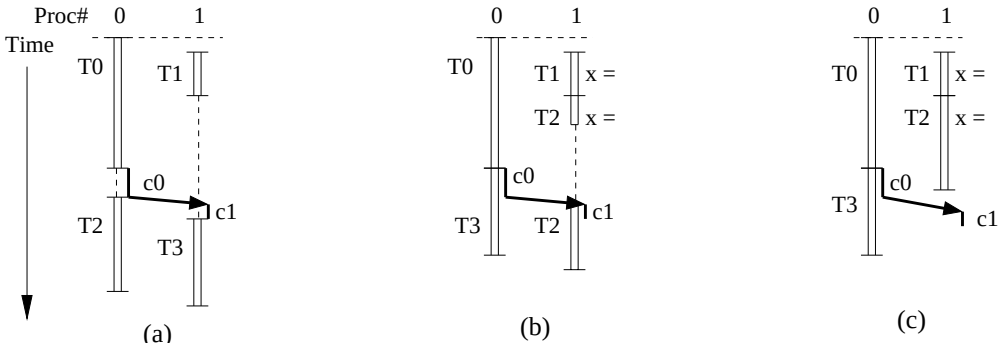

Figure 5.12. Example of four tasks executing under SingleT (a), MultiT&SV (b), and MultiT&MV (c).

1 generates a version of X in task *T1*. Then, it executes task *T2* and stalls when trying to generate a new version of X. When processor *1* receives the commit token for *T1*, the first version of X can be displaced from the cache and *T2* can restart. Notice that in the example version commit happens in the background, since caches are prepared to keep multiple versions from different tasks. Thus, as soon as a task receives the commit token, a new task can start to execute in that processor.

All these stalls are eliminated with MultiT&MV schemes (Figure 5.12-(c)).

Implementation Complexity

In SingleT schemes, the private cache hierarchy of a processor is simple. Indeed, since it can only hold state from a single speculative task, there is no need to tag each line or variable with a task ID. It is enough for the processor to know what task it is executing.

In MultiT schemes, the private cache hierarchy of a processor is more expensive to build and, in addition, requires additional comparisons to access. Indeed, it can hold versions from the multiple speculative tasks that the processor is (or has been) executing. As a result, each line or variable is tagged with a task ID. Furthermore, when a cache is accessed, the address tag and task ID of the chosen entry are compared to the requested address and the ID of the requester task, respectively. An access from the local processor hits only if both comparisons succeed. In an access from a remote processor, the action depends on whether the scheme is MultiT&SV or MultiT&MV.

In a MultiT&SV scheme, the cache hierarchy can only keep a single version of a given variable. Therefore, an external access to a cache can trigger at most one address match. In this case, the relative value of the IDs tells if the external access is out of order. If it is, a squash may be required. Otherwise, the data may be safely returned.

In a MultiT&MV scheme, the cache hierarchy of a processor can keep multiple versions of the same variable. This complicates both the design and the processing of external

accesses. Specifically, a cache must support multiple entries with the same address tag and different task ID. Some proposed designs put such entries in different lines of the same set [CMT00, SCM97]. In this case, an access to the cache retrieves all the lines in a set. Consider the case of an external read. The cache controller has to identify which of the matching entries has the highest task ID that is still lower than the requester's task ID. That one is the correct version to return. This operation may require some serial comparisons that may increase cache occupancy. Alternatively, it may need more hardware for parallel comparisons.

5.5.3 Eager vs Lazy Merging with Main Memory State

The tradeoff is the higher performance and implementation complexity of lazy schemes. Eager schemes are slower because they slow down the commit or the execution wavefront.

Commit and Execution Wavefronts

Program execution under speculative parallelization involves the concurrent advance of two wavefronts: one for task execution and one for task commit. Figure 5.13 shows the wavefronts for a lazy scheme. The *Execution Wavefront* advances as processors execute tasks in parallel (T_i); the *Commit Wavefront* advances as tasks commit in strict sequence (c_i) by passing the commit token. In a lazy scheme, at the end of the speculative section all the versions remaining in the caches are written back and merged with main memory [PGRT01]. This is shown with bolded lines in the figure.

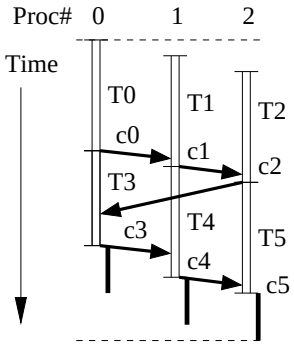

Figure 5.13. Progress of the execution and commit wavefronts for a lazy AMM/FMM system.

Eager schemes eagerly merge versions during execution. This eliminates the global write-back at the end of execution. However, it slows down one of the two wavefronts: the commit wavefront in AMM systems and the execution one in FMM systems. The result may be a slower program.

Consider AMM schemes first. Eager schemes require explicit data write-

backs [CMT00] or ownership transfers [SCZM00] before the commit token can be passed from the non-speculative task to its successor. These operations slow down the commit wavefront. If the commit wavefront appears in the critical path of program execution, the slowdown will be visible to the program, becoming a scalability bottleneck [PGRT01].

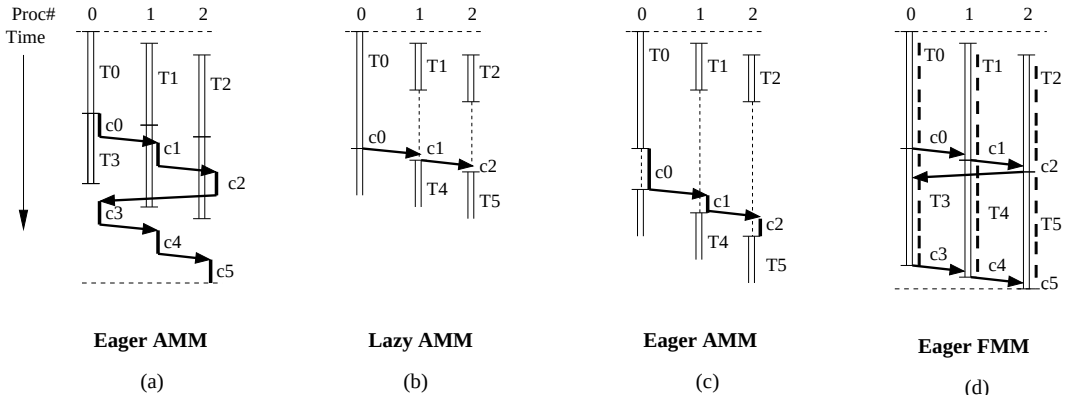

Figure 5.14. Progress of the execution and commit wavefronts in different schemes: eager AMM (a), lazy AMM with stalls (b), eager AMM with stalls (c), and eager FMM (d).

There are two cases when the commit wavefront is in the critical path. In one case, it appears at the end of the speculative section (Figure 5.13). To understand this case, we introduce the *Task Commit Ratio* as the ratio between the average duration of a task commit and a task execution. For a given machine, this ratio is a program characteristic that measures how much state the program generates per execution unit. The commit wavefront appears at the end of the speculative section if the task commit ratio of the program times the number of processors is higher than 1.

The second case when the commit wavefront is in the critical path can appear when there is load imbalance in MultiT&SV or SingleT systems. In these schemes, a processor may have to stall until it receives commit token due to their single version or single task limitations. Figure 5.14-(b) shows one example for a lazy AMM scheme. Clearly, if the scheme is eager as shown in Figure 5.14-(c), the commit wavefront affects the critical path even more.

Now consider an eager FMM scheme. Figure 5.14-(d) shows such a scheme, which is the FMM equivalent to the AMM scheme of Figure 5.13. In such a scheme, a task merges every speculative version that it generates with the coherent state of main memory as soon as it creates it. Again, this can be done through data write-backs (effectively creating a write-through cache) or ownership transfers. Note that a task cannot complete until all its transactions complete. Consequently, the resulting extra write-back or ownership transactions (shown in dashes in Figure 5.14-(d)) slow down the completion of the task. As a result, the progress of the execution wavefront slows down.

Finally, note that, to a lesser extent, eager schemes also slow down the execution wavefront in AMM schemes and the commit wavefront in FMM schemes. The reason is that they induce higher traffic than lazy schemes. Specifically, lazy schemes minimize traffic by exploiting version obsolescence. They lazily leave committed versions in caches. Younger tasks may eventually create younger, committed versions of these variables. In this case, if the old versions have not yet been displaced from the caches, they have become obsolete. They do not need to be merged with the main memory state and can simply be silently discarded without creating traffic.

Implementation Complexity

Two effects make lazy schemes more complex. The first one is the need to ensure that versions of the same variable are merged into the main memory state in task order. Failure to do so could result in version loss. In-order merging is supported by construction only in eager AMM systems. Lazy schemes and even eager FMM schemes need special support. Specifically, lazy and eager FMM systems associate a task-ID tag with each variable under speculation in main memory and use a comparator to discard incoming older versions (Section 5.5.1).

Lazy AMM systems may use the same support or a simpler, yet non-trivial support. For example, Prvulovic01 [PGRT01] and Multiscalar with SVC [GVSS98] extend the write-back transaction of a committed version with the invalidation of all previous versions. Since Prvulovic01 uses multi-word cache lines, the transaction also collects committed versions from the other words in the line.

The second source of complexity in lazy schemes is that we may need to search in order to find the latest architectural (in AMM) or the latest future (in FMM) version of a variable. For example, several different committed versions of the same variable can co-exist. To find the latest version, Multiscalar with SVC uses the VOL Linked List [GVSS98] while Prvulovic01 [PGRT01], Zhang99 [ZRT99], ZhangThesis [Zha99], and Garzaran01 [GPL+01] tag versions with a task ID.

In eager schemes, all the versions that a task produced need to be accessed at commit time. If no support is provided caches and overflow areas need to be traversed. To reduce the commit time, a table in each level of the memory hierarchy can be used. This table keeps the list of addresses that each task must commit. Notice that grouping the versions that a task produced in write-buffers or remapping non-last versions in consecutive positions in memory may also help to reduce version commit.

Effect of Application Characteristics

Based on our discussion, Figure 5.15 shows the main application characteristics that limit performance in each case. Of course, the simpler the mechanism, the higher the number of factors that will limit performance.

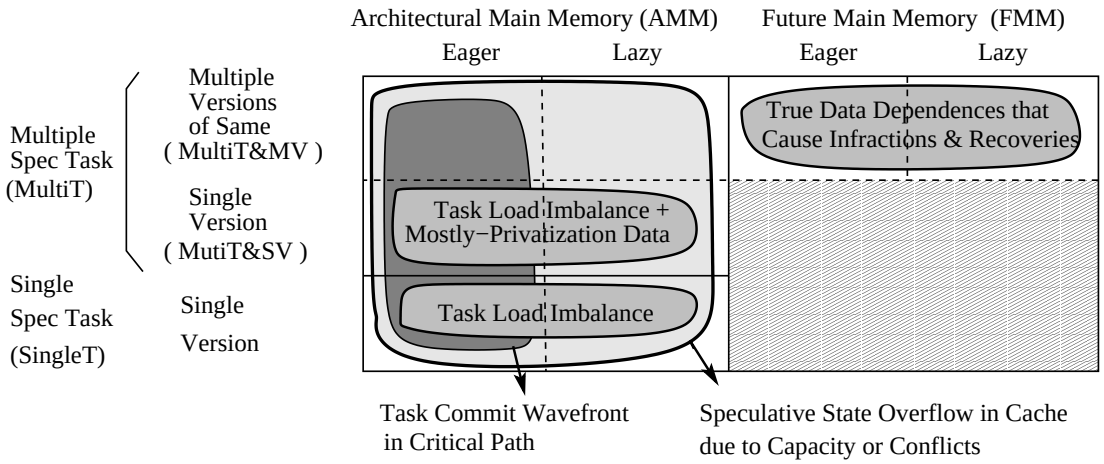

Figure 5.15. Application characteristics that limit performance in each approach.

If we compare AMM systems vs FMM systems, true data dependences (RAW) is the main application characteristic that can result in lower performance of a FMM system when compared to an AMM system. The reason is that if RAW dependences execute out-of-order, they cause task squashes and recoveries, which are more expensive under FMM systems. However, RAW dependences should be rare for the applications executing under speculative thread-level parallelization. The main reason is that RAW dependences between nearby tasks could easily execute out-of-order. If this was the case, the resulting squash overhead would seriously hurt performance in both AMM and FMM systems. In addition, even if RAW dependences executed in-order, they would induce much communication across tasks. Since AMM and FMM are distributed systems, the time needed to satisfy read misses will be high. Thus, we expect that the applications executing under speculative thread level parallelism will have a high degree of parallelism and not many RAW dependences. Otherwise, the additional overheads of these systems (task squashes and extra memory latency) will not be able to speedup the sequential execution.

If we focus on AMM systems, its performance can degrade if the amount of speculative state is high. The speculative state can be high, because the working set of each particular task is already high. Also, under MultiT schemes the amount of speculative state may grow significantly when the application has task load imbalance, and as a result some processors end up holding the speculative state from several tasks while others are busy

with long tasks. Under these circumstances, the speculative state can overflow caches, and the tasks will have to stall. Adding an overflow area in the local memory to hold the speculative state is only a partial solution, because versions still need to be accessed at commit time. Since accessing the local memory is slower than accessing the cache, the commit wavefront can be delayed and appear in the critical path.

Performance of SingleT AMM schemes will be limited by task load imbalance. This limitation can be removed with MultiT schemes. However, if the support only includes MultiT&SV, and the application hast mostly-privatization access patterns, the performance will be similar to the one achieved by SingleT schemes. With MultiT&SV, a processor can start to execute a new speculative task before the previous executed one has committed. However, the new task will almost inmediately stall if it tries to produce a new version of an uncommitted variable which is already buffered by the local processor. This problem can be removed by adding the MultiT&MV support.

If we compare Eager vs Lazy AMM schemes, Eager AMM schemes will suffer in performance when the Task Commit Wavefront appears in the critical path. This can occurs between tasks under SingleT, and MultiT&SV if the applications have mostly privatization access; if not, the Task Commit Wavefront can appear at the end of the task execution under MultiT&MV schemes.

All these tradeoffs are evaluated in Section 5.7.

5.6 Evaluation Methodology

5.6.1 Simulation Environment

To evaluate the performance of the different buffering schemes, we use an execution-driven simulation system based on MINT [VF94a] that supports out-of-order superscalar processors executing MIPS-2 binaries. We model a *16-node* CC-NUMA where each node contains a fraction of the shared memory and directory, and a 4-issue processor. The processor has a 64-entry instruction window and 4 Int, 2 FP, and 2 Ld/St units. It supports 8 pending loads and 16 stores. It has a branch penalty of 8 cycles and has 64 Int and 64 FP rename registers (96-entry physical registers). It also has a 2K-entry BTB with 2-bit saturating counters.

Each node has a 2-way 32-Kbyte L1 D-cache and a 4-way 512-Kbyte L2, both with 64-byte lines and a write-back policy. We use a small L2 because the applications are small. The average no-contention round-trip latencies from the processor to the on-chip L1 cache, L2 cache, and memory in the local node are 2, 12, 75 cycles, respectively. When accessing the memory in a remote node this latency is 208 cycles, and when an additional access

to a processor's cache is required, then the latency is 291 cycles. Contention is accurately modelled in the whole system, except in the global network where we only model the source and destination ports. The caches are kept coherent with a release-consistent protocol like that of DASH [LLG+90]. Pages of shared data are allocated round-robin across the nodes. We choose this allocation because our applications have irregular access patterns. Private data are allocated locally.

We model all the non-shaded buffering approaches in our taxonomy of Figure 5.7-(a). The only exceptions are: eager FMM (which is effectively a cache write-through scheme and causes so much traffic that results in very low performance) and coarse-recovery schemes (which are different than the rest). To detect dependences, we use the speculative protocol in [Zha99], which is also described in Appendix A. The protocol is appropriately modified to adapt to each box in Figure 5.7-(a). Using the same base protocol for all the cases makes it easier to identify the true differences between the schemes. Recall that this protocol supports multiple concurrent versions of the same variable and triggers squashes only on out-of-order RAWs to the same word. We avoid processor stalls due to L2 conflict or capacity limitations using a per-node overflow area in local memory. We support multiple versions in AMM systems using the 4 ways of the L2 and the overflow area. For FMM systems, the per-node MHB is allocated in the local memories. Cache lines displaced from L2 are sent to both local and home memory.

To avoid tagging the L1 with task IDs, in all schemes, the L1 is traversed when a task execution ends and the dirty lines are written back to L2. For task commit in eager AMM systems, we use a hardware table that identifies the lines in the L2 cache and overflow area that need to be written back to main memory [PGRT01, SCZM00]. For lazy AMM systems and FMM systems, main memory is tagged with task IDs used to collect the youngest versions. In MultiT eager AMM systems, version merging is done in the background. Our simulations model all overheads, including dynamic scheduling of tasks to processors, task commit, and recovery from violations. In FMM systems, recovery from violations is performed using software handlers whose execution is fully simulated.

5.6.2 Applications

We use a set of scientific applications where a large fraction of the code is not fully analyzable by a parallelizing compiler. These applications are: *Apsi* from SPECfp2000 [J.L00], *Track* and *Bdna* from Perfect [B+89], *Dsmc3d* from HPF-2 [DSH94], *P3m* from NCSA, and *Tree* from [J. 94]. Appendix B gives a brief description of each one. We use the Polaris parallelizing compiler [BDE+96] to identify the non-analyzable sections and prepare them for speculative parallelization. The source of non-analyzability is that the dependence structure is either too complicated or unknown because it depends on input data

or control flow. For example, the code often has indirect indexing to arrays and complex conditionals that depend on array values. In these sections, Polaris marks the accesses to non-analyzable data, which will trigger speculation protocol actions.

Table 5.2 shows the non-analyzable sections in each application. These sections are loops, and the speculative tasks are chunks of consecutive iterations. The chunks of iterations are dynamically scheduled. The table lists the weight of these loops relative to *Tseq*, the total *sequential* execution time of the application with I/O excluded. This value, which is obtained on a Sun Ultra 5 workstation, is on average 51.4%. Note that the average weight (45.5%) would be even higher in a parallel execution, where the other sections of the code are parallelized. The table also shows the number of invocations of these loops during execution, the number of tasks per invocation, the number of instructions per task, and the ratio between the time it takes for a task to commit and to execute (*Task Commit Ratio*). This ratio was computed under an eager MultiT&MV system where tasks never stall. All the data presented in the evaluation, including speedups, refer only to the code sections in the table.

Appl	Non-Analyz Sections (Loops)	% of Tseq	# Invoc	# Tasks per Invoc	# Instr per Task (Thous)	Task Commit Ratio
P3m	*pp_do100*	56.5	1	97336	69.1	0.003
Tree	*trwalk*	92.2	41	4096	28.7	0.014
Bdna	*actfor_do240*	44.2	1	1499	103.3	0.060
Apsi	*run_do[20,30, 40,50,60,100]*	29.3	900	63	102.6	0.114
Track	*nlfilt_do300*	58.1	56	126	22.3	0.084
Dsmc3d	*move3_goto100*	41.2	80	46777	5.4	0.062
Average		51.4	180	24470	55.3	0.056

Table 5.2. Application characteristics. Each task is one iteration, except in *Track* and *Dsmc3d*, where it is 4 and 16 consecutive iterations, respectively.

Finally, Table 5.3 give a qualitative indication of the behaviour of the applications. The first column refers to the load imbalance between nearby tasks. The second one refers to the importance of mostly-privatization patterns. We estimate it by counting what fraction of the tasks overwrite a version generated by a predecessor without first reading it. The next column refers to how critical can be the commit wavefront in the parallel execution. This column refers to the size of The Task Commit Ratio in last column in Table 5.2. Remember that when the task commit ratio times the number of processors is greater than 1, the task commit wavefront appears in the critical path. Finally, the last column refers to the frequency of task squashes. Run-time dependence violations occur only in *Track* and *Dsmc3d*. In *Track*, one violation occurs in 3 of the 56 loop invocations.

