INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

ARCHITECTURAL SUPPORT FOR MASSIVELY-CONCURRENT PARALLEL COMPUTING

JAYA NARAIN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JUNE 1998
© JAYA NARAIN, 1998

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-39489-1

Canada

Abstract

Architectural Support for Massively-Concurrent Parallel Computing

Jaya Narain

Parallel computing is an intricate mix of marketplace requirements, architectural understanding, technology issues, and issues concerning costs. One controlling myth is that high-volume commodity processors must, by the nature of things, be the common building blocks for both desktop clients and room-size servers. This myth—and the supporting myth of architectural convergence of clients and servers—should be subject to dispassionate analysis.

With only one program counter per processor, conventional processors are becoming increasingly unresponsive in spite of faster clock rates. We will show that reliance on Instruction-Level Parallelism (ILP) for performance drives processor state upwards. When this massive state is not distributed across multiple program counters, processors choke on their own expensive context switches, here reconceptualized to show their true cost.

Within the framework of Little's law from queueing theory, we analyze conventional RISC superscalar processors as a case study of the inadequacy of the class of "ILP" processors. We contrast this to multithreaded processors that exploit both ILP and Thread-Level Parallelism (TLP).

As a contribution to parallel programming, we show how data caches in multithreaded architectures can be used to manage speculative state, and perform atomic updates involving multiple variables.

There is no "convergence architecture"; there are only divergence architectures.

Acknowledgments

Thank you Dr. David Probst for being my friend and guru! Without your excellent supervision, intellectual drive, continuous motivation, patience and financial support, this work would not have been realised so effectively in just 12 months.

I am grateful to the other members of the thesis committee, Drs. R. Jayakumar, H.F Li and Peter Grogono, for their valuable suggestions and reading my thesis in limited time. Dr. Li was, throughout, a virtual sounding board on architectural issues, and a feedback control on excessive religious zeal! I owe a particular debt of gratitude to Dr. Li and Dr. Probst for their generosity in extending financial support to me when I really needed it. My thanks to the Department of Computer Science for awarding me Teaching Fellowships and to the School of Graduate Awards for a partial tuition waiver.

My thanks to all faculty members, the System Analysts Pool and the secretaries for providing an environment congenial to academic research with fewer frustrations. Many thanks to Guy Dumais for answering all my *LATEX* questions, to Rachit, Manolo, Bhaskar, Sriram, Venkat, Jennifer, Manu, Rajat, Shilpee, Vinu and Manish who have provided priceless friendships and made my life in Montreal fun and exciting. I am particularly grateful to Krishna for his magnanimity. I cannot thank enough Ganesh (often my greatest critic!) for being an unfailing friend and standing by me, no matter what.

I am eternally indebted to my parents, my family and my relatives for their love and intellectual motivation. I adore my father for his countless sacrifices and constant struggle to provide me with the best education. I thank my little nephews, the desire to see whom has hastened the completion of this thesis.

Contents

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Motivation: The crisis in high-end computing 1

1.1.1 Why the crisis is real 5

1.2 Little's law 7

1.2.1 Two ways to go 7

1.3 Contributions of this thesis 8

2 Superscalar Architectures 11

2.1 Caches 12

2.2 Superscalar techniques to implement and exploit concurrency . . . 15

2.2.1 Historical perspective 15

2.2.2 Model of a superscalar processor 16

2.2.3 Instruction fetching and branch prediction 19

2.2.4 Instruction decoding, renaming, and dispatch 21

2.2.5 Instruction issuing and parallel execution 24

2.2.6 Handling memory operations 25

2.2.7 Committing or retiring instructions 26

2.3 Out-of-order issue/execution of instructions 26

2.4 Speculative Execution 28

2.5 Summary of the disadvantages of superscalar machines 30

3 Multithreading 31
3.1 Early multithreading 36
3.1.1 Coarse-grained or blocked multithreading 37
3.1.2 Fine-grained or switch-on-every-cycle multithreading 39
3.1.2.1 Fine-grained multithreading with full single-thread support and caching 39
3.1.2.2 Fine-grained multithreading without caches 41
3.2 Simultaneous multithreading 42
3.2.1 Simultaneous multithreading with heavyweight threads (SMT1) 43
3.2.2 Simultaneous multithreading with lightweight threads (SMT2) 46
3.2.2.1 Architectural overview of SMT2 46
3.2.3 Qualitative and Quantitative Comparisons between SMT1 and SMT2 49