In *Dsmc3d*, an average of 30 violations occur in each of the 80 loop invocations.

Appl	Appl Characteristics			
	Load Imbal	Priv Pattern	Critical Commit	Task Squash
P3m	High	Med	No	None
Tree	Med	High	No	None
Bdna	Low	High	Maybe	None
Apsi	Low	High	Yes	None
Track	Med	Low	Yes	Low
Dsmc3d	Low	Low	Maybe	Low

Table 5.3. Qualitative characteristics of the Applications.

5.7 Evaluation

5.7.1 Single vs Multiple Speculative Tasks/Versions per Processor

Figure 5.16 compares the execution time of the applications under schemes where individual processors can support: a single speculative task (*SingleT*), multiple speculative tasks but only single versions (*MultiT&SV*), and multiple speculative tasks and multiple versions (*MultiT&MV*). All schemes are Eager AMM. The bars are normalized to *SingleT* and broken down into instruction execution plus non-memory pipeline hazards (*Busy*), and stalls due to memory access, not enough task/version support, and end-of-loop stall due to the commit wavefront or load imbalance (*Stall*). The numbers on top of the bars show the speedup relative to sequential execution.

The most advanced scheme (*MultiT&MV*) should perform much better than the simplest one (*SingleT*) under two conditions. One is in highly load-imbalanced applications. According to Table 5.3, this is the case for *P3m*, which reflects in Figure 5.16.

The other condition is when the load imbalance is modest but the task commit ratio is sizable. In this case, commits are in the critical path of restarting stalled processors (Figure 5.14-(c)). Note that the task commit ratio should not be too high. If it is, the end-of-loop stall due to the commit wavefront (Figure 5.14-(a)) will be the one to determine the execution time in both *MultiT&MV* and *SingleT*. According to Table 5.3, the applications with modest imbalance and sizable task commit ratio are *Bdna* and *Dsmc3d*. In these applications, *MultiT&MV* in Figure 5.16 is much faster than *SingleT*. For the other applications, the task commit ratio is either too high or too low for *MultiT&MV* to be much faster than *SingleT*.

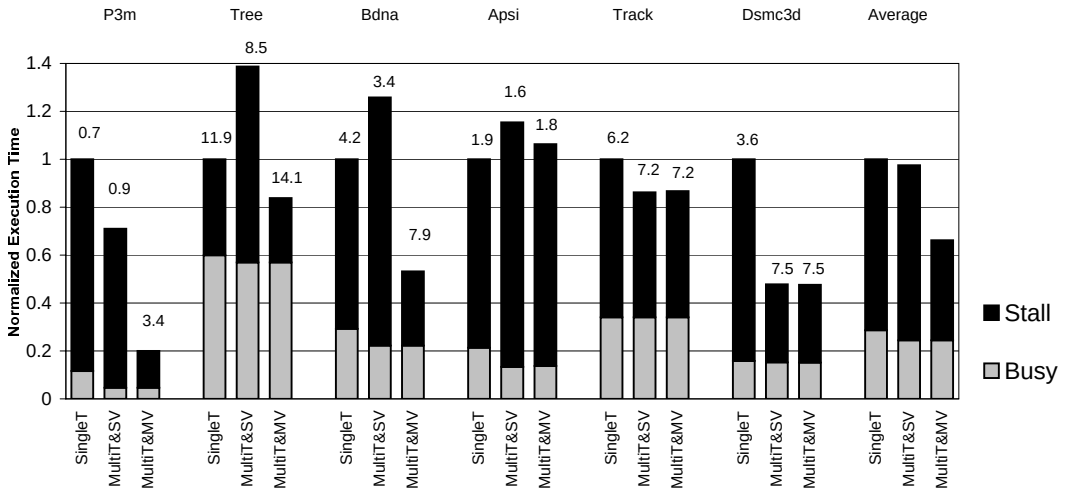

Figure 5.16. Effect of supporting single or multiple speculative tasks or versions per processor.

MultiT&SV should match *MultiT&MV* unless mostly-privatization patterns dominate, in which case it should resemble *SingleT*. According to Table 5.3, *Track* and *Dsmc3d* do not have such patterns. Consequently, *MultiT&SV* and *MultiT&MV* in Figure 5.16 are similar for them. *P3m* has some of these patterns, and so *MultiT&SV* is between the other two bars.

The remaining applications (*Tree*, *Bdna*, and *Apsi*) have such patterns. *MultiT&SV* should have no advantage over *SingleT*: in practice, a processor cannot make progress until the task that it has just executed becomes non-speculative. This means that in both schemes only P tasks can make progress at a time, where P is number of processors. Under *SingleT*, since tasks are effectively assigned round-robin, there is a window of only P assigned speculative tasks at any time. However, under *MultiT&SV*, since tasks are greedily assigned, the window can grow longer than P. In this case, there is no control of which tasks are making progress. It may be that some tasks far away from the commit point are executing, while other tasks closer to it are temporarily stalled. This effect delays the commit wavefront and therefore slows down *MultiT&SV* for these applications in Figure 5.16. More details together with an example is shown in Appendix C.

Overall, *MultiT&MV* is a good scheme: the average execution time of the applications is 35% lower than under *SingleT*.

5.7.2 Eager vs Lazy Merging with Main Memory State

Figure 5.17 compares the execution time under eager and lazy merging schemes. The figure repeats Figure 5.16 but adds a *Lazy* bar to each scheme. As usual, all schemes are AMM.

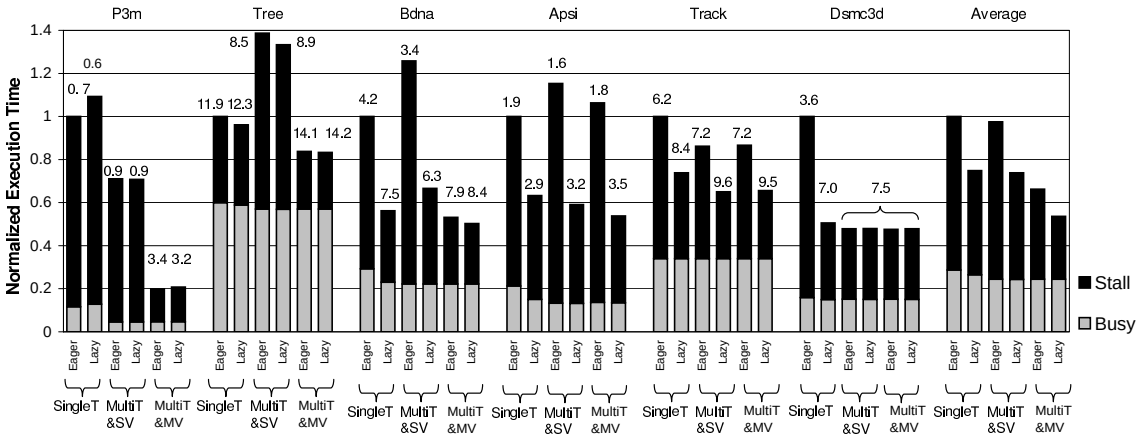

Figure 5.17. Effect of supporting eager or lazy merging with main memory state.

A lazy scheme can only speed-up execution if the corresponding eager scheme has the commit wavefront in its critical path of execution. As discussed in Section 5.5.3, the commit wavefront can appear in the critical path either between task execution (Figure 5.14-(c)) or at the end of the speculative execution (Figure 5.14-(a)). The first scenario appears when processors stall during task execution, typically under *SingleT* and, if mostly-privatization patterns dominate, under *MultiT&SV*. The second scenario appears when processors do not stall during task execution but the task commit ratio times the number of processors is higher than 1.

The first scenario occurs frequently in our applications: in all cases under eager *SingleT* and for the privatization applications (*P3m*, *Tree*, *Bdna*, and *Apsi*) under eager *MultiT&SV*. In all these cases, the impact of laziness will be proportional to the application's task commit ratio. From Table 5.2, we see that the ratio is significant for all applications except *P3m* and *Tree*. Consequently, for this first scenario, laziness should improve *SingleT* for *Bdna*, *Apsi*, *Track*, and *Dsmc3d*, and *MultiT&SV* for *Bdna* and *Apsi*. This is consistent with Figure 5.17.

The second scenario could appear for all applications under *MultiT&MV*, and for the non-privatization ones (*Track* and *Dsmc3d*) under *MultiT&SV*. However, according to Table 5.2, only *Apsi* and *Track* have a task commit ratio sufficiently high such that, when multiplied by 16, it is clearly over 1. Consequently, laziness should improve *MultiT&MV* for *Apsi* and *Track*, and *MultiT&SV* for *Track*. This is consistent with Figure 5.17[3].

[3]Our conclusions on laziness agree with [PGRT01] for 16 processors for the common applications: *Tree*, *Bdna*, and *Track* (*Apsi* cannot be compared because the problem sizes are different). Our *MultiT&MV* eager and lazy schemes correspond to their *Opt* and something in between *OptNoCT* and *OptNoCTL1*, respectively. Overall speedup figures are somewhat different because the speculation protocols used here

Overall, laziness is effective across the board. For the simpler schemes (*SingleT* and *MultiT&SV*), it reduces the average execution time of the applications by 25%, while for *MultiT&MV* the reduction is 20%. In particular, lazy *SingleT* schemes constitute a cost-effective design point for lightly-imbalance applications (all our applications but *P3m*).

Finally, we have also evaluated a *MultiT&MV* system where the number of supported versions is limited by the size and associativity of the L2 cache: 512-Kbyte and 4, respectively, in our experiments. Figure 5.18 shows results for an eager and a lazy scheme. For each scheme there are two bars. The first one supports unlimited number of versions (*Infinite*), and repeats the bar from Figure 5.17. In the second one, a task stalls in case of displacement of a cache line with uncommitted data (*4vers*). The stalled task proceeds when the task that produced the data that is about to be displaced becomes non-speculative. The Figure show differences only for *P3m* and *Bdna*.

In *P3m*, which is a highly imbalanced application, limiting the amount of speculative versions hurts performance seriously in both *Eager* and *Lazy* schemes. In *Bdna*, *4vers* runs 16% slower than *Infinite* with an eager system. However, the difference between *4vers* and *Infinite* with a lazy system is only 7%. The reason is that with a lazy scheme, commit is faster. This speeds-up the transfer of the commit token, reducing the probability of task stall. For the rest of our applications, since they are only slightly imbalanced and their working sets fit in the cache, there are almost no differences between *Infinite* and *4vers*.

Thus, a conclusion of this experiment is that when speculative state overflows moderately, laziness helps reducing the stall time. However, in case of high imbalance where speculative state overflows significantly, laziness has no effect, as the execution time is dominated by the task stall time.

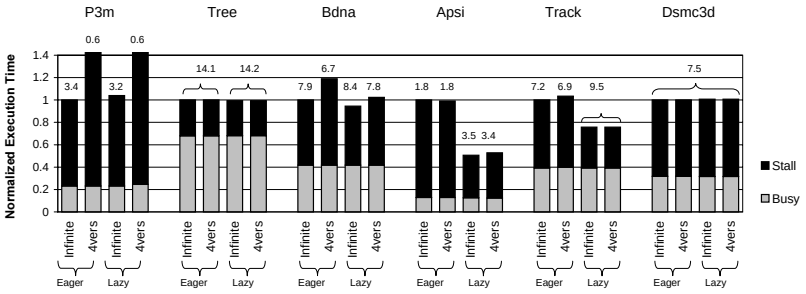

Figure 5.18. Effect of limited support for eager or lazy merging with main memory state.

5.7.3 Architectural vs Future Main Memory System

Figure 5.19 compares the execution time under architectural and future main memory system schemes. All bars use *MultiT&MV* support. For the AMM system, we repeat the

and in [PGRT01] have differences.

eager and lazy bars from Figure 5.17. For the FMM system, we show the baseline lazy scheme (*Lazy*) and the same scheme but where the copying of versions to the memory history buffer is done in software (*Lazy.SW*). In such a scheme, which was presented in Garzaran01 [GPL+01], the application includes extra instructions that perform the copying. A detailed explanation of this software scheme follows in the next chapter. As indicated in Section 6.5.1, we do not evaluate an eager FMM design.

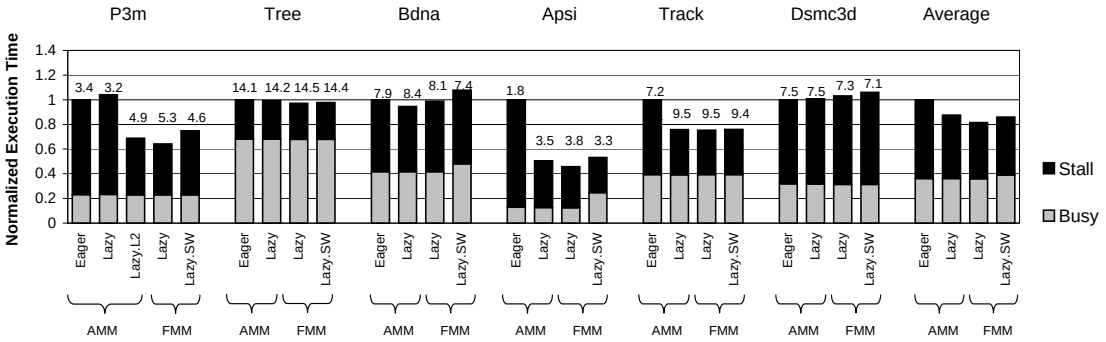

Figure 5.19. Effect of supporting an architectural or a future main memory system.

FMM schemes are better suited to version commit, while AMM schemes to version recovery. In our applications, dependence violations are not especially frequent (Table 5.3).Therefore, we would expect AMM schemes to be slower.

In practice, while the figure shows that the simple Eager AMM scheme is slow, the more sophisticated Lazy AMM has a performance very close to Lazy FMM. The average execution time on Lazy FMM is only 7% shorter than on Lazy AMM. We conclude, therefore, that the additional hardware added to support laziness in AMM is effective.

Examining the differences between Lazy FMM and Lazy AMM, we see that the recoveries required in *Dsmc3d* and *Track* do not affect the schemes much. The only major difference between the two schemes occurs in *P3m*, where Lazy AMM is slower. *P3m* has load imbalance and mostly-privatization patterns. As a result, under AMM, the memory reorder buffer in a processor keeps the state of numerous speculative tasks, with multiple versions of the same variable. The large size of the state induces cache displacements due to capacity to the overflow area. Furthermore, since all versions of the same variable map into the same set, L2 also suffers displacements due to conflicts. Overall, the resulting loss of locality slows down *P3m*.

To eliminate this problem, we have increased L2's size and associativity to 4 Mbytes and 16 ways, respectively (Lazy.L2 bar in *P3m*). In this case, AMM performs just as well as FMM.

Finally, Figure 5.19 agrees with [GPL+01] in that the performance hit of copying

versions to the memory history buffer in software (Lazy.SW) is modest. Lazy.SW takes only 7% longer than Lazy FMM to run the average application. While it eliminates some hardware, it still needs the most complex Lazy FMM hardware, namely the tagging of versions in memory.

5.7.4 Summary

These results lead us to several conclusions. First, the analysis of the multiple tasks/versions per processor indicates that *MultiT&SV* is undesirable: its performance is similar to *SingleT* while its complexity is nearly as high as *MultiT&MV*.

Starting from the simplest scheme (*SingleT* Eager AMM), we have the choice of adding support for multiple tasks&versions or for laziness. Our second conclusion is that adding multiple tasks&versions is more cost-effective. Indeed, the reduction in execution time is higher: 35% versus 25% for laziness. Furthermore, the hardware complexity is lower for adding multiple tasks&versions. Specifically, *MultiT&MV* requires task IDs in caches and logic to manipulate them. Laziness requires supports to know the order of the committed versions left in caches, and to collect them correctly in memory. The order can be known with task IDs in caches [PGRT01] or with a linked list [GVSS98]. Collection in memory requires support for merging on displacement [PGRT01] or tags in memory [Zha99].

A third conclusion is that the improvements due to multiple tasks&versions and due to laziness are fairly orthogonal. Indeed, adding laziness to the *MultiT&MV* Eager AMM scheme reduces the execution time by an additional 20%, resulting in a high-performance system.

Finally, the resulting system (*MultiT&MV* Lazy AMM) is shown to have sufficient features to be generally competitive against the most expensive and fastest system (*MultiT&MV* Lazy FMM). The AMM scheme is usually as fast as the FMM scheme. Its only weakness is that it is not as robust. Indeed, applications with sizable working sets, significant load imbalance and, possibly, mostly-privatization patterns put stress on the cache hierarchy of *MultiT&MV* Lazy AMM. The *MultiT&MV* Lazy FMM scheme does not have this problem. However, it is more expensive, mostly because it needs memory tags [Zha99].

5.8 Conclusion

This chapter made two main contributions. First, it introduced a novel taxonomy of approaches to buffer multi-version memory state for speculative parallelization in multiprocessors. We applied three criteria: a novel application to memory state of the concept

of architectural vs future state, lazy vs eager merging, and multiple vs single speculative tasks and versions per processor. On this taxonomy, we mapped buffering schemes proposed elsewhere.

The second contribution was a tradeoff analysis and a detailed performance evaluation of the different degrees of freedom in our taxonomy. The performance evaluation was based on simulations of all the approaches under a single baseline architectural framework.

Our key insights are that support for multiple speculative tasks&versions is more cost-effective than support for lazy merging. Moreover, both supports have largely orthogonal effectiveness and can be combined. A lazy, multi-task&version scheme under AMM is nearly as fast as the same scheme under FMM. However, it is not as robust as the latter under applications that stress the cache hierarchy. Overall, the lazy, multi-task&version FMM scheme is the fastest and most robust. However, it is also the most expensive.

Our future work is two-fold. First, we are examining software solutions to simplify the hardware of the lazy, multi-task&version FMM scheme. Second, we are examining applications that have frequent violations, where AMM may have an edge.

References

[B+89] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[BDE+96] 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. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[CMT00] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 13–24, June 2000.

[DSH94] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[ELPS98] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 140–151, February 1998.

[FF01] R. Figueiredo and J. Fortes. Hardware Support for Extracting Coarse-grain Speculative Parallelism in Distributed Shared-memory Multiprocesors. In *Proc. of the Int'l Conference on Parallel Processing (ICPP'01)*, September 2001.

[FLA01] M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Technical Report MIT/LCS Technical Memo 619, July 2001.

[FS96] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Trans. on Computers*, 45(5):552–571, May 1996.

[GN98] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proc. of Supercomputing '1998*, November 1998.

[GPL+01] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and Josep Torrellas. Software Logging under Speculative Parallelization. In *Workshop on Memory Performance Issues hold in conjunction with ISCA'01*, July 2001.

[GPL+02] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Software Logging for Multi-Version Buffering under Speculative Parallelization. Technical Report RR-02-04, Departamento de Informática e Ingeniería de Sistemas, March 2002.

[GVSS98] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 195–205, February 1998.

[HK99] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proc. of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 171–181, January 1999.

[HWO98] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *In Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 58–69, October 1998.

[J. 94] J. E. Barnes. Treecode. Technical Report, Institute for Astronomy, University of Hawaii, 1994.

[J.L00] J.L. Henning. SPEC CPU2000: Measuring Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[Kni86] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[KT99] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, pages 866–880, September 1999.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int'l Symposium on Computer Architecture (ISCA'90)*, pages 148–159, May 1990.

[MG99] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *Proc. of the 1999 Int'l Conference on Supercomputing (ICS'99)*, pages 365–372, June 1999.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. of the 1995 Int'l Conference on Parallel Processing (ICPP'95)*, pages 1–10, August 1995.

[OWP+01] C.L. Ooi, S. Wook, K.I. Park, R. Eigenmann, B. Falsafi, and T.N. Vijaykumar. Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor. In *Proc. of the 2001 Int'l Conference on Supercomputing (ICS'01)*, June 2001.

[PGRT01] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proc. of the 28th Annual Int'l Symposium on Computer Architecture (ISCA'01)*, pages 204–215, July 2001.

[RP95] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. of the SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 218–232, June 1995.

[RS00] P. Rundberg and P. Stenstrom. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[SBV95] G. S. Sohi, S. Breach, and S. Vajapeyam. Multiscalar Processors. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 414–425, June 1995.

[SCM97] J.G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[SCZM00] J.G. Steffan, C.B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 1–12, June 2000.

[SP88] J.E. Smith and A.R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. on Computers*, C-37(5):562–573, May 1988.

[THA+99] J.Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.C.Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):881–902, September 1999.

[VF94] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[Zha99] Y. Zhang. Hardware for Speculative Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, May 1999.

[ZRT98] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 162–174, February 1998.

[ZRT99] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 135–139, January 1999.

Chapter 6

Software Buffering Under Speculative Thread-Level Parallelization

Future Main Memory state buffering appears to be a good solution for speculative thread-level parallelization. However, systems based in this approach have a high hardware complexity, that comes in part, from the need of buffering speculative state. This complexity could be removed if buffering were triggered by software actions.

In this chapter, we explore a software-only implementation to buffer speculative state for lazy Future Main Memory systems. We take a speculation protocol, and build a software-only buffering scheme with support to buffer state from multiple tasks and multiple versions. Our design examines the major issues like software access to task-IDs, and overhead reduction by filtering first stores, bypassing the cache, or using predication.

Our simulations of a 16-processor CC-NUMA show very promising results. Our software implementation only adds an overhead of a 7% over a similar hardware-only scheme, while it removes most of the hardware complexity.

6.1 Introduction

In the previous chapter, we have studied the main tradeoffs to buffer the speculative memory state from multiple tasks and multiple versions (MultiT&MV) under speculative thread-level parallelization. We have also presented a novel taxonomy that classifies existent schemes based on the approach taken to buffer this speculative memory state. Our taxonomy differentiates between Architectural Main Memory (AMM) Systems and Fu-

ture Main Memory (FMM) Systems. With AMM systems, caches, write buffers or special buffers keep speculative state to prevent the architectural state in main memory to be corrupted. However, with FMM systems main memory keeps the most advanced speculative state, while distributed memory history buffers (MHB) keep the previous state (either architectural or speculative), so that if errors are found at run-time, the architectural state of main memory can be restored. Our experiments have shown that FMM systems appear to be a robust system, but have the inconvenience of a higher hardware complexity.

The hardware complexity of FMM systems comes from different places. Part of it is shared with AMM systems, and refers to the speculation engine that checks for dependence errors and the need for task squashes, or finds the correct version in an in-order RAW dependence. However, FMM systems have extra complexity to buffer speculative state. On one side, they need to tag main memory with task-IDs to allow out-of order version displacement. On the other side, implementing a distributed MHB for a MultiT&MV speculative system where only out-of-order RAW dependences cause errors is complex. Each processor needs to keep a MHB where previous updates are recorded. Moreover, if speculative state can overflow caches, extra support is needed. This extra support may be an overflow area in the local memory of the processor where speculative state can overflow into.

As suggested at the end of the previous chapter, we feel that exploring solutions to simplify the hardware complexity of lazy FMM systems is worthwhile. Thus, in this chapter we explore memory state buffering scheme for such MultiT&MV systems. Our proposed scheme is implemented through software-only algorithms. In particular, we design, implement and evaluate a software-only MultiT&MV buffering scheme that runs on top of the speculation protocol presented in [Zha99, ZPG+99] (and explained in detail in Appendix A). In our designs, we examine the major issues like software access to task-IDs, overhead reduction by filtering first stores, bypassing the cache, or using predication. We evaluate the whole scheme with simulations, and we find that the implementation of our software MultiT&MV buffering scheme is very effective. For a 16-processor CC-NUMA running speculatively parallelized applications, our scheme adds an overhead of only 7% to the execution time of a similar hardware-only scheme. We also evaluate performance of several alternative designs.

This chapter is organized as follows: Section 6.2 sketches the speculative protocol that we use; Section 6.3 present an overview of the history buffer operation; Section 6.4 presents our implementation; Section 6.5 describes our evaluation methodology; Section 6.6 presents the evaluation; and Section 6.7 summarizes.

6.2 Speculation Protocol Used

To give a concrete example of how a software implementation of a buffering scheme for a lazy FMM system could be used, we use the speculative parallelization protocol from [Zha99, ZPG+99]. That protocol included a MHB or log managed in hardware with support to buffer MultiT&MV memory state. For that, each node had a hardware controller embedded in the directory module that worked in the background with very small overhead. In this chapter, we take that protocol and explore an inexpensive software implementation of the MHB. We now briefly describe some details of the protocol. More details of the protocol and the hardware-based buffering scheme can be found in Appendix A.

The architectural platform assumed in [Zha99, ZPG+99] is a CC-NUMA machine. In this system, only out-of-order RAW dependences cause the squash of the tasks. The rest of the dependences can be detected and solved at run-time. Speculative accesses are marked with special load and store instructions that trigger the protocol. When one of these accesses reaches a shared data page for the first time, a page of shared task IDs is allocated. This page, which is kept in the home memory contains a *MaxWDisp* task-ID per word in the data page. *MaxWDisp* will record the version of the word in the memory, which is the ID of the youngest task that wrote the word and displaced it to the home. At any time, the home only accepts incoming versions of words that are younger than those it already has.

In addition, to avoid task squashes due to cache conflicts or capacity problems, this protocol uses an overflow area in the local memory of the processor, where versions in the cache can flow into. Thus, the first time a node displaces a cache line that has speculatively been modified, the OS places a new page in the node's local memory, where the displaced line is saved. The node's local memory also keeps a page of local *task IDs* for each page of data that has been speculatively accessed. These task IDs will record, for each word, the ID of the youngest local task that writes the word (*PmaxW*), and the ID of the youngest local task that reads it without writing it first (*PmaxR1st*). The latter operation is an *exposed load*. These task IDs are automatically updated by a dependence-detecting hardware engine with small overhead. For a given task and variable, *PmaxR1st* and *PmaxW* serve as a filter. They detect events that must be communicated to the home, namely when a version from outside the node is needed and when a successor task may have to be squashed respectively.

As a summary of the local time stamps, the L1 and L2 cache tags keep a *Read1st* and a *Write* bit per word. They indicate whether the current task has issued an exposed load or

has written the word, respectively. These bits, which are cleared when a new task starts, are used to reduce messaging to the task IDs in the local memory.

6.3 Overview of the History Buffer Operation

In this section, we give an overview of a MultiT&MV buffering scheme based in a history buffer approach. A possible organization of the MHB for a lazy FMM system supporting multiple tasks and multiple versions is shown in Figure 6.1. Each processor in the CC-NUMA machine allocates one history buffer in its local memory. The compiler estimates the size of the sectors and history buffer based on the number of writes in a task and the number of tasks per processor that are likely to be uncommitted at a time, respectively. The history buffer is broken down into fixed-sized sectors that keep the state overwritten by individual tasks.

When a task starts running, it is dynamically assigned an entry in the Task Pointer Table and one sector in the history buffer. Free sectors are obtained from the Free Sector Stack. Two pointers in the Task Pointer Table point to the Next entry to fill and the End entry to check for overflow. If the task needs more entries than a sector, another sector is dynamically assigned and linked to the previous one, while the Overflow bit is set.

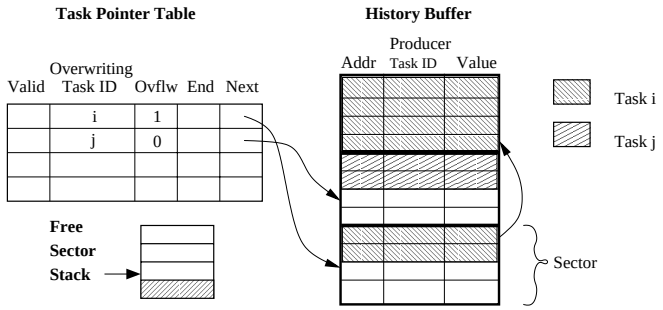

Figure 6.1. Per-processor structures that we use for the history buffer.

A history buffer supports four operations: saving a record in the history buffer (*Insertion*), freeing up the entries in the history buffer that are useless (*Recycle*), unwinding the history buffer to restore the architectural state after a violation (*Recovery*), and finding a record in the history buffer (*Retrieval*).

Insertion. Before a version of a variable is overwritten by a new one, the old version is saved into the history buffer. A record includes the following information about the old version: its virtual address (the only one the software knows), its value before the update, and the its producer task ID. After the record is inserted at run time, the Next pointer

is incremented. At the end of a task, all the versions that it overwrote are in contiguous locations in one or more sectors, easily retrievable through the Task Pointer Table with the task ID.

Recycle. After a task has committed, all the versions in its sector become obsolete. Therefore, the sector can be recycled. This is done by invalidating the corresponding Task Pointer Table entry and returning the sector to the Free Sector Stack.

Recovery. When an out-of-order RAW dependence between tasks is detected, we recover by undoing the version updates performed by the successor tasks to the writing task. This process is done in parallel by a software handler running on each of the processors that ran any of the tasks that needs to be undone.

The process involves two actions. First, the cached versions belonging to these tasks are written back to memory with their task IDs. Second, the history buffer sectors belonging to these tasks are traversed in decreasing task ID order to recover versions, addresses, and producer task IDs. These versions are also written back to memory with their task IDs. In all write backs, the memory only accepts the versions whose task ID is higher than or equal to the task ID of its current version and at the same time smaller than the ID of the offending reader task. This is required because the write backs reach memory out of order. At the end, the memory recovers the architectural state.

Retrieval. Retrieval may be required when a exposed load involved in an in-order RAW dependence require to access a non-last version. In this case, a read by a task running on a processor cannot be satisfied by the cache of the second processor where the producer task executed. The reason is that a newer task on the second processor has overwritten the variable and pushed the required version into the history buffer. In this case, we search the history buffer in the producer processor in decreasing task ID order, looking for the version overwritten by the oldest task still younger than the reader task.

Since these two cases happen infrequently, they can be solved with software exception handlers that access the history buffers.The algorithms for Recovery and Retrieval are discussed with more detail in Appendix A.

6.4 Efficient Software Implementation

The system that we have just described can be implemented by constructing the MHB and all the operations of Section 6.3 in hardware [Zha99, ZPG+99]. In this section, we explore a software-based implementation [GPL+00, GPL+01]. Our solution is to implement the MHB as a plain software data structure. We now examine the design of a software

MultiT&MV buffering scheme built on top of the speculation protocol presented in 6.2 (and extended in Appendix A). In the following, we examine several challenging design issues: accessing task IDs from software (Section 6.4.1), caching task IDs (Section 6.4.2), main overheads (Section 6.4.3), filtering first stores (Section 6.4.4), filtering using predication (Section 6.4.4), exposed loads (Section 6.4.5), and other issues (Section 6.4.6). Finally, we explain the atomicity (Section 6.4.7) that can appear in other speculation protocols, and we compare our software solution with the hardware-only implementation (Section 6.4.8).

6.4.1 Accessing Local Task-IDs in Software

The task IDs associated with each cached word are allocated in local memory pages. They are read and updated automatically by the dependence-detecting hardware as part of the speculation protocol. In fact, these pages containing task-IDs are not even mapped in the virtual address space of the tasks. Nevertheless, we must find a way to make them visible to the software.

To this end, when a processor access speculatively for the first time a page of shared data, the OS copies it to the node's overflow area. For that, it uses an even-numbered physical page number. Then, it allocates the task IDs in the immediately following odd physical page. With this setup, we can access a task ID in software with a special load:

lh_TID Reg, AddrVar

where *AddrVar* is the virtual address of the variable. When this instruction executes, the hardware takes the physical page number as it comes out from the TLB and flips the least significant bit. The result is that the address issued to memory is that of the task IDs associated with the variable. That location contains the 16-bit *PmaxR1st* and the 16-bit *PmaxW* task IDs. Since we only want to load the least significant 16 bits, where *PmaxW* is stored, the instruction works as a half-word load (*lh*). Overall, with this modest support, the software can read the *PmaxW* task IDs as data, while needing only the virtual address of the variable. Figure 6.2 shows an example of how this mechanism works.

Notice that the TLB translates the shared virtual address of the data that are speculatively accessed into the local physical address in the overflow area. This local physical address is also used to tag caches. However, to access the data in the home we need the shared physical address. To that end, the Address Traslation Module in Figure 6.2 is used. This table takes the local physical address, and returns the shared physical one. This translation will be needed on the first exposed load and first write in a task, when a special message needs to be sent to the home. In addition, this table also contains the reverse translation, from the shared physical address to the local physical one. This will

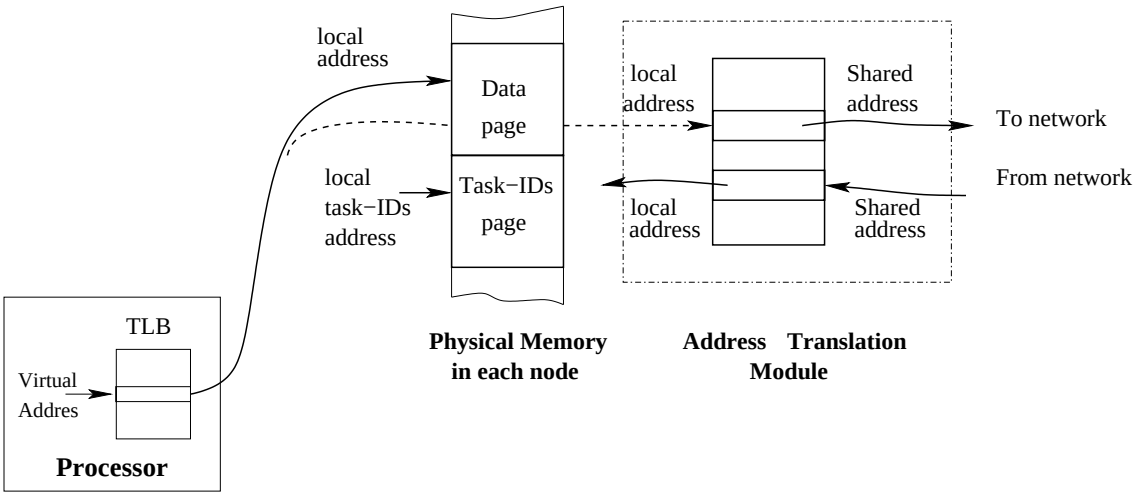

Figure 6.2. Special mapping to access task-IDs from software and Address Translation Module.

be needed on a remote access from the home node that tries, for example, to find the correct version in an in-order RAW dependence. This translation module is similar to that in Prism [ELPS98], WildFire [HK99], or S3.MP [NAB+95].

This particular implementation allows the software to access the task IDs when using the speculation protocol sketched in Section 6.2 that has an overflow area in the local memory node. However, we can also address the task IDs from software in a speculation protocol that does not support an overflow area. Conceptually, the solution is to add an extra field to each TLB entry. The new field contains the frame number of the task IDs. A TLB entry now points to the physical pages of both data and task IDs. Then, we can use the same load instruction we have just described. When *lh_TID Reg, AddrVar* is executed, the TLB delivers the corresponding page of task IDs.

Fortunately, this idea can be implemented without modifying the TLB. Specifically, when a data page is speculatively accessed for the first time, a page of task IDs is allocated, and the TLB mapping of the data page is changed. The new mapping is a non-existent local physical page located at a fixed address offset from the task ID physical page. Subsequent accesses to the *data* will obtain from the TLB such (non-existent) local physical addresses, which will be used to tag caches. In case of a cache miss, the shared address at the home node is obtained from the Address Translation Module we just explained. On the other hand, *lh_TID* instructions to the virtual address of the data obtain the non-existent local physical addresses from the TLB and automatically subtract the offset. The result is an access to the task IDs, as intended.

6.4.2 Caching Task-IDs

We want a *lh_TID* to bring copies of *PmaxW* into the cache and access them from there with the subsequent *lh_TID*. However, the *PmaxW* master copies in the local memory may be modified in hardware by the speculation protocol at speculative accesses. Consequently, to keep the cached copies up to date, we treat them as reagular data and we update them in software to the same value that the hardware updates the master copies to.

This is accomplished with a *store half-word task ID (sh_TID)* instruction. After the record is created, we take the ID of the executing task and write it on the cached copy of *PmaxW* using *sh_TID*. This operation updates the cached copy to the correct version number. As in *lh_TID*, *sh_TID* takes as argument the virtual address of the variable and only updates the half word where the *PmaxW* copy is, leaving the adjacent *PmaxR1st* unaffected. Such an update does not affect the page with the task IDs in memory. The task IDs pages are *read-only* for the software, so that the displacement of dirty task IDs lines from the cache does not overwrite them.

Overall, we are allowing software-updated task IDs in the cache while keeping the hardware-managed master copy in memory. This is a trade-off decision that maximizes performance while minimizing the additional support required. The coherence of cached data is ensured, even in the presence of cache displacements, if we issue *sh_TID* at every instrumented store. Note also that the hardware guarantees that accesses to a variable and to its task ID are always issued to the memory system in order. This is because all these instructions use the same virtual address.

6.4.3 Quantifying Insertion Overhead.

Insertion is the most overhead-sensitive operation since it occurs frequently. At compile time, the compiler instruments individual stores with instructions to save a record in the MHB. Inserting a record in the buffer of Figure 6.1 involves collecting the items to save, saving them in sequence using the *Next* pointer, and advancing the pointer. As a result, the records are organized in simple sequence in the order in which they were created. While such an organization makes it harder to search for a given address in a retrieve operation, retrieve operations are much less frequent than insertions.

Figure 6.3 shows the MIPS assembly instructions added before every speculative store that we want to buffer. All memory accesses are ordinary ones and do not trigger dependence checking. We can see that we need a total of 9 instructions: 1 to check for sector overflow, 6 to collect and insert the information, 1 to increment the pointer, and 1 to update the cached task ID. Note that the load task ID instruction (*lh_TID*) loads the

16-bit $PmaxW$, which is later stored in a full word.

```
                ; r1 = upper limit of the sector
                ; r2 = address in memory to insert the record
                ; offset(r3) = address of the variable to update
                bgt     r1, r2, insertion
                ... allocate another sector
insertion:
                addu    r4, r3, offset          ; compute address of variable
                sw      r4, 0(r2)               ; store in history buffer
                lh_ID   r4, offset(r3)          ; load PmaxW task ID
                sw      r4, 4(r2)               ; store in history buffer
                lw      r4, offset(r3)          ; load value of variable
                sw      r4, 8(r2)               ; store in history buffer
                addu    r2, r2, record_size
                sh_TID  r5, offset(r3)          ; update cached PmaxW task ID
```

Figure 6.3. Instructions added before a speculative store.

Occasionally, some of the instructions can be eliminated. For example, if the compiler knows the maximum number of stores to buffer in a region, it may only need to check for overflow once at the beginning of the region: if there is enough space left, there is no need to check again. Furthermore, if the compiler knows the exact number and order of the stores to buffer, it can hardwire the offsets in the instructions that store to the buffer and avoid incrementing the pointer every time.

Finally, to be able to undo tasks, the history buffer also has to contain versions of variables overwritten by non-speculative stores, like stores to the array B in the example of Figure 6.4. B[i] stores are not speculative, since the compiler can identify that they do not have dependences with the memory references of other tasks. However, their updates need to be buffered, because in case of infraction their modifications need to be undone. In our instrumentation, we also instrument these plain stores. However, the instrumentation does not need to insert task IDs because these variables do not have them. A difference between hardware buffering schemes and our software one is that hardware schemes require special operation codes to mark that these stores need to be buffered, although no dependence detection mechanism needs to be applied. Finally, notice that these stores that cannot cause dependence infractions, but need to be buffered, may already be in the sequential code, or may appear as a result of a transformation (e. g. a privatization transformation) that the compiler has applied to parallelize the loop.

```
for (i=1; i<n; i++)
{
  ... = A[L[i]];
          ...
  A[k[i]]= ....;
  B[i] = B[i] + A[k[i]];
}
```

Figure 6.4. Stores to the array B are non-speculative but need to be buffered.