4 Shared Responsibility for Parallellism 53
4.1 Explicit Programmer Control Decomposition 55
4.1.1 The CC++ Programming Model (Explicit Control Constructs) 55
4.1.2 The OpenMP Programming Model (Control Constructs as Compiler Directives) 60
4.1.3 Tera Loop Directives 65
4.2 Examples of Explicit and Implicit Parallelism 68
4.2.1 Examples of exlicit and implicit parallelism in loops and vectors 69
4.2.2 Parallelization versus vectorization 75
4.3 Wait-free or lock-free transactions 77

5 Conclusions and future work 83

Bibliography 86

List of Figures

1.1 Little's law says: concurrency equals bandwidth times latency 7

2.1 Memory hierarchy with primary (L_1) and secondary (L_2) caches . . . 13

2.2 Flow Diagram of Lockup or Blocking Cache 13

2.3 Flow Diagram of a Lockup-free or Nonblocking Cache 14

2.4 Stages of parallel processing in a superscalar execution. 17

2.5 Organization of a Superscalar Processor. 18

2.6 A Sample Program with corresponding Control-flow and Dataflow Graphs. 23

2.7 A reorder buffer. 28

3.1 Comparison of issue slot partitioning in various architectures. 44

3.2 Empty issue slots as vertical or horizontal wastes 45

4.1 An example of *cyclic reduction* using 9 threads to compute the sum of the first 9 array elements starting from 1. The maximum number of steps is $\lceil log_2 9 \rceil$. 74

4.2 Thread 1's transaction on variables a and b. 80

4.3 Thread 2's transaction on variables b and c. 81

4.4 The simple address register $AR[\,]$ required to increment the multiword sync variable nc. 82

List of Tables

Chapter 1

Introduction

"There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things."

— Niccolo Machiavelli, The Prince (1532)

Today, high-end computing is at a crossroads. We offer a possible scenario as to how we arrived at this crisis. The thesis itself provides increased architectural understanding of a proposed solution to this crisis in the time frame of the next five years. In a word, we explain why multiple program counters matter: they are the prequisites for the massive concurrency required to run a new class of large-scale irregular problems.

1.1 Motivation: The crisis in high-end computing

The traditional design space of parallel computers is fairly well agreed on. In grouping processors together, one chooses weak or powerful processors, and one chooses few or thousands of processors. Few, powerful processors is the world of conventional mainframes and supercomputers. Thousands of weak processors is the world of the traditional massively parallel processors. The former are costly, with limited scalability, while the latter are often special-purpose machines. Designers have looked for

a "sweet spot" in this design space by designing parallel computers with hundreds (rather than thousands) of rather more powerful uniprocessors, or tightly-coupled SMP clusters as the basic repeatable computer engine. While we cannot address all the issues in this design space, we can refer to some trends that will help situate the problem of this thesis, and its proposed solution.

Five years ago, there was little talk of the "end of architecture", and little sense of a crisis in high-end computing. Some users had migrated to **symmetric multiprocessors** (SMPs) because they had modest performance needs. They benefited from the standard programming model (i.e., performance model). The performance of an application on a true SMP is independent of data layout. And this has enormous benefits for programming, for the availability of independent software vendor (ISV) applications software, etc. The problem with SMPs is that the computer industry has *no way to scale* them to higher and higher number of processors *without losing memory uniformity,* and hence the desirable programming model.

If SMPs are not enough, then—always in the view of the computer industry—we are left with two[1] alternative architectures that might possibly permit high-end computation. The first architecture is that of **parallel vector processors** like the Cray T90 and the NEC SX-4. A significant portion of the technical community has grown dependent on vector processors, and is loath to give them up. There are problems, however. First, for economic reasons, vector vendors cannot afford—aside from the Japanese manufacturers of course—to invest the R&D money necessary to design follow-on generations. Hence, users are either deserting vector processors for mid-range machines with better price/performance, or are very, very unhappy at the unavailability of new generations of vector machines. They may certainly buy Japanese machines, however, when this is permitted by law.

Why is it difficult for vector users to switch? Simply because legacy vector codes do not port to other parallel architectures without profound reprogramming, which is often prohibitively expensive. Also, vector processors have trouble attracting new user communities because many of the new application programs—coming from commercial users or sophisticated technical users with modern simulation and visualization codes—do not match well with vector strengths. These applications have short vector

[1]We could also mention message-passing multicomputers, which forsake shared memory altogether, but multicomputers are outside the scope of this thesis. Although much work has been done on them, they are not the royal road to general-purpose parallel computing.

or even scalar arithmetic, frequent conditional branches, etc., which do not perform well on vector machines.

The crisis in high-end computing started with the realization that SMP multiprocessors and vector machines were together inadequate to meet the needs of several important user communities. The most prominent of these are: weather-prediction agencies, automobile and aircraft manufacturers, the Pentagon, the DoD's High-Performance Computing and Modernization Office (HPCMO), and the intelligence community (e.g., the standard three-letter agencies).

Vector processors and SMPs provide complementary approaches to supporting parallelism. Vector processors exploit parallelism in inner loops, and use a very fast, very expensive, memory system. Ironically, parallel vector processors do not scale because of memory contention at memory banks. Commodity processors exploit parallelism by aggressive attempts to expose instruction-level parallelism (ILP), and avoid latency by heavy reliance on data caches—so as to require less parallelism. As clock rates scale, vector lengths must become longer, and caches must become larger and hierarchical. The result is that such machines support an increasingly smaller fraction of application programs. Cray Research could not scale the Triton: as processors beyond 32 were added, it became impossible to sustain flat memory. Essentially, Cray did not spend enough money on its interconnection network. The inadequate bandwidth impeded scalability because of memory contention. Both vector processors and SMPs are predicated on flat memories, and cannot reconcile high processor count and flatness. Because there appeared to be no alternative, computer vendors turned to the idea of grouping together large numbers of "commodity" processors into parallel computers.

The second basic architectural alternative for high-end computing in 1998 might as well be called **massively parallel processing (MPP)**. The idea is to take large numbers of scalar processors that are produced in great volumes by Intel, MIPS Co., Sun Microsystems Inc., etc., and to group them together into larger and larger multiprocessors—conceivably with as many as 10,000 or 20,000 CPUs. The computer industry of the MPP flavor is reconciled to nonuniform shared memory, which abandons the SMP programming model, but tries to find a middle ground by building reasonably large SMP clusters, and then going out to a more loosely-coupled interconnect as the computation outgrows its SMP node. Hence, the trend now is to back off from uniform shared memory, and to pay for scalability with nonuniform shared

memory as a programming/performance model. These attempts are called scalable parallel systems (e.g., the IBM SP), massively parallel systems (e.g., ASCI Red), or cluster computers (e.g., the SGI/Cray Origin2000). Nonuniform shared memory results in reprogramming, and extensive retuning to minimize communication cost and enhance cache effectiveness.

With the exception of Intel processors, most commodity processors are conventional Reduced Instruction Set Computing (RISC) superscalar designs. These designs share the design decision that there should be one program counter per processor. The problem is that the limited ILP budget must be spent on both high superscalar issue rate and sustained feeding of multiple function units. The solution as understood was in finding many parallel operations by developing sophisticated approaches to mining individual threads for ILP. Also, aggressive *branch prediction* was a standard technique to increase ILP by creating larger and larger basic blocks.

Nonetheless, the RISC processors in MPPs suffered from latency disease except for easy cases such as applications where either working sets could be held in the cache, or data-access patterns were regular and predictable. Increasing memory nonuniformity in cache-coherent non-uniform memory architecture (CC-NUMA) machines produced an unfortunate coupling between control and data decomposition. One important question is, is it possible to retain the basic design decisions of conventional RISC superscalar processors and extract enough parallelism to overcome latency disease? We think the answer is no. We believe that there are fundamental limits to ILP. ILP needs to be supplemented by other forms of parallelism. We have spent considerable time in this thesis, analysing the consequences of single-program-counter design. We show that as one attempts to extract more ILP, threads become heavyweight, making it impossible to benefit from thread-level parallalism (TLP). Since ILP is limited, so is sustained performance of multiprocessors whose only source of processor concurrency is ILP.

Our abstract goal is to increase the degree of processor-operation concurrency. Cache-based systems rely on massive data locality. RISC superscalar designs and Merced rely on out-of-order execution permitted by ILP. The problem is that ILP is limited. Vector processors rely on long vectors and massive vectorization. Only multithreading allows us to find concurrency in both ILP and TLP. Although we do not discuss Merced, which uses an alternative to speculative branch prediction to get more ILP, and which moves out-of-order control from the hardware to the compiler,

there are reasons to believe that, because it depends on ILP alone for parallelism, it too has no fundamental solution to providing massive concurrency of processor operations. We leave to future work the demonstration that Merced is intrinsically unable to profit from TLP, albeit for very different reasons than the ones that make RISC processors unscalable and unable to tolerate latency. Another problem for future work would be to point out that parallel computers built from Merced suffer the low processor utilization of their RISC based cousins.