6.4.4 Filtering the First Store

As pointed out in the previous chapter, a processor only needs to keep the last version produced by each task, that is, although a task can write several times to the same variable, the processor only keeps the version with the value written by the last write.

As a result, the MHB only needs to save the value overwritten by the *first store* to the variable in the task. Consequently, to reduce overhead, the compiler should identify first stores and instrument only those. Furthermore, variables that are provably written in every task before being read do not need to be buffered.

Identifying first stores is easy for variables accessed with plain accesses, since the dependence structure is usually analyzable. However, it is not easy for variables accessed with speculative accesses because their dependence structure is non-analyzable. For the latter variables, we usually need to modify the instrumentation presented in Section 6.4.3 to dynamically test whether or not a store is a first store, and save it into the buffer or not based on the result. Since the testing code has some overhead, the compiler may sometimes decide not to perform this run-time test and buffer unconditionally. Correctness is unaffected by this unnecessary buffering, although performance may suffer.

To identify first speculative stores, we propose to use one of two possible approaches, based on task IDs or extended loads, respectively. Both approaches can be further enhanced with the use of predication. We consider all these issues next.

Filtering Using Task-IDs

One way to identify first stores is to use the $PmaxW$ task IDs, which is the ID of the youngest local task that updated the variable. Consequently, on a store, we compare $PmaxW$ to the ID of the currently-executing task. If they are the same, there is no need to save a record into the buffer because this is not a first store.

This approach is used in Figure 6.5-(a), which repeats the example in Figure 6.3 with

the new extension. For simplicity, the code does not include the check for sector overflow.

```
; r2 = address in memory to insert the record
; offset(r3) = address of the variable to write
; r5 = ID of the executing task

  lh_TID r6, offset(r3)       ; load PmaxW
  beq    r6, r5, no_insert    ; first store?
  addu   r4, r3, offset       ; insert as usual
  sw     r4, 0(r2)
  sw     r6, 4(r2)
  lw     r4, offset(r3)
  sw     r4, 8(r2)
  addu   r2, r2,record_size
  sh_TID r5, offset(r3)       ; update cached
no_insert:                    ; PmaxW
```

(a)

```
  xlw    r6, r1, of fset(r3)  ;Write bit into r6
  bgtz   r6, no_insert        ; first store?
  addu   r4, r3, offset       ; insert as usual
  sw     r4, 0(r 2)
  lh_TID r4, offset(r3)
  sw     r4, 4(r 2)
  sw     r1, 8(r 2)
  addu   r2, r2, record_size
  sh_TID r5, offset(r3)
no_insert:
```

(b)

Figure 6.5. Filtering first stores using task IDs (a) and extended loads (b). For simplicity, we do not include the check for sector overflow.

Filtering Using Extended Loads

The second way to test for first stores is to use the *Write* bit present in the tags of the L1 and L2 caches. As indicated in Section 6.2, this bit is set for a word when the current task updates the word for the first time.

This approach pollutes the cache less than the previous one. Indeed, in the previous approach, every instrumented store involves loading the task ID, irrespective of whether the latter will be buffered. In this approach, instead, every instrumented store only involves loading the *Write* bit into a register. The task ID is only loaded if it needs to be buffered.

The disadvantage of this approach, however, is that it needs all the hardware support of the previous one plus an extended load. An *Extended Load* is a load that loads a variable into a register and its *Write* bit into another one. This second register is immediately checked in software. Based on the result, we can save it or not.

The resulting per-store instrumentation using extended loads is shown in Figure 6.5-(b). In the figure, we use *xlw Ri Rj Addr* for an extended load that loads the *Write* bit into register *Ri*.

Using Predication

The default implementation of the two mechanisms described uses a branch instruction (Figures 6.5-(a) and (b)). Branches are undesirable because they can limit ILP. We now show how to eliminate the branches with a simple and a more advanced use of predication [PS91]. In our examples, we use the extended load, although the task ID approach can be similarly used.

In a simple use of predication, we load the *Write* bit using an extended load and then set a predicate register if the bit is zero. After this, we predicate all the buffering instructions on this predicate register. Figure 6.6-(a) shows the code. In the figure, *pred_eq* sets predicate *p1* to true unconditionally if *Write* is zero, and all the instructions that follow it are predicated on *p1*.

```
; r2 = address in memory to insert the record
; offset(r3) = address of the variable to write
; r5 = ID of the executing task

    xlw        r6, r1, offset(r3)   ;Write bit goes to r6
    pred_eq    p1_U, r6, 0          ; set p1
    ; following instructions predicated on p1
    addu       r4, r3, offset (p1)  ; insert as usual
    sw         r4, 0(r2) (p1)
    lh_TID     r4, offset(r3) (p1)
    sw         r4, 4(r2) (p1)
    sw         r1, 8(r2) (p1)
    sh_TID     r5, offset(r3) (p1)
no_insert:
```

(a)

```
; p2 is true if the task is spec ulative
; following two instructions predi cated on p2

    xlw        r6, r1, off set(r3) (p2)
    pred_eq    p1_U , r6, 0 (p2)
    ; following instructions predicated on p1
    addu       r4, r3, offset (p1)     ; insert as usual
    sw         r4, 0(r2) (p1)
    lh_TID     r4, offset(r3) (p1)
    sw         r4, 4(r2) (p1)
    sw         r1, 8(r2) (p1)
    sh_TID     r5, offset(r 3) (p1)
no_insert:
```

(b)

Figure 6.6. Eliminating the branches with the use of simple (a) and more advanced (b) predication.

With this approach, the branch disappears. Every instrumented store involves the execution of at least the two unpredicated instructions. The predicated instructions are also fetched. If the predicate is true, they execute normally. Otherwise, they are prevented from modifying any state when they execute or, if the predicate is false before they issue, they are not even issued. The result is a potentially faster execution.

A more advanced use of predication additionally uses the information of whether or not the task is speculative. The non-speculative task does not need to save an entry in the history buffer because its updates are safe. Consequently, we set up a second predicate *p2* to be true if the task is speculative. We use *p2* to predicate the first two instructions of Figure 6.6-(a), which include the setting of *p1*. The rest of the instructions

are predicated on *p1*. The resulting code is shown in Figure 6.6-(b). With this approach, a non-speculative task does not have to execute a single instruction: *p2* is false and, as a result, *p1* too.

The setting of *p2* proceeds as follows. Typically, when a task starts, its *p2* is 1 because the task is speculative. At a certain time, all its predecessor tasks finish and commit, and the task becomes non-speculative. At this point, the task must be notified, so that it can clear its *p2*. An obvious way of notifying the task is to send it an interrupt. Consequently, when a task finishes, if it is able to advance the commit point, it can send an interrupt to the new non-speculative task.

Unfortunately, this approach has significant overhead. It should not be used for workloads with largely-balanced tasks: a given task spends little time as non-speculative and processing an interrupt is too costly relative to the savings in buffering. This approach should only be used when there is significant imbalance: some tasks spend much time as non-speculative and can benefit significantly from not having to save records in the history buffer. It is therefore important to identify situations with high load imbalance with low overhead.

We use two simple ways to identify load imbalance. The first one is to check the size of the window of uncommitted tasks. Specifically, when a processor finishes a task, it checks the size of the window. If it is larger than a threshold, it sends an interrupt to the non-speculative task so that the latter can clear its *p2*.

A second way to estimate if there is load imbalance is to detect if a task needs to allocate extra sectors for its history buffer. If a task needs more sectors, it probably means that it is larger than the average task and that there is load imbalance. Consequently, in the code that allocates new sectors, we can insert code for a task to check if it is the current non-speculative task and, if so, clear its *p2*. This approach is interrupt free.

6.4.5 Exposed Loads that Need to Be Buffered

When we insert a record in the history buffer we are saving local versions of variables before they are killed. While versions are usually killed with writes, there is one case where a local version may be killed with a load: an exposed load. An exposed load can bring into a task a version of a variable generated by another task, therefore killing the local version. Consequently, we may need to buffer the previous version before exposed loads.

Fortunately, not all exposed loads require that we save a record. We only need to buffer when the exposed load kills a version generated locally that has not been saved

locally yet. This is the case where, at the time of the exposed load, $PmaxR1st <= PmaxW$. With this condition, we are eliminating the case where the local version was brought in from the outside with another exposed load and has not been written since ($PmaxR1st > PmaxW$).

In practice, instrumenting the code to perform this buffering is hard. The reason is that it is often very difficult for the compiler to distinguish exposed loads from the other loads in these hard-to-analyze codes. Often, the only safe approach is to conservatively instrument a large fraction of the loads, which induces high overhead.

Consequently, we choose not to instrument any loads in the code. Instead, we note that the local dependence-detecting hardware engine already detects exposed loads that send a request to the home to get a new version (Section 6.2). Consequently, when this happens, we also have the engine send an interrupt to the processor. In the processor, a software handler checks the previous condition and saves the version if the condition is true.

Clearly, if these interrupts happened frequently, this scheme would be impractical. In practice, there usually are very few exposed loads that are propagated to the home to find a version. The reason is that these transactions, which involve a transfer of information from one task to another, constitute the RAW dependences. If they were not rare, unless the RAW dependences were between far-off tasks, there would likely be frequent violations. The resulting squash overhead would seriously hurt speedups anyway.

Note that we are not including in this case exposed loads to variables that have only been read, not written. Such loads may be common, for example when there are read-only data structures. However, exposed loads to these variables do not trigger a transaction to the home to get a version. Instead, exposed loads to lines marked as not written by any processor so far are not propagated to the home memory; the local copy of the page is accessed. Not receiving an interrupt in this case is exactly what we want: there is no need to buffer anything because no version gets killed.

6.4.6 Other Issues

The only problem left here is how to back up the *initial* value of the shared element. The first time that a thread makes a private copy of the page, the processor where this thread executes owns the master copy. The Operating System marks in the home which processor is the master, and initializes its page of task IDs with 0; for the rest of processors, the page of task IDs will be initialized with -1. When a thread overwrites the initial value, it will buffer the initial task ID from its local page, 0 or -1. In case of an out-of-order RAW

dependence, if during the recovery procedure the version with the task ID -1 arrives to the home, the home will know that the version is not reliable and it will copy the version from the master processor. In the master processor, the version can be found in the local memory or in an entry of the log with a 0 in the task ID.

A final issue related to using task IDs to filter writes has to do with page faults. Recall from Section 6.4.1 that first-time speculative accesses to a page of shared data trigger special events, including the allocation of task ID pages and of local copies of the data page. Now that stores are preceded by task ID accesses with the same virtual address, task-ID loads may be the first ones to access the virtual numbers of pages with speculative data. Consequently, the OS must be modified to support the same allocation and copying operations in a faulting *lh_TID* instruction as in a faulting speculative load or store.

6.4.7 Correct Interleaving

A final issue in MHB insertion is that the algorithm has to guarantee that the value buffered is the same as the value overwritten. Otherwise, correct recovery may be impossible. This issue is not a problem in a speculation protocol like ours. Our protocol allows different versions of the same variable to exist in the different caches, and each processor buffers its *local* version, unaffected by another processor's write. However, in protocols that allow only a single version of each variable like [ZRT98], this is a problem.

In [ZRT98] speculative versions are shared between all the processors, as a regular coherent variable. As an example, Figure 6.7 shows two tasks that speculatively store to the same variable. The dots plus the logging load represent the instructions that insert a record in the log. Depending on the timing of the execution, it is possible that the two logging loads read the same value, which is then stored in the two logs. However, the store in *Task(i)* may invalidate the copy in *Task(i+j)*, forcing the store in *Task(i+j)* to re-request the data and overwrite the value generated by *Task(i)*. The MHB in Task(i+j) would then have a wrong value.

For these single version speculative protocols, where correct interleaving is not guaranteed, we offer two solutions. The first one is to use speculative load instructions in place of the plain loads for buffering. When these speculative loads execute the hardware performs dependence checking. Consider Figure 6.7 again. Using speculative loads guarantees that, for each task, the saved data is the same as the overwritten one. The reason is that the load that saves an entry in the history buffer in *Task(i+j)* must read the value written by the speculative store in *Task(i)*: if the load occurs earlier, an out-of-order RAW is detected and *Task(i+j)* is squashed and restarted. The result, therefore, is a full serialization of the two pairs of accesses.

```
        Task i                                  Task i + j
          .                                       .
          .                                       .
lw  r1, offset(r2)   ; buffering load    lw  r1, offset(r2) ; buffering load
          .                                       .
sw  r3, offset(r2)   ; speculative store
                                         sw  r3, offset(r2) ; speculative store
```

Figure 6.7. Example that may result in incorrectly-buffered information.

This approach is simple and does not require any extra hardware support. However, its main disadvantage is that it introduces artificial RAW dependences between speculative stores and later buffering loads. This leads to unnecessary restrictions. Specifically, suppose that the original speculation protocol was able to handle WAW dependence violations gracefully without squashing. This means that *Task(i+j)* and *Task(i)* could have executed out of order. However, by also marking buffering loads as speculative, we introduce a RAW and force *Task(i)* and *Task(i+j)* to execute in order. Finally, note that, although by marking buffering loads as speculative we may introduce exposed loads, single-version protocols do not need special actions for exposed loads: there is only a single version of the variable to access anyway.

Overall, while this first approach effectively ensures atomicity, it has the shortcoming of sometimes causing task squashing and recovery, which is slow (Section 6.3).

The second approach to ensure correct buffering is to replace the speculative stores by enhanced implementations of existing synchronization instructions that also perform dependence checking. We examine the Swap instruction of SPARC processors [WG94] and the Load Linked and Store Conditional of MIPS processors [Pri95]. Consider Swap first, which atomically exchanges the contents of a register with those of a memory location. We use it to perform the speculative store with the usual dependence checking while, at the same time, returning the old value in a register. Because the instruction is atomic, the value returned is the one we are interested in. We then insert a record using the returned value.

For the MIPS primitive, we use a plain load-linked instruction in place of the buffering load. It returns the value from the location that will be overwritten. We immediately follow it with the enhanced store conditional, which takes the place of our speculative store. If another processor has issued a store conditional to the same location in between

our load linked and store conditional, the latter fails and does not perform dependence checking. In this case, we retry starting from the load linked. Otherwise, the store conditional performs the checking and the value overwritten is guaranteed to be the one that was read with the load linked. Again, after the store conditional succeeds, we insert a record using the returned value.

The advantage of this second approach is that it does not introduce any RAW dependences. Therefore, the extra instrumentation will never cause any task squash. A second, related advantage is that, by not making the buffering load speculative, we save some messages on the network required for dependence checking. Unfortunately, the main disadvantage of this approach is that we need to enhance synchronization instructions.

6.4.8 Hardware-Intensive Solution

The operations described in 6.3 were implemented in hardware in [Zha99, ZPG+99] using a hardware History Buffer Controller embedded in the directory module of each node. Next, we compare it against our software-only implementation.

To insert a new record in hardware, the underlying speculation protocol detects the first update to a word and saves a new record into the MHB. For the detection, the *Write* bit in cache, or the *PmaxW* task-ID in the local memory are checked. A message with the old value before being destroyed needs to be sent to the hardware History Buffer Controller in each node. The History Buffer Controller will insert a new entry into the MHB, and will update the corresponding pointers. In this hardware implementation, the MHB is kept in the local memory in each node, which avoids cache pollution. Our software implementation does not require any History Buffer Controller, since all the operations are handled in software. It does not require special hardware to detect the first update to a word in a task either. However, it still requires the hardware that detects when an entry needs to be inserted under an exposed load. On the other hand, since the MHB of our software implementation is considered a plain data structure, it pollutes L1 and L2 caches. We evaluate in our simulations the difference in performance when the MHB is forced to bypass the L1 cache.

Notice, that the hardware management of the MHB adds little overhead to the execution of the program, but has two drawbacks. First, if the history buffer runs out of space, an interrupt must be enforced to request the OS the allocation of additional physical pages. Second, since a hardware implementation can only record the *physical* addresses of the saved variable, the recovery must be performed by exception handlers with privilege to write directly to physical addresses.

On the other hand, we have introduced some changes in the Address Mapping scheme used by the protocol in [Zha99, ZPG+99], since we needed a mechanism to access the task-IDs from software. That protocol uses an Address Mapping Module containing two tables. The first table is used to obtain the physical address in the overflow area and local task-IDs. The second table is used to obtain the shared task-IDs. Both tables are indexed with the shared physical address of the variable, which is also used to tag caches (Appendix A). Our mechanism still needs the second table, but not the first one. We have changed the translation of the virtual address into the local and shared physical addresses, and the memory mapping of the local task-IDs. In particular, using our software scheme, the OS must place the page of data speculatively accessed in an even-numbered physical page number, so that the next page is used for the task-IDs. After the TLB translation, our proposed solution needs no extra translation to access the local data or task-IDs. However, the access to the shared data in the home node and the incoming transactions willing to access the local task-ID, or searching for a particular version in cache or overflow area will require and extra translation through the Address Translation Module of Figure 6.2 and explained in Section 6.4.1. Finally, notice that the protocol in [Zha99, ZPG+99], only allocates a page of data in the overflow area in the local node if the processor's node speculatively writes to a cache line of the shared page, and displaces it. Thus, pages of data that a processor reads but does not write, do not need to be allocated in the local node. On the contrary, with our scheme under the first speculative access (read or write) to a shared page of data, the requesting processor must place a new page of data in its node's memory, or at least reserve its corresponding even page. In any case, we do not see this to be an important issue, because the page with the local task-IDs needs to be placed under the first access in both schemes.

6.5 Evaluation Methodology

We evaluate the performance of the software multi-version buffering scheme for a FMM system using a history buffer like that one we have just described. For that we use simulations driven by several applications. In this section, we describe the simulation environment and the applications.

6.5.1 Simulation Environment

We use an execution-driven simulation system based on MINT [VF94a] to model in detail a CC-NUMA with 16 nodes like the one used in the previous chapter. We repeat here its main characteristics and give some additional details of the software buffering scheme.

Each node of the CC-NUMA machine contains a fraction of the shared memory and

directory, and a 4-issue dynamic superscalar. The processor has a 64-entry instruction window and 4 Int, 2 FP, and 2 Ld/St units. It supports 8 pending loads and 16 stores. It has a branch penalty of 8 cycles and has 64 Int and 64 FP rename registers. It also has a 2K-entry BTB with 2-bit saturating counters. Each node has a 2-way 32-Kbyte L1 D-cache and a 4-way 512-Kbyte L2, both with 64-byte lines and a write-back policy. We use a small L2 because the applications are small. The nodes are connected with a 2D mesh. The average no-contention round-trip latencies from the processor to the on-chip L1 cache, L2 cache, memory in the local node, and memory in a remote node 2 and 3 protocol hops away are 2, 12, 75, 208 and 291 cycles, respectively. Contention is accurately modelled in the whole system, except in the global network where we only model the source and destination ports. The caches are kept coherent with a release-consistent protocol like that of DASH. For speculation, we use the protocol of Section 6.2. Pages of shared data are allocated round-robin across the nodes. We choose this allocation because our applications have irregular access patterns. Private data are allocated locally.

In the evaluation, we simulate all software overheads, including allocation and recycling of history buffer sectors, and the dynamic scheduling and committing of tasks. We wrote software handlers for parallel recovery after a dependence violation (Section 6.3), deciding whether an exposed load needs to be buffered (Section 6.4.5), and retrieving data from the history buffer (Section 6.3). A processor receiving an interrupt is penalized with an additional 300 cycles over the running of the software handlers for miscellaneous overheads, and a page fault costs 4,000 cycles.

6.5.2 Applications

We use a set of scientific applications where a large fraction of the code is not fully analyzable by a parallelizing compiler. These applications are: *Apsi* from SPECfp2000 [Hen00], *Track* and *Bdna* from Perfect [B+89], *Dsmc3d* from HPF-2 [DSH94], *P3m* from NCSA, and *Tree* from [J. 94]. We use the Polaris parallelizing compiler [BDE+96] to identify the non-analyzable sections and prepare them for speculative parallelization. The source of non-analyzability is that the dependence structure is either too complicated or unknown because it depends on input data. For example, the code often has indirect indexing to arrays. The code also has sections that have complex control flow, with conditionals that depend on array values and jump to code sections that modify the same or other arrays. In these sections, Polaris marks the accesses to non-analyzable data, which we call speculative references and will trigger speculation protocol actions. Polaris also identifies other plain updates that may need to be saved into the history buffer and we also instrument them.

Table 6.1 shows the non-analyzable sections in each application. These sections are

loops that a state-of-the-art parallelizing compiler like Polaris cannot decide whether or not they have dependences. The table lists the weight of these loops relative to the total *sequential* execution time of the application (%Tseq), with I/O excluded. This value, which is obtained on a single-processor Sun Ultra 5 workstation is on average 51.4%. The table also shows the number of invocations of these loops during program execution, the average number of iterations per invocation, and the average number of instructions per iteration. Finally, the last three columns show the weight of several types of references: speculative reads and speculative writes as a percentage of the *total references* in the section, and instrumented writes (including both speculative and not) as a percentage of the *total instructions* in the section. All counts are dynamic. Note that all the data presented in the evaluation (Section 6.6), including speedups, refer only to the code sections in the table.

Appl	Non-Analyzable Sections (Loops)	% of Tseq	# of Invoc	Iters per Invoc	Instruc per Iter	Spec Refs / Total Refs (%)		Instrum Wr / Total Instructs (%)
						Rd	Wr	
P3m	*pp_do100*	56.5	1	97336	69165	11.1	2.1	0.8
Tree	*trwalk*	92.2	41	4096	28746	2.9	2.9	0.8
Bdna	*actfor_do240*	44.2	1	1499	103339	7.1	6.8	2.3
Apsi	*run_do[20,30,* 40,50,60,100]	29.3	900	63	102639	49.4	33.4	11.6
Track	*nlfilt_do300*	58.1	56	126	22308	0.3	0.2	0.4
Dsmc3d	*move3_goto100*	41.2	80	46777	5442	0.0	0.0	1.2
Average		51.4	180	24470	55273	11.8	7.56	3.2

Table 6.1. Application characteristics. In *Track* and *Dsmc3d*, the data corresponds to unrolling the loop 3 and 15 times, respectively.

The recovery exception handler invoked after an out-of-order RAW dependence (Section 6.3) is executed once in 3 loop invocations (once in each) in *Track*, and an average of 30 times in each of the 80 loop invocations in *Dsmc3d*. The retrieval exception handler (Section 6.3) is not executed in any of the applications: although in-order RAW dependences appear in *Dsmc3d*, the requested version is a last version and is found in the cache instead of in the history buffer.

6.6 Evaluation

To evaluate our proposed software buffering scheme that supports multiple versions per cache, we perform three experiments: impact on execution time of the software buffering scheme (Section 6.6.1), analysis of implementation overheads in the advanced scheme (Section 6.6.2), and alternative policies for the advanced scheme (Section 6.6.3).

6.6.1 Impact of Software Buffering on Execution Time

We compare the execution time of our applications under two different buffering schemes. One is our advanced scheme (*Sw*) from Figure 6.5-(a) that filters first stores using task IDs. The another one is *Hw* from Section 6.4.8, which implements the buffering in hardware without any of the software overheads presented in Section 6.4. In *Hw*, all the operations on the history buffer are performed with no instruction overhead or cache pollution. To minimize L1 pollution in *Sw*, the history buffer is L1-uncacheable, and recycled. L1-uncacheable means that the history buffer is stored in the L2-cache and memory, but never in L1. For recycling, a processor attempts to recycle local history buffer records before it starts a new task. It reads the variable that records the last-committed task to know what records to recycle.

Figure 6.8 compares the execution time of these two systems. The scheme from Section 6.4 is *Sw*, and the one implemented in hardware is *Hw*. For each application, the bars are normalized to *Hw* and broken down into execution of instructions (*Useful*), waiting on data, control, and structural pipeline hazards (*Hazard*), synchronization (*Sync*), and waiting on data from the memory system (*Memory*). A sixth category, measuring the execution of software handlers for data recovery and retrieval, after data dependences, is too small to be seen. Finally, the numbers on top of each bar show the speedup relative to the sequential execution of the code, with all the application data placed in the local memory of the single active processor.

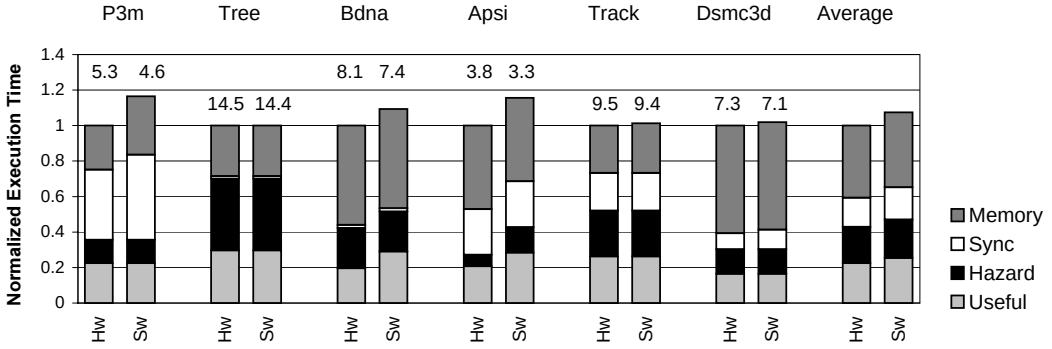

Figure 6.8. Execution time under different schemes for a 16-node multiprocessor.

Comparing *Hw* to *Sw* we can see the overhead introduced by our software implementation of the history buffer. This overhead is more noticeable in P3m, Apsi and Bdna. This overhead comes from the more instructions executed (*Useful* time) and the higher memory stall due to the history buffers (*Memory* time). As a result, *Hazard* also increases.

Overall, our implementation of multi-version buffering is very effective: it only introduces an overhead of 7% when compared to a similar only-hardware scheme, and removes

most of the hardware complexity. In the following section, we will be examining in detail the sources of this overhead.

6.6.2 Implementation Overheads of Advanced Buffering

To gain further insight into our software multi-version buffering scheme in Section 6.4, we now characterize its sources of overhead, namely additional instruction execution and memory accesses. While all history buffer management operations add up to this overhead, the bulk of the overhead is induced by the record insertion instructions added before a store that may be a first store to a variable in a task. Such a store we call an *instrumented* store.

The instructions that our algorithm adds to each instrumented store are shown in Figure 6.5. At run time, two different cases are possible. If the store turns out not to be a first store, only 2 instructions are executed, (including the check for sector overflow), of which 1 accesses memory; if the store is a first store, 10 instructions are executed, of which 6 access memory.

If we compare *Sw* to *Hw* in Figure 6.8, we see the effects of these overheads. Instruction overhead directly reflects into an increases in *Useful* time. Memory access overhead is more subtle, due to the ability of superscalar processors to hide memory latency. Memory access overhead plus some contribution of instruction overhead reflect into an increase in *Hazard*, *Sync*, and *Memory*. To explain these changes, we now analyze the two overheads separately.

Instruction Execution Overhead

The last column of Table 6.1 helps explain the impact of instruction execution overhead. The column shows the number of instrumented stores as a fraction of all instructions executed in the program before instrumentation. Most applications have a small fraction. The exceptions are *Apsi* and, to a lesser extent, *Bdna*, where the fractions are 11.6% and 2.3%, respectively. These are the applications that show an increase in *Useful* time for *Sw* in Figure 6.8.

However, instrumented stores have different costs. Figure 6.9 classifies all the stores in the code into three classes: instrumented and first (*Inst.First*), instrumented and not first (*Inst.Non_First*), and not instrumented (*Non_Inst*). The figure shows the relative weight of each class. The figure includes both speculative and non-speculative stores since the latter may also get instrumented.

Figure 6.9 shows that attempting to identify true first-stores at run time paid off: *Apsi*, *Tree*, and *P3m* all have many instrumented stores that are proved not to be first stores

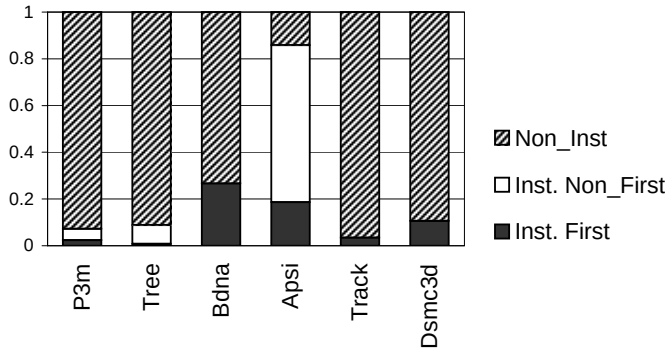

Figure 6.9. Breakdown of dynamic stores.

at run time. In such cases, much overhead is eliminated. Furthermore, the figure explains why the *Useful* time for *Bdna* in Figure 6.8 increases proportionally more than for *Apsi*, despite the lower fraction of instrumented instructions in Table 6.1: the instrumented stores in *Bdna* are always of the expensive class.

Memory Access Overhead

This overhead is due to accesses to two new objects, namely history buffer and task IDs. Focusing on the history buffer first, Table 6.2 shows the average size of the history buffer entries created per task (Column 2) and the number of tasks per history buffer (Columns 4-5). We will examine the other columns later. The column with the history buffer size per task is labeled *Filter* to indicate that our algorithm filters the first store (Section 6.4). We can see that for several applications, the average task needs 20-40 Kbytes. The columns with the number of tasks per history buffer are labeled *Recycle* because obsolete sections of the history buffer are dynamically recycled when a processor grabs a new task. We show both the average of all processors and the maximum. These values are obtained by taking a snapshot every time that a processor finishes a task. Note that if we multiply either of these values by the size of the history buffer per task, we often get large memory footprints.

Appl	History Buffer Size per Task (Kbytes)		# Tasks in the History Buffer of a Processor			
			Recycle (Measured at End of Task)		No Recycle (Measured at End of Code)	
	Filter	All	Maximum	Average	Maximum	Average
P3m	2.5	7.9	100	50.0	132	80.3
Tree	0.4	3.5	8	2.0	70	64.0
Bdna	39.8	39.8	4	1.6	99	93.7
Apsi	40.0	184.0	4	1.8	5	3.9
Track	1.2	1.2	4	1.3	11	6.4
Dsmc3d	1.0	1.0	3	1.1	1136	489.3
Average	14.1	39.6	20.5	9.6	242.2	122.9

Table 6.2. History buffer statistics. The experiments are for 16-processor runs. The records of the history buffer are 16 bytes.

Overall, although the history buffer can grow large, it is primarily accessed through

stores to consecutive locations when a record is inserted. Since write latency can be easily hidden, we do not expect the history buffer to affect *Memory* stall much.

The second type of object accessed is the task IDs. Our algorithm requires that task IDs are loaded at every instrumented speculative store irrespective of whether or not the store turns out to be a first store. Table 6.3 shows the contribution of these task ID loads to the total L1 read miss rate. In the table, we focus on the columns under *History Buffer L1 Bypass*, and among those, the first and the third one, which are labeled *Task-ID Filtering*. These two columns correspond to our software algorithm *Sw* that filters with task IDs (Section 6.4.4) They show the total L1 read miss rate of the application and the percentage of L1 read misses that are due to task ID loads, respectively. The other columns in the table will be discussed later. The data show that task ID loads usually contribute with only 10% or less of the read misses. The exception is *Apsi*, where they contribute with 40.4%, but the total miss rate of *Apsi* is only 1.01%. Overall, we do not expect task ID accesses to affect *Memory* stall much, either.

Appl	History Buffer L1 Bypass				No History Buffer L1 Bypass			
	Total L1 Rd Miss Rate (%)		L1 Rd Misses from Task-ID Rd (%)		Total L1 Rd Miss Rate (%)		L1 Rd Misses from Task-ID Rd (%)	
	Task-ID Filtering	Xload Filtering	Task-ID Filtering	Xload Filtering	Task-ID Filtering	Xload Filtering	Task-ID Filtering	Xload Filtering
P3m	8.86	6.20	8.4	0.6	8.86	6.24	7.7	0.6
Tree	21.02	21.11	0.0	0.1	21.11	21.11	0.0	0.0
Bdna	5.03	5.03	5.4	5.4	5.12	5.12	5.3	5.3
Apsi	1.01	0.60	40.4	32.6	2.01	0.80	22.2	29.3
Track	1.29	1.29	9.3	9.3	1.37	1.37	9.8	9.8
Dsmc3d	1.90	1.90	0.0	0.0	1.94	1.94	0.1	0.1
Average	6.51	6.02	10.5	8.0	6.73	6.09	7.5	7.5

Table 6.3. Effect of task IDs loads on the L1 miss rate. The experiments correspond to 16-processor runs. *Task ID* and *Xload* stand for task ID and extended load, respectively. The history buffer is recycled in all the cases.

These expectations are confirmed by Figure 6.8. The figure shows that the *Memory*, *Sync*, and *Hazard* times increase only modestly as we go from *Sw* to *Hw*. Therefore, we conclude that the additional memory requests induced by accesses to the history buffer, and task IDs do not hurt performance much.

6.6.3 Alternative Designs for Software Logging

Based on the analysis thus far, we now assess alternative designs for software buffering. Specifically, we consider four optimizations: reducing hazard overhead by eliminating first-store filtering, reducing memory overhead caused by recycling the history buffer or

bypassing L1, and reducing memory overhead associated with task-ID loading by using extended loads.

Eliminating First-Store Filtering

Our default software multi-version buffering algorithm filters the first store to reduce the number of history buffer insertions (Section 6.4.4). We now eliminate the branch in Figure 6.5-(a), therefore saving a record in the history buffer before all instrumented stores. This change eliminates one instruction when the store is actually a first store and, in all cases, potentially reduces control hazards. However, when the store was not a true first store, more instructions are executed that before and, in addition, longer buffers are generated.

Figure 6.10 shows the execution time with this new algorithm, which we label *All* because all instrumented stores create a record. The system is compared to our default *Sw* system of Figure 6.8, which we have normalized and re-labeled *Filter*. From the figure, we see that *All* is no faster and, in the case of *Apsi* much slower, than *Filter*. For the applications where, according to Figure 6.9, many instrumented stores are *Inst.Non_First*, *All* is slower. There are several reasons for it. First, since more instructions are executed, *Useful* is higher. In addition, more memory accesses are performed, some of them to buffer values. In particular, the third column of Table 6.2 (labeled *All*) shows the new size of the history buffer per task. The history buffer is now over 4 times its filtered size in *Apsi*. Finally, *Hazard* in Figure 6.10 does not decrease noticeably for the applications with only *Inst.First* (*Bdna*, *Dsmc3d*, and *Track*) and it increases for the other applications due to increases in other categories. Overall, we conclude that we should use filtering.

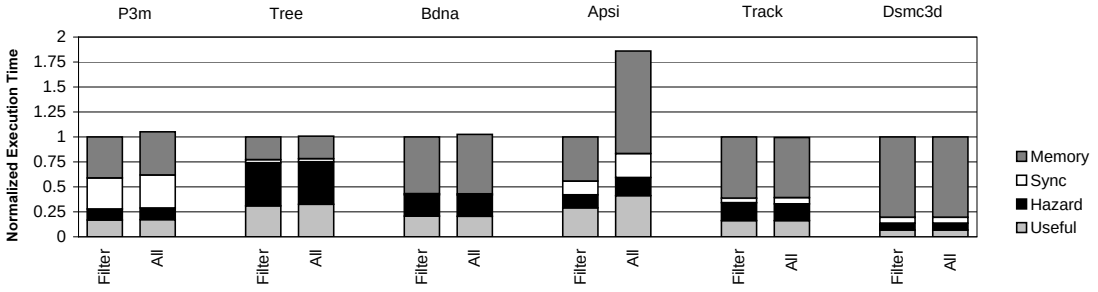

Figure 6.10. Effect of eliminating first-store filtering.

Recycling History Buffer and Bypassing L1

Since the history buffer may grow quite large, (*Memory*) time may decrease if history buffer bypass the L1 cache and/or their space is recycled.

To reduce the memory overhead caused by accesses to the history buffer, we now consider two optimizations that minimize L1 pollution: recycling the history buffer space at run time and forcing history buffer accesses to bypass L1. History buffer records can be recycled when the task that generated them commits. The space can then be used for other records. This optimization reduces the history buffer footprint at the expense of some bookkeeping. The last two columns of Table 6.2 show that recycling greatly reduces the history buffer footprint for all applications. The columns show the average and maximum number of tasks that keep state in an individual history buffer. To measure these parameters, we take snapshots every time that a processor finishes a task. These parameters are much smaller than the ones in the columns to their left, obtained at the end of the code without recycling.

Forcing history buffer accesses to bypass L1 should reduce L1 pollution without noticeably slowing down the processor because write latency can be usually hidden. As it is, we find that the resulting L1 miss reduction is small. This can be seen from the section of Table 6.3 labeled *L1 Bypass*, which shows overall miss rates (leftmost column) not much different from those under *No L1 Bypass*. Only *Apsi* has noticeable miss rate change, from 2.00% to 1.00%.

To see the effect of these optimizations on the execution time, we run the applications with 16 processors under all four scenarios (Figure 6.11). We run them with or without L1 bypass for the history buffer (*By* and *NoBy*, respectively), and with or without history buffer recycle (*Rec* and *NoRec*, respectively). For each application, the bars are normalized to bypass and recycle (*By.Rec*), which is our baseline *Sw* from Figure 6.8. While we use the default filtering algorithm for buffering in all applications (*Filter*), for comparison purposes, we also show bars with the *All* algorithm of Figure 6.10 for *Apsi*. *Apsi* was the only application where the choice of algorithm in Figure 6.10 made a difference.

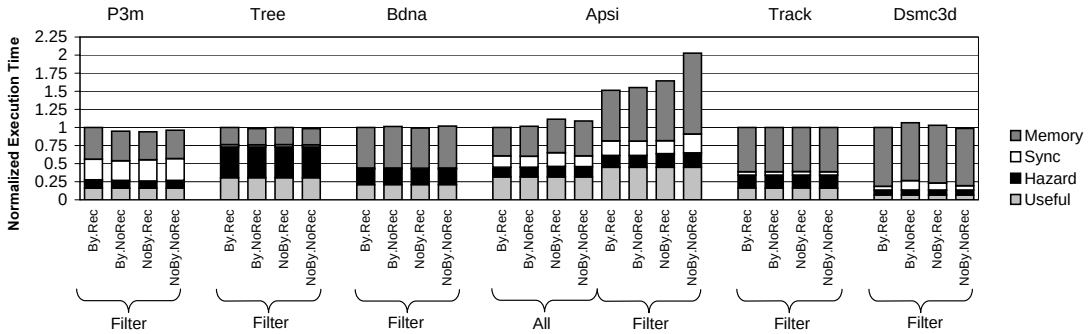

Figure 6.11. Effect of log recycling and L1 cache bypassing for logs.

The figure shows that, for a highly-tuned software buffering algorithm like our baseline *Filter*, history buffer recycling or L1 bypassing are not very useful. Neither optimization

reduces *Memory* much. The only exception is *Apsi*, where L1 bypassing is effective. The reason is that tasks in *Apsi* generate large history buffers (Table 6.2) and, unlike in *Bdna*, spread their buffering along their execution, maximizing potential L1 pollution. However, in some cases like *P3m*, the combination of both recycling and bypassing is bad: recycling involves pointer access for bookkeeping, which may be slow if they need to reach L2.