1.1.1 Why the crisis is real

The crisis in high-end computing is real because major (e.g., mission-critical) software applications (e.g., structural-analysis simulations) do not scale on existing parallel computers. Urgent large problems exist that do not run well on any existing parallel machine. Many experts believe that large applications with irregular, dynamically changing grids or poor data locality due to algorithmic constraints will not scale on any of today's highly parallel systems.

The difficult larger applications that require the reinvention of computer architecture can be characterized as follows:

1. They require large amounts of time and memory, and produce large amounts of output.
2. Data is organized using irregular meshes rather than simple arrays or grids.
3. Memory accesses exhibit very little spatial and temporal locality.
4. The shape of the problem changes as the computation proceeds.

Thus, while there are many applications that do run well on existing parallel computers, there is a growing class of large problems with novel computational attributes that cannot be solved efficiently unless we succeed in implementing programmable, scalable, general-purpose parallel computers. In this thesis, we explore how massive concurrency is the key to successful implementation of such computers.

Until net-centric computing takes over completely—and perhaps even after then—there are powerful, significant user communities who desperately need massive computing engines. Some customers can, of course, design or buy special-purpose parallel

computers. But, if the parallel computing **industry** is to survive, then market share must grow, and this means that someone has to design, implement, and sell

- easily programmable,
- general-purpose parallel computers (which we may call, servers),
- with multiteraflops (or whatever) performance.

This is our problem domain. The solutions are often guided by some myths that are widely shared:

- Myth 1: There is no way to build a parallel computer unless you use **high-volume commodity processors.** Competing approaches are doomed to failure for reasons of economies of scale. Clients and servers should use the same processors.
- Myth 2: Nowadays, there are no significant architectural differences between one multiprocessor and another. SMPs and CC-NUMA DSM machines have identical cache-coherence protocols. Shared-memory machines are moving to a universal "convergence architecture." If you really need message passing, we'll add it for you! Etc.

These myths are a recipe for stagnation. But, reality is sinking in slowly. In general, the *hardware prerequisites* for *scalable, parallel* servers are:

- scalable bisection bandwidth (which is an interconnection problem),
- scalable latency tolerance (which is a processor problem),
- fine-grained synchronization facilities (which is a processor/memory problem).

Roughly, processors must request and receive data constantly in order to compute. High bisection bandwidth in a multiprocessor allows all processors to do this. Scalable latency tolerance is essential to keep processor utilization high. And, finally, fine-grained synchronization is required to support fine-grained parallelism. All this is intimately related to architectural support for massive concurrency. We do agree though, that not all machines must have these properties; some machines must have these properties. Customers will, for example, buy low- or high-bandwidth machines depending on their needs.

1.2 Little's law

Elementary queueing theory places easily understood limits on any architecture that supports massive concurrency. Little's law says that,

$$C = b \times t$$

where C is concurrency, b is the bandwidth (desired machine performance) and t is the latency of processor operations, where we have taken averages everywhere. This means, for example, that if a processor accepts b operands and delivers b floating-point results per cycle, and if the average flop latency is t cycles, then the number of concurrent flops in progress during each cycle must be at least $C = b \times t$.

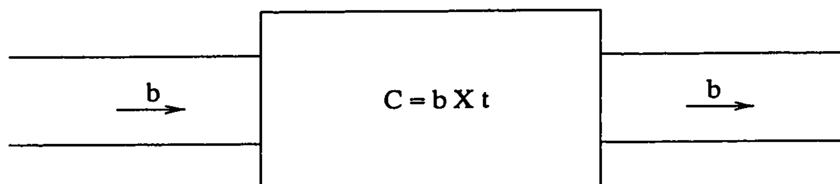

Figure 1.1: Little's law says: concurrency equals bandwidth times latency

Suppose, for example, that we have a processor that is required to sustain 10^{15} operations per second. Suppose also, that the average processor-operation is 10^{-9} second. An inevitable consequence is that the processor must have 10^{6} operations in progress at all times to sustain this bandwidth (that is performance) in the face of this latency. *Concurrency* hence implies and becomes a shorthand for "has large number of operations in progress at all times." We also see that to achieve a petaflops (10^{15}) per second performance we will require memory-reference concurrency of up to a billion outstanding memory references, and up to ten million concurrently executing threads assuming 100 outstanding operations per thread per cycle. We believe that it is possible that superscalar processors are incapable in principle of supporting the large number of real or virtual threads that are required for massively-concurrent parallel computing. The question is **how do we generate this massive concurrency?**

1.2.1 Two ways to go

There are essentially two alternatives that hope to provide massive concurrency:

– having a **single** program counter per processor or,

– having **multiple** (100) program counters per processor.