We note, however, that if frequent buffering is required, these optimizations can be useful. This can be seen for the less optimized *All* algorithm on *Apsi*. Either L1 bypassing or history buffer recycling reduces the pollution enough to speed up the execution of *Apsi* 20% or more. The combination of both techniques produces marginal gains.

Using Extended Loads

To reduce the memory overhead induced by task-ID loading, we can use extended loads (Section 6.4.4). With extended loads, instrumented stores add the same number of additional instructions and memory accesses as with task IDs. However, the task ID is loaded only if it is needed (i.e. a record needs to be created). Otherwise, only the data to be written is loaded, therefore reducing cache pollution. The result is a lower miss rate.

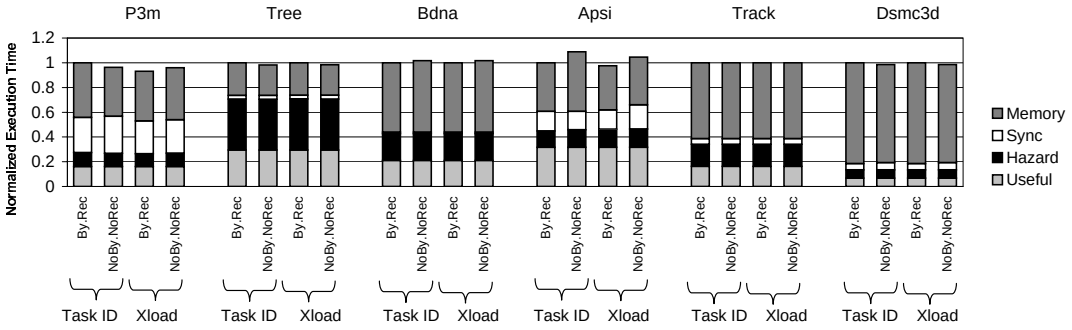

Figure 6.12. Comparing filtering with task IDs to filtering with extended loads.

Table 6.3 compares the L1 miss rate using task IDs (*Task ID* columns) and extended loads (*Xload* columns). Any difference between the schemes is likely occur only for applications where a large fraction of the misses come from task-ID loading (Columns 4-5 and 8-9) and where many instrumented stores turn out not to be first store (Figure 6.9). The only such application is *Apsi*. From the table we see that, if we do not use L1 bypassing, using extended loads in *Apsi* reduces the miss rate from 2.00 to 0.8%.

Figure 6.12 shows the resulting impact on the execution time of a 16-processor system. The figure compares the default system, which uses task IDs for first-store filtering (*Task-ID*) to a system that uses extended loads (*Xload*). For each case, we show an environment with L1 bypass and log recycle (*By.Rec*) and to match the data in the table with L1 bypass and history buffer recycle (*NoBy.Rec*). From the figure, we can see that *Apsi*

benefits slightly from using extended loads. The gains are larger when there is no L1 bypassing, since we can then remove more L1 pollution with extended loads in that case. Snce using extended loads does not improve execution time noticeably, we choose not to use the special support required for extended loads.

6.7 Summary

To support buffering of speculative state under speculative parallelization an approach is the one taken by FMM systems, where main memory keeps the future state while a history buffer keeps previous versions.

The main contribution of this chapter is the design, implementation and evaluation of a software only multi-version buffering scheme for a FMM system that uses a history buffer and runs on top of a speculative protocol. This approach is inexpensive, and it delivers high performance. Using simulations of a 16-processor CC-NUMA, the overhead of our proposed implementation for multi-version buffering is only 7% of a similar only hardware approach. We also found that bypassing the cache or recycling the log was less important than maximizing logging efficiency.

References

[B+89] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[BDE+96] 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. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[DSH94] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[ELPS98] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 140–151, February 1998.

[GPL+00] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Buffering State with Software Logging in Scalable Speculative Parallelization. Technical Report CSRD-1581, University of Illinois at Urbana-Champaign, July 2000.

[GPL+01] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and Josep Torrellas. Software Logging under Speculative Parallelization. In *Workshop on Memory Performance Issues hold in conjunction with ISCA'01*, July 2001.

[Hen00] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[HK99] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proc. of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 171–181, January 1999.

[J. 94] J. E. Barnes. Treecode. Technical Report, Institute for Astronomy, University of Hawaii, 1994.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. of the 1995 Int'l Conference on Parallel Processing (ICPP'95)*, pages 1–10, August 1995.

[Pri95] C. Price. *MIPS IV Instruction Set, Revision 3.2.* MIPS Technologies, Mountain View, CA, September 1995.

[PS91] J.C. Park and M.S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlet Packard Laboratories, Palo Alto, CA, May 1991.

[VF94] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[WG94] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9.* Prentice Hall, Englewood Cliffs, N.J., 1994.

[Zha99] Y. Zhang. Hardware for Speculative Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, May 1999.

[ZPG+99] Y. Zhang, M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas. A Framework for Speculative Parallelization in Distributed Shared-Memory Multiprocessors. Technical Report CSRD-1582, University of Illinois at Urbana-Champaign, July 1999.

[ZRT98] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 162–174, February 1998.

Chapter 7

Conclusions and Future Work

In this thesis we have considered mechanisms to improve the performance obtained by a multiprocessor system executing in parallel threads or tasks from a single application. We have also studied mechanisms to enable parallel execution of codes that are hard to analyze, but have a significant degree of parallelism. In particular we have characterize several low-cost prefetching mechanisms, and have selected the combinations that deliver the higher speedups for a broad design space of bus-based multiprocessor systems. We have also proposed an architectural support to parallelize the reduction operations that appear in many scientific codes. Our support is particularly well suited for the reductions that appear in sparse and dynamic codes. Finally, we study the challenge of buffering speculative state when using speculative thread-level parallelization. We introduce a taxonomy of speculative state management, and evaluate the performance and cost of all the design points using a common speculative framework. Next, we summarize our main conclusions and present future work.

7.1 Conclusions

7.1.1 Hardware Prefetch

The increasing gap between the frequency of the processor and the time to access main memory limits performance for many applications. This limitation is even higher in a multiprocesor system, where the latency to access memory is larger.

A solution to this problem that many researchers have studied is data prefetch. In this thesis we take a similar approach. For that, we have first evaluated the characteristics of parallel applications related with the address regularities we are able to capture. Then,

we have evaluated four hardware prefetchers that are able to detect the corresponding patterns.

We have characterized a subset of the SPLASH-2 applications. We have measured the percentage of loads that follow a scalar, sequential, stride, linked list or index list pattern. We have also measured the sequence lenghts. This characterization has been done varying the number of processors from 1 to 32. Our results show:

- Dominance of sequential pattern, with meaningful presence of stride pattern for some particular applications. The average percentages of scalar, sequential and stride patterns are 30%, 40%, and 11%, respectively. The remaining 9% correspond to loads that follow a pattern that we cannot recognize.
- Practical absence of single-load linked list and index list traversal, with percentages falling always below 1%.
- Existence for some particular applications of a significant fraction of loads that follow a pattern different of the ones that we tried (56%, 30%, and 35% for Radix, FMM and Radiosity, respectively). Thus, new pattern detectors could be useful for these applications.
- Average reduction of 15% in the number of loads that can be predicted when the learning time is taken into account. This reduction is significant (45.9%) in LU-non, and it is due to the loss in capturing the sequential pattern.
- Persistance of the patterns as the number of processors changes. The only exceptions are Cholesky and Radix, which are applications with a higher communication-to-computation ratio.
- Sequence lenghts also remain constant when the number of processors changes.

After this characterization we have analyzed four cost-effective hardware prefetching mechanisms: Sequential Tagged (Ts), Load Cache (LC), Load Cache with On Miss Insertion (LCm), and Load Cache with On Miss Insertion plus Sequential Tagged (LCms). These mechanisms have been evaluated in a bus-based multiprocessor system. For the evaluation we have varied the number of processors (from 1 to 32). We have used two different sets of cache-sizes for the two levels of cache of our simulated system. The cache-sizes of the two sets have been chosen to compare two different scenarios: one that strongly presses the memory system, and another that does not press it. The results of this evaluation reveal that:

- The performance obtained with a particular prefetch mechanism depends more on the application than on the cache size.

- A LCm prefetcher with 16 entries performs always better than a LC with a traditional insertion policy, and the same number of entries.
- LCms and LCm appear to be the bests of the evaluated prefetchers. Taking the best of LCm and LCms prefetchers, the average reduction in execution time relative to the same system without prefetch are of 9.97%, 5.03%, 2.54% and 0.43% for 4, 8, 16 and 32 processors respectively.
- As the number of processors increase, the performance benefit of a system with prefetch reduces when compared to the same system without prefetch: in 72% of the cases for 32 processors, and 50% for 16 processors, the performance of a system with prefetch is lower than the performance of its equivalent one without prefetch. Looking at each particular prefetcher, LCm, LCms and Ts yield increase the execution time in 8.3%, 9%, and 12% of the cases.
- Since sequential prefetch can degrade performance as the number of processors increase, we suggest that the sequential mechanism should be disconnected in software if it proves to be harmful for a large number of processors.

7.1.2 Parallelizing Reductions

Parallelization of reduction is a critical transformation in many loop-parallel scientific codes. Several software proposals try to exctract a significant parallelism from reductions. However, it is usually hard to find the appropiate transformation. This is specially true in the case of sparse, and dynamic applications. In addition, conventional reduction parallelization algorithms are not scalable.

We propose a new architectural support to speed-up parallel reductions that we call Private Cache Line Reduction (PCLR). This supports makes reductions scalable when executing in Distributed Shared Memory (DSM) multiprocessors. The proposed support is mostly confined to the directory controller of a DSM multiprocessor, and can be implemented using a hardwired directory controller (Hw), or a programmable directory controller (Flex).

The main idea of PCLR is to use non-coherent cache lines as a temporary private storage, where processors acummulate the partial results of a reduction. When the cache lines are displaced from the cache, their values are accumulated onto the shared reduction value in memory. PCLR eliminates the need for the final merge step that typically appears in the conventional software algorithms that use replicated local buffers. With our proposed support, parallel reduction only needs a final cache flush step that takes a time proportional to the cache size. Our main conclusions are:

- Simulations with 16 processors reveal that the Hw implementation of PCLR achieves an average speed-up of 7.6 for 16 processors, while the Flex one achieves 6.4, and a software-only implementation achieves 2.7.
- Simulations of 1, 4, 8 and 16 processors show that our PCLR mechanism (for both Hw and Flex) scales well as the number of processors increase, while a software-only implementation scales poorly.

7.1.3 State Buffering in Speculative Thread-Level Parallelization

In the context of speculative thread-level parallelization we have focused on the problem of buffering speculative state. Our contributions here are two-fold: a novel taxonomy of approacches to buffer and manage speculative state, and the design of a software-only buffering scheme that works on top of a speculation protocol.

Our taxonomy includes a novel application of the concepts of architectural and future state to memory state. It also classifies schemes based on the support for multiple tasks and versions. We perform a tradeoff analysis and detailed performance evaluation of the different approaches under a single framework for a set of scientific applications. The performance comparison of the different systems with architectural main memory (AMM) reveals:

- The support for multiple tasks and a single version is not cost-effective, since its performance is similar to the one achieved with support for a single task, while its complexity is very close to the one required for multiple tasks and multiple versions.
- The support for multiple tasks is more cost-effective than the support for lazy memory update. The reduction in execution time is higher: 35% versus 25% for the evaluated applications. Furthermore, the hardware complexity for adding multiple tasks and versions is lower.
- The improvement due to multiple tasks and versions, and due to laziness are fairly orthogonal. Indeed, adding laziness to a system supporting multiple tasks and versions reduces execution time by an additional 20%.

We have also compared AMM systems with Future Main Memory (FMM) systems. The most advanced AMM system, the one supporting multiple tasks and versions and lazy main memory update, is very competitive when compared against the most expensive and fastest system with future main memory (FMM) system. The only disadvantage of the AMM sytems when compared to the FMM systems appears when the applications have

sizable working sets, load-imbalance, and possibly mostly-privatization patterns that put stress on the cache hierarchy of the AMM systems.

Our second contribution in this context is the design of a software-only buffering scheme for FMM systems that supports multiple tasks and versions. We have evaluated several of the alternative designs that we propose, and we find that:

- Filtering first stores can significantly reduce the execution time for some applications, and it does not increase the execution of those applications that do not need it. Thus, we recommend to use it.
- If history buffers bypass the L1 cache or are recycled, the overhead reduces significantly if the application requires substantial buffering. If buffering is not that frequent, these alternatives do not seem to be very useful, but since they do not increase the execution time, we also recommend to use them.
- Extended Loads do not reduce execution time much, and since it requires extra support, we recommend not to use them.

We have compared our software-only buffering scheme with a similar hardware-only implementation and we find that this solution is very attractive. It adds only a 7% overhead, and removes most of the hardware complexity. For the comparison we used a software implementation with all the previous recommendations: first stores are filtered, and history buffers are recycled and bypass the L1 cache.

7.2 Future Work

Next we discuss new directions that may guide future research from the work done in this thesis.

When studying prefetching we have found that for some applications there is a high percentage of loads that follow a pattern that we cannot recognize. Thus, new pattern detectors have to be tried. Also, we think that SPLASH-2 applications are very well programmed, and that it would be worthful to extend the work on characterization and hardware prefetch to other parallel programs. On the other hand, companys have released or announced commercial chips that integrate multiple processors with SRAM or even DRAM. In this context of multiprocessors on a chip, new research is needed to evaluate how to take advantage of the system integration, and which is the performance that a new prefetching scheme can deliver. This study should look at new organizations of the

memory or its processor interconnection that could eliminate or reduce the bottlenecks of tightly coupled systems.

As for Speculative Thread-Level Parallelization we think that it is a very promising research area. However, further research is needed to understand how the parallelizing compiler and the speculation hardware interact, so that we can extend the coverage and the efficiendy of speculative parallelization. Lines that seem interesting in this field is the study of how the scheduling of tasks to processors affects the performance, or how to choose the most appropiate size grain of the tasks. Long tasks reduce scheduling overheads, but poses the problem of more work to be squashed in case of an infraction. Thus, the election of the size will largely depend on the memory latencies of the multiprocessor where the application is going to run, and on the dependence infractions rate. Dynamic algorithms may help at finding the most appropiate grain-size. On the other hand, we feel that it is still unclear the potentials of speculative execution for these codes in DSM systems. Therefore, our evaluation needs to be extended to other applications, like integer applications and pointer-based applications, where many more dependences can be expected.

Finally, the work done with the software buffering is a step ahead at reducing the complexity of the hardware protocols. But more work needs to be done in this line. A possible idea would be to use the multithreading possibilities of the new released Simultaneous Multithreading Chips (like the Hyper-threaded Intel Xeon), in order to insert new instructions to check for dependences, buffer speculative state, and so on.

Appendix A

Speculation Protocol for MultiT&MV lazy FMM systems

This appendix explains some important details of the speculation protocol proposed for a Distributed Shared Memory (DSM) multiprocessor system in [Zha99]. The protocol is used for speculative thread level parallelization of codes that the compiler cannot fully analyze. With this approach, the compiler extracts tasks that can be executed in parallel. As tasks execute, the speculation protocol checks for memory dependences across tasks. To do this, the speculation protocol extends the transactions of the coherence protocol. However, while the coherence protocol works at the granularity of cache lines, this protocol works at the granularity of 32 bit words, that in the following we call *variables*. This avoid false dependence detection due to false sharing.

In this protocol, tasks are dynamically scheduled to processors. When a task finishes, it cannot be considered committed until all predecessor tasks have also finished and committed. However, to neutralize load-imbalance, the processor that ran the task does not wait: it goes on to execute the next task. Figure 7.1-(a) shows two finished tasks (F) that cannot commit. Every time a processor finishes a task, it tries to advance the Commit Point (CP). If it succeeds, several tasks can commit at a time, Figure 7.1-(b). In this protocol, committing a task does not involve any data copying or flushing: it simply involves an update to a shared variable called the *Commit Point*.

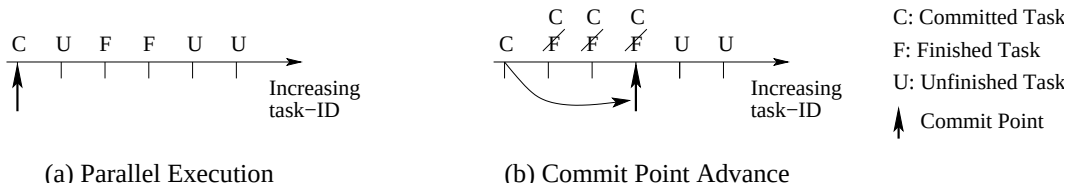

Figure 7.1. Sliding commit of tasks.

This protocol has been proposed in a DSM context, and therefore, when talking about a cache block we will use the terms *home* or *local* node. The *home* node refers to the node in whose memory the block is allocated, while the *local* node refers to the node containing the processor that issues the request. The protocol assumes that data whose access patterns cannot be analyzed at compile time (and thus, can cause data dependences across tasks) are distributed among the memories of all the nodes in the multiprocessor. Private data to each thread are allocated in the memory of the node that contains the processor where

the thread executes. The access to the non-analyzable data is marked with instructions with special operation codes that trigger the speculation protocol actions. For the non-marked accesses, the plain coherence protocol is used. We assume a full-map directory, invalidation-based scheme as the underlying coherence protocol [CF78]. Coherence states are not used in the regular way for the cache lines accessed with the speculation protocol. Instead, variables are time-stamped to track memory dependences, as we will explain later. However, some of the transactions of the speculation protocol use the information recorded in the bit vector of the directory, like the list of sharers for a particular block.

Main Idea

This protocol only requires task squashes under out-of order RAW dependences. It can eliminate WAR and WAW dependences, and gracefully process in-order RAW dependences.

To eliminate WAR and WAW hazards, writes of a processors are prevented from invalidate the caches of other sharer processors. This change enables multiple versions of a variable to exist in different caches. Furthermore, when a dirty line is displaced from a cache, in addition to being sent to its home, it is also stored in the memory in the local node. This way, later accesses from the same task can obtain the data locally. However, since these multiple versions can be displaced out-of order, when a dirty displaced line reaches its home, it will overwrite the shared data in the home only if the home contains an older version of the variable. With these changes, data are effectively privatized and writes can be executed by different processors in any order without causing WAR or WAW dependences.

To gracefully process RAW dependences, an exposed load (a load of a variable in a task that is not preceded by a write in the same task) to a variable has to proceed all the way to the home node. The cache coherence protocol is modified to find the correct version, that can be located either in the shared memory at the home, or in any of the sharer processors. Once located, the protocol supplies the version to the requester. With this support, in-order RAW dependences are satisfied seamlessly.

Required Support

The protocol allows multi-version caching and conditional write-backs at the home. To support these features, several data structures are mapped to each memory node, namely shared and local task-IDs, an overflow area, and a Memory History Buffer (MHB). The allocation and access to these structures is performed by the Directory Controller with the help of an Address Mapping Module. We consider all the elements in turn.

1) Task-IDs

To identify dependences, two shared task-IDs for each speculatively accessed variable are used: *MaxR1st*, which identifies the youngest task in the system that has executed an exposed load of the variable, and *MinW* which identifies the oldest task that has written to it so far in the system. In addition, to support conditional write-backs at the home, a shared *MaxWDisp* task-ID is kept for each variable. This task-ID identifies the youngest task that wrote to that variable and then displaced it to the home. These task-IDs are kept in the main memory of the home node.

Speculatively accessed variables are also locally tagged. For each variable a processor access, two private task-IDs are kept in its local memory: *PmaxR1st*, which identifies the youngest task ran by the local processor that executed an exposed load for that variable, and *PmaxW*, which identifies the youngest task executed by the local processor that has written that variable so far. Shared and private task-IDs are initialized to zero before the speculative section starts.

Finally, to further reduce messages with the local memory, cache tags keep summaries of the local task-IDs. Specifically, for each variable, a *Read1st* and a *Write* bit are kept. These bits are cleared when a new task starts. The *Read1st* bit is set only if the first access of a task to a variable is a read. The *Write* bit is set the first time the task writes a variable.

2) Overflow Area and Address Mapping Module

This protocol keeps a space-efficient area in each node memory where cache can overflow into without the need of task squashing. The first time that a dirty line containing speculatively modified data is displaced from the cache, in addition to being sent to its home, the local directory allocates a page in the memory of the local node, and stores the data. With this support, the local overflow area can be considered as an extension of the local cache. When either a request from the local processor or a remote request from the home misses in the local cache, the local overflow is interrogated. The overflow area will supply the data if it was displaced in dirty state earlier during task execution. This happens in two cases. First, in a non-exposed load from the local processor and, second, in an incoming read from the home that is trying to satisfy an in-order RAW dependence.

To access the local overflow area an address mapping table is needed. This table maps a physical shared address to the physical local address at the page granularity. Thus, for a given shared page address (SPA), this table keeps a pointer to the suitable overflow area page, and another pointer to the corresponding page of local task-IDs. Notice, that only the dirty displaced lines need to be valid in the overflow area. In addition, each node has a second table for the shared variable that are homed locally. This second table has a

pointer to the shared task-IDS. Both tables are integrated in the Address Mapping Module in each node, as shown in Figure 7.2.

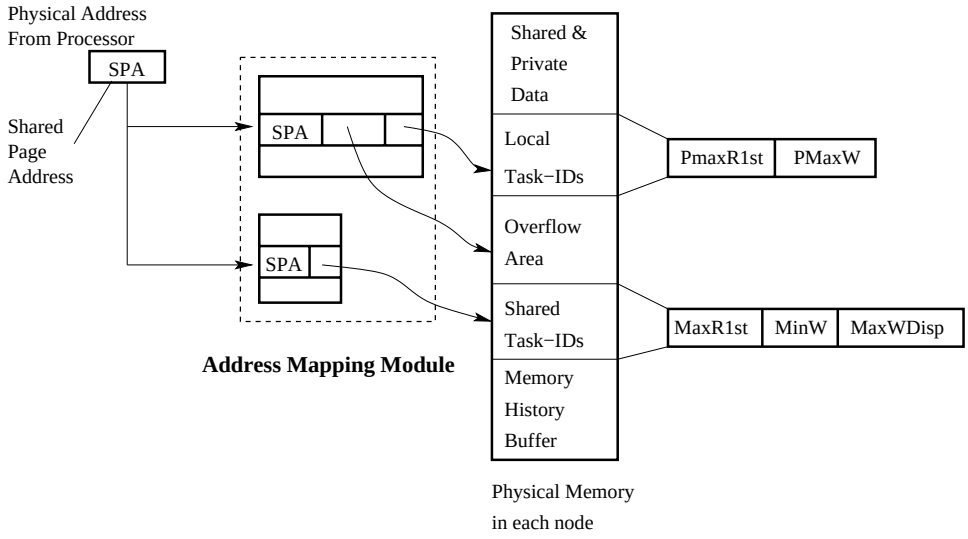

Figure 7.2. Address Mapping Module.

3) Multi-version Caching

To support multiple versions of a given variable in different caches, a simple procedure is needed. After a processor writes and the local directory controller receives a home-bound invalidation message, the directory controller intercepts it and replies with a completion message. The directory controller access the local task-IDs of the speculatively accessed data by means of the Address Mapping Module, and determines whether this is the first write of the task by comparing the local *PmaxW* to the ID of the current task being executed by the local processor. If *PmaxW* is lower, it sends a *Write-First* message to the home, and sets *PmaxW* to the current task-ID.

4) Conditional Write-Back at the Home

To support out-of-order WAW dependences, conditional write-back support is used at the home. Such support involves the use of *MaxWDisp* for each shared variable in the home. Remember that this task-ID keeps the number of the youngest task that wrote the element and then displaced it to the home. When a cache line is received at its home, each variable comes with the *PmaxW* it had when displaced. Then, for each variable in the line, its *PmaxW* and *MaxWDisp* are compared. If *PmaxW* is lower than *MaxWDisp*, the incoming variable is discarded. Otherwise, the variable overwrites the copy at home and *MaxWDisp* is set to *PmaxW*.

5) Memory History Buffer (MHB)

Since several tasks are executed in the same processor, a processor may end up buffering multiple versions of the same variable. In this case, the last version, the one produced by the youngest task that ran in the processor will be in the cache or overflow area; the previous ones are stored in a memory history buffer (MHB) which is placed in the local memory of each processor.

When a task is about to destroy a version of a variable that was created by a previous task that ran in the same processor, the old version is saved in an entry of its local MHB. The information that we need to save is the old value before being destroyed, the ID of the task that produced the value, and its physical address. All the entries created by a task are grouped together. The reason is that when a task commits, all its changes become safe, and thus all its entries in the MHB can be deallocated and its spaced can be reused.

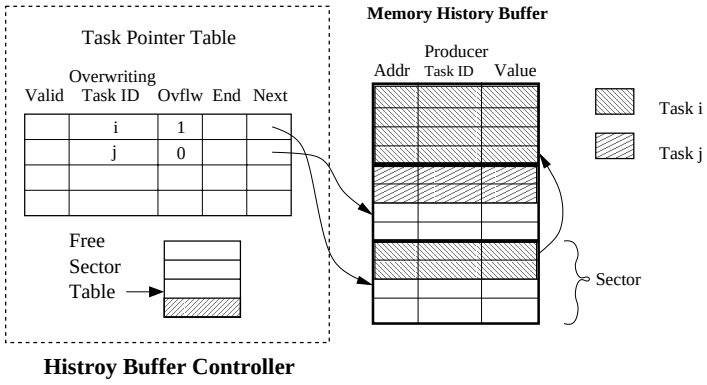

Figure 7.3. Per-processor structures required for the Memory History Buffer.

All the information in the MHB is handled in hardware using a history buffer controller shown in Figure 7.3. This module contains a Task Pointer Table, and a Free Sector Table. When a task starts running, it is dynamically assigned an entry in the Task Pointer Table and one sector in the history buffer. Free sectors are obtained from the Free Sector Table. Two pointers in the Task Pointer Table point to the Next entry to fill and the End entry to check for overflow. If the task needs more entries than a sector, another sector is dynamically assigned and linked to the previous one, while the Overflow bit is set. To insert a new entry into the MHB, the speculation protocol must detect when a version is about to be destroyed. On a write hit, the *Write* bit in the tag caches is checked. If the *Write* bit is zero, it is the first write of the task to that variable, and the old value in cache needs to be sent to the history buffer controller in the local node. However, if the write misses, the local $PmaxW$ task-ID is checked. If the $PmaxW$ task-ID of the variable is smaller than the ID of the task, the old value needs to be read from the overflow area and

saved into the MHB. Finally, there is a situation, that we explain later, where an entry needs to be inserted in the MHB on an exposed load. For that, the local task-IDs *PmaxW* and *PmaxR1st* are checked. Notice that all these operations are done in the background with fairly small overhead.

Finally, Figure 7.4 shows the set of modifications required in each node by the complete speculation framework. The components modified are the directory controller and the cache tags. In addition, the processor must support special load/store instruction for the memory accesses that can cause dependence hazards.

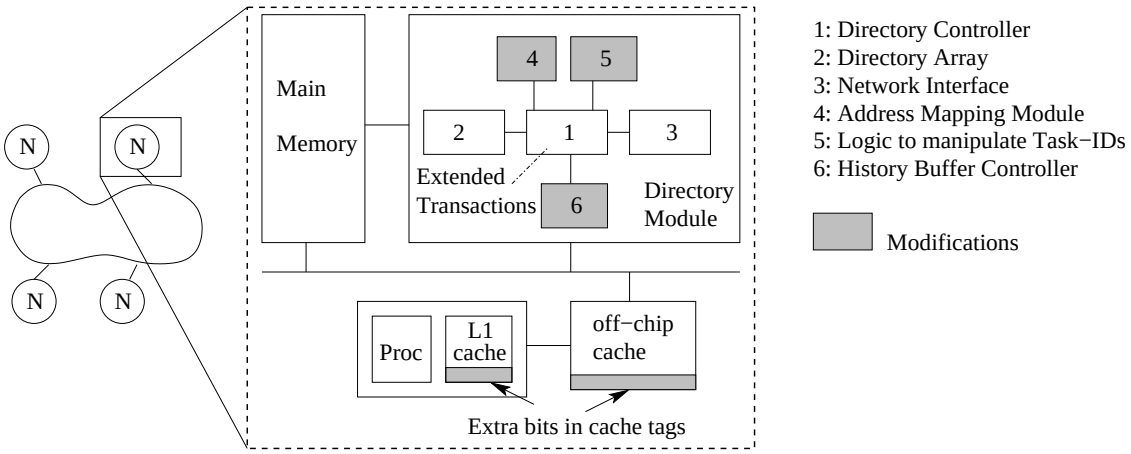

Figure 7.4. Modifications required in a DSM machine to support the complete speculation protocol.

Overall Algorithm

The overall algorithm can be divided into the actions performed by the processor locally (*Local Algorithm*) and those performed at the home (*Home Algorithm*). Figure 7.5-(a) shows a simplified algorithm with the actions performed by the local directory controller on reads and writes. For the read, it checks whether it is the first exposed load of the task. For that, the ID of the current task (TID in the algorithm) is compared to the value of *PmaxW* and *PmaxR1st*. If TID is higher, *PmaxR1st* is updated and a *Read-First* message is sent to the home. On a write, a similar algorithm is used. In this case, TID is compared to the *PmaxW* task-ID. If TID is higher, this is the first write of the variable by the task. *PmaxW* is updated, and and a *Write-First* message is sent to the home.

Remember that the cache tags are extended with *Read1st* and *Write* bits. These bits are used to reduce messages to the local directory controller. In a write hit, the *Write* bit in cache is checked. If it is reset, this is the first write to the variable by this task, the *Write* bit is set and a message is sent to the local directory controller, which will proceed as explained before. Otherwise, if the *Write* is already set, no message is sent to the local

memory, since the first write was already notified. Note that all these messages are sent in the background. Thus, the processor does not wait for the message to complete to proceed with the computation. For the *Read1st* bit a similar algorithm is used. *Read1st* and *Write* bits are cleared when a new task starts.

```
if Read
  if (PmaxR1st <= PmaxW < TID)
      insert_MHB (TID, old_value, addr, PmaxW)
  endif
  if ((PmaxR1st < TID) && (PmaxW < TID))
      send Read-First with TID to home
      PmaxR1st = TID
  endif
endif
if Write
  if ((PmaxR1st <= PmaxW < TID) ||
     (PmaxW == 0))
      insert_MHB (TID, old_value, addr, PmaxW)
  endif
  if (PmaxW < TID)
      send Write-First with TID to home
      PmaxW = TID
  endif
endif
```

(a) Local Algorithm

```
if Read-First
    if ((MinW != 0) &&
      ((MinW < TID) || (MaxWDisp != 0)))
        send Write-Back to all sharers
        if (MaxWDisp > TID)
            exception (WAR, addr, TID, PmaxR1st)
        endif
    endif
    MaxR1st = max (MaxR1st, TID)
endif
if Write-First
    if (TID < MaxR1st)
        exception (RAW, addr, TID, MaxR1st)
    endif
    if (MinW == 0)
        insert_MHB(TID, old_value, addr, 0)
    endif
    MinW = min (MinW, TID)
endif
```

(b) Home Algorithm

Figure 7.5. Algorithm for the speculation protocol at the Local Memory Node (a) and at the Home Memory Node (b). TID is the ID of the executing task.

Figure 7.5-(a) shows the algorithm executed by the Home. When the home receives a *Read-First* request, it must find the correct version. This version will be the one located in the shared memory when the variable was never written (*MinW* is equal to 0), or when the variable was written, but *PmaxR1st* is lower than *MinW* and no version has been displaced to the home (*MaxWDisp* is equal to 0). Otherwise, the home node must send a *Write-Back* request to all the sharer processors. The sharer processors will answer with its version in cache or overflow area (the last one) and its *PmaxW* task-ID. The home will select the version with the highest *PmaxW* task-ID and will store the data and task-ID in the shared memory location and *MaxWDisp* respectively. If at this point, the home finds that *MaxWDisp* is higher than the *PmaxR1st* of the Read-First request, it means that a WAR infraction would arise if the home supplied the requester the *MaxWDisp* version. To resolve this infraction, an exception handler needs to be run. This issue will be explained later in more detail. If, however, *MaxWDisp* is smaller than *PmaxR1st*, the version in the

home is sent to the requester processor. In any case, *MaxR1st* is updated to the maximum of the incoming *PmaxR1st* and *MaxR1st*.

If the home receives a *Write-First* request, the *PmaxW* of the arriving message is compared to the shared *MaxR1st*. If *PmaxW* is smaller than *MaxR1st*, a task has prematurely loaded a variable. Thus, an exception handler must deal with this out-of-order RAW dependence. This handler will squash all tasks with an ID higher than the writing one, and will revert memory to a safe state. If on the contrary, the *PmaxW* is equal or larger than *MaxR1st*, no dependence infraction is detected. The *MinW* task-ID is updated if necessary with the arriving *PmaxW*.

MHB Insertion

Notice that there are three points in the algorithms of Figure 7.5 where an entry needs to be inserted in the MHB. The most obvious one corresponds to a first write by a task in a processor. In this case, a version from an older task is about to be overwritten. Thus, the local directory controller inserts a new entry in its local MHB where it keeps the old value of the variable, its *PmaxW*, and the address. An special case occurs when *PmaxW* is zero. In this case, the write is the first write in the processor. Thus, the value that needs to be inserted into the MHB is the initial value of the variable before the speculation started. Instead of that, an entry with a value of `-1` for the producer task-ID is inserted into the MHB. Thus, if later a recovery process finds this entry when undoing a task, it will know that the initial value needs to be copied back to the home.

The initial value is backed up by the home node the first time that a *First-Write* message reaches its memory home (Figure 7.5-(b)). The home saves an entry in its node's MHB with the initial value, the address and the zero producer task-ID.

Unfortunately, another situation where an entry needs to be inserted in the MHB is on an exposed load. On an exposed load a new version from outside the processor can kill a local version. Suppose that a task i has executed in a processor and has generated the version i of a variable that has not yet saved in the MHB. This version will be in the cache, or in the overflow area if it was displaced. Suppose now that a new task $i + k$ in the same processor executes an exposed load of that variable. This exposed load will cause a *Read-First* request message to be sent to the home node, which will supply the correct version (a version j, with j in between i and $i+k$). In this case, version j can destroy the unsaved version of the previous task i.

However, we do not need to save an entry under every exposed load that a processor executes. We only need to save an entry when the condition (PmaxR1st `<=` PmaxW `<` TID) is true. In this case, the processor has a version that it did not save in its MHB. This condition says that we only need to save a version on the first exposed load of a task.

In addition, only on the first exposed load, after a writing task, we need to insert an entry in the MHB.

Dealing with Dependences

Using this protocol WAR and WAW hazards are eliminated and in-order RAW dependences are gracefully processed. Table 7.1 lists how the dependences are handled, and what are the corresponding actions that need to be undertaken. We explain them next.

Dependence	Run-time order	Notes	Actions taken
RAW	in-order	Search for the right producer data	Hardware Software if also WAR
	out-of-order	Squash and rerun tasks larger than producer task	Software
WAR	in-order	No problem	-
	out-of-order	If the home has been updated, search to read the right data	Software
WAW	in-order	No problem. Home chooses the last write among the displacements	Hardware
	in-order	Same as above	

Table 7.1. How to handle dependences between tasks.

In an in-order RAW dependence, the correct version needs to be searched. If this is the only dependence that has occurred, the search will be done in hardware. The version required may be found in the shared memory, or in any of the sharer processors in either the cache or the overflow area in the local memory's node. If the in-order RAW dependence is mixed with an out-of-order WAR dependence, the search needs to be done in software. In an out-of order RAW dependence, the reading task has already consumed a wrong value. Thus, special actions needs to be taken. They are done in software.

Obviously, an in-order WAR does not pose any problem. Both reading and writing tasks have executed in the correct order. However, the read of an out-of order WAR dependence cannot find the initial value if the shared memory location in the home has been updated by a cache displacement with the value of the youngest version. In this case, overflow areas and maybe MHB need to be searched. This process is also done in software.

In-order and out-of order WAW dependence are not problematic, since they are solved by the home directory which always takes the youngest displaced version.

Thus, the only two dependence infractions that need special actions are out-of order RAW and a out-of order WAR. These two infractions need to be repaired in software. They are considered in detail next.

Out-of-oder RAW: Recovery by Undoing and Re-Running Tasks

An out-of order RAW dependence is detected when a write update message brings to the home a *PmaxW* that is lower than the home's *MaxR1st*. The reader has read prematurely. In this case, the recovery consists of a software handler that undoes the effects done by tasks greater than the producer one.

After the error is detected, all tasks with ID higher than the writing one are squashed. The other tasks are allowed to complete. When they complete, the ID of the writing task becomes the *Commit Point* (CP). At this moment, the home memory contains three types of data: *uncommitted, committed* and *last-update*. *Uncommitted* data was generated by tasks higher than the writing one and, therefore, their *MaxWDisp* is higher than the CP. *Committed* data was generated by the other tasks and their *MaxWDisp* is lower or equal to the CP. Finally, *last-update* is one type of *committed* data: the one generated by the highest write lower or equal to the commit point. The recovery process will simply involve setting all the speculatively modified data in the home to their *last-update* value.

The first step is cache flushing. As dirty data is flushed from the caches and arrives at the home, two cases are possible. If the data at the home is *committed* (*MaxWDisp* <= *CP*), we only accept the incoming data if it brings us closer to the *last-update* value (Figure 7.6-(a)). This occurs when the incoming datas *PmaxW* is such that *MaxWDisp* <= *PmaxW* <= *CP*. If, instead, the data in the home is *uncommitted* (*MaxWDisp* > CP), we only accept the incoming data if it takes us to a *committed* version (arrow 1 in Figure 7.6-(b)). This occurs when the incoming datas *PmaxW* is such that *PmaxW* <= *CP*.

Unfortunately, the change from *uncommitted* to *committed* does not guarantee ever producing the *last-update* version, even after flushing all caches. Indeed, such a version may not be in any cache any more: it may have been displaced from a cache during the regular execution of the algorithm and discarded because the home already had a newer (the uncommitted) version. In this case, the *last-update* version may be in a local memory. Consequently, after the change from *uncommitted* to *committed*, the home automatically request a further write back of the line from all the sharers of the line, as indicated by the directory. For each sharer, this write back request will get the data from the local memory. As each of these writebacks returns data to the home, the same algorithm is used to decide if it is to be accepted. These writebacks can get us closer to the *last-update* value (arrow 2 in Figure 7.6-(b)).

The second step if for each processor to restore data from its own history buffer. Only tasks with IDs higher than the CP are restored. The order of the restores does not matter, since the home uses the algorithms described above to accept or reject versions as they are written back. As before, every time that we record a transition like arrow 1 in

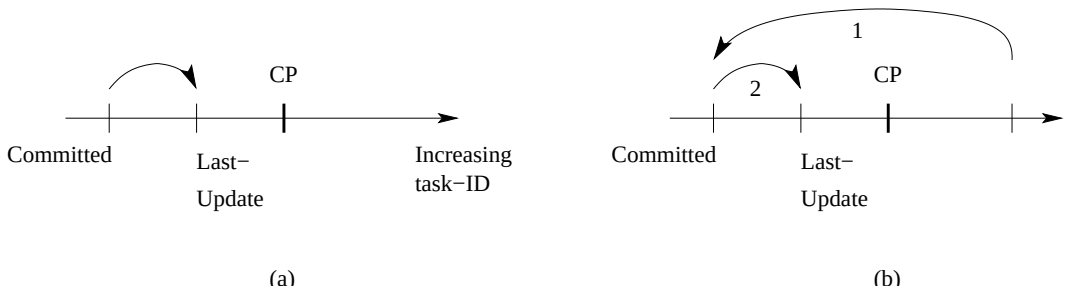

Figure 7.6. Recovering the state in an out-of-order RAW dependence infraction.

Figure 7.6-(b), the home requests a further *Write-Back* from all the sharers. Overall, after the restoration of the history buffers, the home only contains *last-update* versions. The space in the history buffers is then recycled, the task IDs are reset, and parallel execution is restarted.

Out-of-Order WAR: Retrieve the Correct Version

Unfortunately, there is one additional case where recovery is needed: an out-of-order WAR resulting from the on-demand merging of this protocol. Recall that, when a task executes an exposed load, it send a message to the home node. When the home receives this message, it request a *Write-Back* to all the sharer processors, selects the one with the highest *PmaxW* task-ID, and stores data and task-ID in the shared memory location and *MaxWDisp* respectively. If, at this point, it finds that *MaxWDisp* is higher than the *PmaxR1st* of the exposed load, it means that there is out-of-order WAR. To recover, a software handler has to dig out the correct version of the variable from caches, overflow areas or MHBs. The correct version will be the one with the higher task-ID, that is still lower than the reading task. The software handler runs in the processor executing the reading task. No other processor needs to be involved and no task needs to be squashed.

Appendix B

Description of the Applications

This Appendix gives a brief explanation of the parallel applications used in the experimental framework of Chapters 5and 6. These applications are: *Apsi* from SPECfp2000 [J.L00], *Track* and *Bdna* from Perfect [B$^+$89], *Dsmc3d* from HPF-2 [DSH94], *P3m* from NCSA, and *Tree* from [J. 94].

- *P3m* This applications implements a N-body simulation problem. The loop under test is invoked 1 time, with 97336 iterations. Each iteration computes the forces at one particular node. These forces depend on the number of neighbour nodes and the distance between them. Thus, nodes placed in a populated area, will receive force interactions from many nodes. However, nodes placed far away from other nodes will receive little force interactions. A consequence of this is that this application is very imbalanced. A few iterations spend a lot of time computing forces, while many of them almost do not have forces to compute. On the other hand, the arrays that are speculative accessed can in practice be privatized. However, the dependence structure cannot be fully analyzed and therefore the compiler does not parallelize this loop. While the loop has 97,336 iterations, we only use the first 9,000 iterations in the evaluation.
- *Tree* This application implements a hierarchical N-body simulation of the evolution of collisionless systems. The loop under test is invoked 41 times and consists of a while loop that traverses a tree structure, with 4096 iterations. The speculative array is used as a stack and in practice elements are only poped and used by an iteration after they have been previously written and pushed on the stack by the same iteration. This array is therefore privatizable, although this pattern cannot be fully resolved at compile time.
- *Bdna* This application simulates the hydration, structure, and dynamics of nucleic acids by molecular dynamics simulations of biomolecules in water. The loop under test is invoked 5 times, with 1499 iterations each. The speculative arrays are privatizable. At each invocation several elements are written in succession and later on read in succession. As the reads use subscripted subscripts the loop is non-analyzable at compile time. In practice, however, the set of elements read at each iteration is fully contained in the set of elements written previously by the same iteration.

- *Apsi* This application simulates, in three dimensions, pollutant concentration and deposition patterns in lake shore environments. We use an input grid of 512x1x64. The loop under test is invoked 900 times, with 63 iterations each. All invocations present the same behaviour. There is only an array which is speculatively accessed. At run-time, some portions of the array are only read, while other portions are privatizable, since a write always precedes a read in each iteration using a certain datum. This behaviour, however, cannot be fully determined at compile time. At run-time there are no same-word RAW dependences.
- *Dsmc3d* This application performs a randomized simulation of gas particles using a Direct Simulation Monte Carlo method. The loop under test is invoked 80 times, with 758972 iterations each. The speculative array is accessed very sparsely and has a few same-word RAW dependences. At run time some of them are executed out-of-order.
- *Track* This application simultaneously tracks the trajectory of an unknown number of missiles launched from several different sites. The loop under test is invoked 56 times, each executing a different number of iterations, with an average of 502 iterations per invocation. A few arrays must be treated speculatively, but only one of those actually has same-word RAW dependences. The writes to this particular array are guarded by a loop variant condition, whereby the loop is not analyzable at compile time. Out-of-order RAW, however, occur very sparsely in practice.

Appendix C

Example of Poor MultiT&SV Performance

Consider a program with 6 tasks that we run on 3 processors. The tasks are load-imbalanced as shown in Figure 7.7-(a) and have mostly-privatization patterns. Suppose that tasks *T0*, *T1*, and *T2* have been assigned to processors *P0*, *P1*, and *P2*, respectively. Figure 7.7-(b) shows the time when *P2* finishes *T2*. Under SingleT (Figure 7.7-(c)), *P2* will wait until *T2* commits (point *c2*) and then execute *T5*. Under MultiT&SV (Figure 7.7-(d)), *P2* will greedily start *T3* and then stall. Unfortunately, in this case, *P0* will later get a task (*T4*) far from the non-speculative one at that time (*T1*). As a result, after it finishes it and takes *T5*, it will remain stalled for a long time, wasting many cycles.

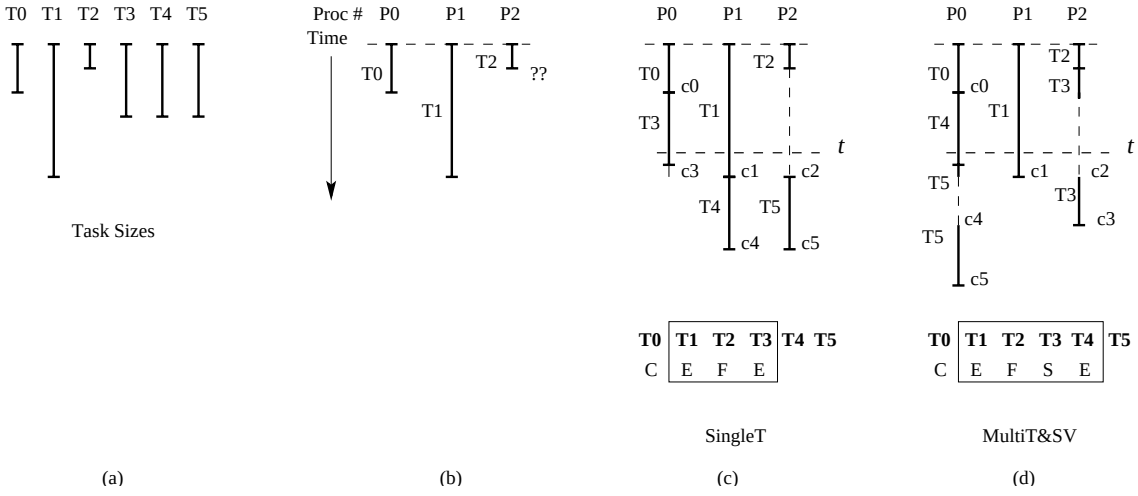

Figure 7.7. Example where MultiT&SV is slower than SingleT. The example assumes that tasks commit without visible delay.

The overall result is that MultiT&SV is slower: *T5* finishes and commits much later (*c5*) than in SingleT. The reason is that under SingleT tasks end up being round-robin assigned, what guarantees that the tasks making progress are the ones closer to the commit point. However, under MultiT&SV tasks are greedily assigned. This greedy assignment may result in tasks far from the commit point making progress, while other tasks closer to it are stalled. This effect delays the commit wavefront, which is counter-productive for MultiT&SV schemes running applications with mostly-privatization The reason is that under these circumstances the commit wavefront is in the critical path because a stalled task cannot resume with the execution until the previous one commits. Therefore, delaying

the task commit results in delays of task execution.

All these effect are shown in the example of the Figure, which also shows the window of uncommitted tasks at time t under SingleT and MultiT&SV. This window includes the non-speculative task and all the other speculative ones. Under SingleT, this window contains three tasks, *T1*, *T2* and *T3*. Under, MultiT&SV the window has grown larger and also contains *T4*. However, the number of tasks making progress (either being executed (E) or finished (F)) is the same in both cases because *T3* is stalled under MultiT&SV. The result of this stall is that *T3* will commit later under MultiT&SV than under SingleT, and thus will delay the task commit wavefront of the MultiT&SV scheme.

References

[AI92] B. R. Allison and C. Van Ingen. Technical Description of the DEC 7000 and DEC 10000 AXP Family. *Digital Technical Journal*, 4(4):1–1, Special Issue 1992.

[B+89] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[BBO+83] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[BC91] J.L. Baer and T.F. Chen. An Effective on-chip Preloading Scheme to Reduce Data Access Penalty. In *Proc. of Supercomputing'91*, pages 176–186, 1991.

[BC02] G. Bell and J. Cray. What's Next in High-Performance Computing? *Communications of the ACM*, 45(2):91–95, February 2002.

[BDE+96] 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. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[Bel85] C. G. Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–467, April 1985.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 782–789, 1985.

[CB94] T.F. Chen and J.L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture (ISCA'94)*, pages 223–232, April 1994.

[CC98] C.H. Chi and C.M. Cheung. Hardware Prefetching for Pointer Data References. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 377–384, 1998.

[CF78] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. on Computers*, 27(2):1112–1118, Dec 1978.

[CHS+99] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture (HPCA'99)*, pages 70–79, January 1999.

[CMT00] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 13–24, June 2000.

[Cor96] Digital Equipment Corporation. Alpha Architecture Handbook. Version 3. October 1996.

[CTY94] D. K. Chen, J. Torrellas, and P. C. Yew. An Efficient Algorithm for the Run-Time Parallelization of Do-Across Loops. In *In Proc. of Supercomputing '1994*, pages 518–527, November 1994.

[DDS93] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 Int'l Conference on Parallel Processing (ICPP'93)*, volume 1, pages 56–63, August 1993.

[DGP+02] F. Dang, M.J. Garzarán, M. Prvulovic, A. Jula, H. Yu, N. Amato, L. Rauchwerger, and J.Torrellas. SmartApps, an Application Centric Approach to High Performance Computing: Compiler-Assisted Software and Hardware Support for Reduction Operations. In *Workshop on Performance Engineering Technology and Research Sponsored Under the NSF Next Generation Software Program*, February 2002.

[DMRV95] I. Duff, M. Marrone, G. Radiacti, and C. Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for Sparse Matrices. Technical Report RAL-TR-95-049, Rutherford Appleton Laboratory, 1995.

[DS95] F. Dahlgren and P. Stenström. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of the 1st Int'l Symposium on High-Performance Computer Architecture (HPCA'95)*, pages 68–77, January 1995.

[DS96] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[DS98] F. Dahlgren and P. Stenström. Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 47(18):385–398, October 1998.

[DSH94] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[DUSH94] R. Das, M. Uysal, J. Saltz, and Y.S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[ELPS98] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 140–151, February 1998.

[FF01] R. Figueiredo and J. Fortes. Hardware Support for Extracting Coarse-grain Speculative Parallelism in Distributed Shared-memory Multiprocesors. In *Proc. of the Int'l Conference on Parallel Processing (ICPP'01)*, September 2001.

[FLA01] M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Technical Report MIT/LCS Technical Memo 619, July 2001.

[FP92] J.W.C. Fu and J.H. Patel. Stride Directed Prefetching in Scalar Processors. In *Proc. of the 25th MICRO*, pages 102–110, 1992.

[FS96] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Trans. on Computers*, 45(5):552–571, May 1996.

[GB88] W. Gunsteren and H. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical Report, Laboratory of Physical Chemistry, University of Groningen, 1988.

[GBIV01] M.J. Garzarán, J. Briz, P. Ibáñez, and V. Viñals. Hardware Prefetching in Bus-based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In *Proc. of the 2001 Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 345–354, 2001.

[GGK+83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. In *IEEE Trans. on Computers*, pages 175–189, February 1983.

[GN98] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proc. of Supercomputing '1998*, November 1998.

[GPL+00] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Buffering State with Software Logging in Scalable Speculative Parallelization. Technical Report CSRD-1581, University of Illinois at Urbana-Champaign, July 2000.

[GPL+01] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and Josep Torrellas. Software Logging under Speculative Parallelization. In *Workshop on Memory Performance Issues hold in conjunction with ISCA'01*, July 2001.

[GPL+02] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Software Logging for Multi-Version Buffering under Speculative Parallelization. Technical Report RR-02-04, Departamento de Informática e Ingeniería de Sistemas, March 2002.

[GPZ+01] M.J. Garzarán, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pages 243–254, September 2001.

[GVSS98] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 195–205, February 1998.

[GVW89] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. of the 3rd Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 64–73, April 1989.

[HAA+96] 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.

[Hen00] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[HH98] C. Tseng H. Han. Improving Compiler and Run-Time Support for Adaptive Irregular Codes. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[HK99] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proc. of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 171–181, January 1999.

[HKD+94] M. Heinrich, J. Kuskin, D.Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operationg Systems (ASPLOS'94)*, pages 274–285, October 1994.

[HWO98] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *In Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 58–69, October 1998.

[Iba] Pablo Ibáñez. Gestión Multinivel y Prebúsqueda Hardware en Memorias Cache Integradas. Ph.D. Thesis, Universidad de Zaragoza, Departamento de Informática e Ingeniería de Sistemas., July 1998.

[IVBG98] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 369–376, 1998.

[J. 94] J. E. Barnes. Treecode. Technical Report, Institute for Astronomy, University of Hawaii, 1994.

[JG97] D. Josephand and D. Grunwald. Prefetching Using Markow Predictors. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture (ISCA'97)*, pages 252–263, June 1997.

[JG99] D. Joseph and D. Grundwald. Prefetching using Markov Predictors. *IEEE Trans. on Computers*, 48(2):121–133, February 1999.

[J.L00] J.L. Henning. SPEC CPU2000: Measuring Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[JT93] Y. Jegou and O. Temam. Speculative Prefetching. In *Proc. of the 1993 Int'l Conference on Supercomputing (ICS'93)*, pages 1–11, 1993.

[KCPT95] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, pages 1250–1264, December 1996. A shorter version appeared in Proc. of the 9th Int'l Conference on Supercomputing (ICS'95), pages 255-264, July 1995.

[KDL+93] D.J. Kuck, E.S. Davidson, D.H. Lawrie, A.H. Sameh, and C.Q. Zhu. The Cedar System and an Initial Performance Study. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 213–224, May 1993.

[KDLS87] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. *Experimental Parallel Computing Architectures: Volume 1 - Special Topics in Supercomputing*, chapter Parallel Supercomputing Today and the Cedar Approach, pages 1–23. J. J. Dongarra editor, North-Holland, New York, 1987.

[Kni86] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[KRS88] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization on Multiprocessors with Shared Memory. *ACM Trans. on Programming Languages and Systems*, 10(4):579–601, October 1988.

[Kru86] C. Kruskal. Efficient Parallel Algorithms for Graph Problems. In *Proc. of the 1986 Int'l Conference on Parallel Processing (ICPP'86)*, pages 869–876, August 1986.

[KT98] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[KT99] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, pages 866–880, September 1999.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int'l Symposium on Computer Architecture (ISCA'90)*, pages 148–159, May 1990.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[LP98] Y. Lin and D. Padua. On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In *Proc. of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, 1998.

[LRV94] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 208–218, October 1994.

[MG91] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[MG99] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *Proc. of the 1999 Int'l Conference on Supercomputing (ICS'99)*, pages 365–372, June 1999.

[MGT98] P. Marcuello, A. González, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 1998 Int'l Conference on Supercomputing (ICS'98)*, pages 77–84, July 1998.

[MH96] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proc. of the 1996 Int'l Conference on Supercomputing (ICS'96)*, pages 133–143, 1996.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. of the 1995 Int'l Conference on Parallel Processing (ICPP'95)*, pages 1–10, August 1995.

[OWP+01] C.L. Ooi, S. Wook, K.I. Park, R. Eigenmann, B. Falsafi, and T.N. Vijaykumar. Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor. In *Proc. of the 2001 Int'l Conference on Supercomputing (ICS'01)*, June 2001.

[PBG+85] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 764–771, 1985.

[PGRT01] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proc. of the 28th Annual Int'l Symposium on Computer Architecture (ISCA'01)*, pages 204–215, July 2001.

[PN85] G. Pfister and A. Norton. 'Hot Spot' Contention and Combining in Multistage Interconnection Networks. In *Proc. of the 1985 Int'l Conference on Parallel Processing (ICPP'85)*, pages 790–797, August 1985.

[Pot97] W.M. Pottenger. Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs. Technical Report, University of Illinois at Urbana-

Champaign, Center for Supercomputing Research and Developement, May 1997.

[PP84] M. S. Papamarcos and J. H. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. of the 11th Annual Int'l Symposium on Computer Architecture (ISCA'84)*, pages 348–354, June 1984.

[Pri95] C. Price. *MIPS IV Instruction Set, Revision 3.2.* MIPS Technologies, Mountain View, CA, September 1995.

[PS91] J.C. Park and M.S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlet Packard Laboratories, Palo Alto, CA, May 1991.

[PTA97] V.S. Pai, P. Tanganathan, and S.V. Adve. The Impact of Instruction-Level Parallelism onn Multiprocessor Performance and Simulation Methodology. In *3rd Int'l Symposium on High-Performance Computer Architecture (HPCA'97)*, pages 72–83, February 1997.

[RIVL00] L. Ramos, P. Ibáñez, V. Viñals, and J.M. Llabería. Modeling Load Address Behaviour Through Recurrences. In *Proc. of the 2000 Int'l Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pages 101–108, April 2000.

[RMS98] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 115–126, October 1998.

[RP94] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. of the 1994 Int'l Conference on Supercomputing (ICS'94)*, pages 33–43, July 1994.

[RP95] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. of the SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 218–232, June 1995.

[RP99] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.

[RS00] P. Rundberg and P. Stenstrom. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[SBV95] G. S. Sohi, S. Breach, and S. Vajapeyam. Multiscalar Processors. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 414–425, June 1995.

[SCM97] J.G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[SCZM00] J.G. Steffan, C.B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture (ISCA'00)*, pages 1–12, June 2000.

[SJG92] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 80–91, 1992.

[Smi78] Alan Jay Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[Smi81] B.J. Smith. Architecture and Aplications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[Smi82] A. J. Smith. Cache Memories. In *Computing Surveys*, pages 473–530, September 1982.

[SP88] J.E. Smith and A.R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. on Computers*, C-37(5):562–573, May 1988.

[SSC+99] H. Saito, N. Stavrakos, S. Carrol, C. Polychronopoulos, and A. Nicolau. The Design of the PROMIS Compiler. In *Proc. of the Int'l Conference on Compiler Construction*, pages 562–573, March 1999.

[TE93] D.M. Tullsen and S.J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*, pages 278–288, may 1993.

[TE95] D.M. Tullsen and S.J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Trans. on Computer Systems*, 13(1):57–58, February 1995.

[THA+99] J.Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.C.Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):881–902, September 1999.

[VF94a] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[VF94b] J.E. Veenstra and R.J. Fowler. The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors. Technical Report number 490, Computer Science Departament. Univ. of Rochester. New York, 1994.

[WG94] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[YR00] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization. In *Proc. of the 2000 Int'l Conference on Supercomputing (ICS'00)*, May 2000.

[Zha99] Y. Zhang. Hardware for Speculative Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, May 1999.

[Zim91] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

[ZPG+99] Y. Zhang, M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas. A Framework for Speculative Parallelization in Distributed Shared-Memory Multiprocessors. Technical Report CSRD-1582, University of Illinois at Urbana-Champaign, July 1999.

[ZRT98] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th Int'l Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 162–174, February 1998.

[ZRT99] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th Int'l Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 135–139, January 1999.

[ZT95] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, pages 188–199, June 22–24, 1995.

[ZY87] C.Q. Zhu and P.C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. In *IEEE Trans. on Software Engineering*, pages 726–739, June 1987.