Our study and analysis in Chapters 2 and 3, proposes that the use of multiple program counters per processor provides support for massive concurrency by sharing the enormous processor state among those multiple program counters (see Section-3.2.3).

1.3 Contributions of this thesis

The major contribution is analytical: we explain why certain architectural distinctions matter, and why. Although we do make some architectural proposals, our real work has been to compose in a *new way* two well-known architectural families: RISC processors and multithreaded processors. Specifically,

1. we explain why conventional RISC superscalar processors *do not* provide architectural support for massive concurrency (see Chapter 2).

2. we explain why multithreaded architectures *do* support fine-grained parallelism and fine-grained synchronization by providing architectural support for massive concurrency (see Chapter 3).

3. we propose a division of labor between the programmer and the compiler in extracting and exploiting parallelism that places a reasonable burden on the complexity-management skills of programmers (see Chapter 4).

4. we make a contribution to parallel programming methodology by extending the *full/empty bit* structure of memory words in the Tera MTA architecture [Tera, SC97] to implement a *Multiword Compare-and-Swap* operation. In particular, we shift the problem of the atomicity of a set of transactional variables to the problem of the atomicity of a single multiword synchronization variable which has been partitioned into counters, thus providing a practical and efficient implementation (see Chapter 4).

In Chapter 2, we explain simply and clearly, with novel analysis, why conventional RISC superscalar processors do not provide architectural support for massive concurrency. Our fundamental conclusion is that, because concurrency is sought exclusively from instruction-level parallelism (ILP), and because data locality is fundamental in determining performance, threads depend on the state built up in both their data cache and the caches that are used for branch prediction. When context switching occurs, this state must be laboriously rebuilt, leading to a long interval during which threads that resume, regain their former performance levels. We quantify all this by including the time to rebuild this *nonarchitectural state* as part of the "effective context-switch time". When this is large, it is the same thing as heavyweight threads, and also means absence of support for fine-grained parallelism—the biggest downside of conventional RISC superscalar processors. This is the fundamental reason that grouping larger and larger number of such processors is unlikely to lead to multi teraflops performance.

In Chapter 3, working as analysts rather than as designers, we explain why multithreaded architectures can support fine-grained parallelism and fine-grained synchronization, and relate the costs of context-switching on a multithreaded machine (which only sometimes involves movement of execution contexts in or out of processors) to the corresponding effective context-switch time of RISC superscalar processors.

Finally, we propose a division of labor in extracting and exploiting parallelism that reconciles the need for truly massive concurrency with the limitations of the human mind in managing complexity. Reasonably, we leave most of the burden to the compiler. The compiler needs some help, however, so we advocate both explicit parallelism (within reason) and compiler directives which lead to more effective language translation.

We also make a contribution to parallel programming methodology by extending the *full/empty bit* structure of memory words in the Tera MTA architecture to design a *Multiword Compare-and-Swap* operation. The primary value of this is to allow database-like programming where agreed-upon lock orderings are hard to come by, when we do not wish to be bothered by the danger of deadlocks.

Other authors [Herlihy, Moss 91] [Herlihy, Moss 93] have suggested implementing *Compare-and-Swap-Two* using the conventional hardware-managed cache-coherence protocol. We do not use such protocols in multithreaded machines, and find an efficient dynamic (i.e., addresses of shared transaction variables are known only at

runtime) implementation that allows us to do transactional programming in the spirit of optimistic concurrency control. Apart from the sector data-cache, the only new hardware instructions we require are a few multiword synchronization variables and the ability to increment indexed fields in these shared variables. The fully dynamic solutions do require some address registers so that shared variables whose addresses are disambiguated at runtime can be associated with counters in the multiword synchronization variable.

Chapter 2

Superscalar Architectures

"He is one of those people who would be enormously improved by death."

— H. H. Munro

Since the beginning of this decade, seeking microprocessors with higher and higher performance has led to many advanced microarchitectural designs. There is a heavy emphasis at present on **superscalar microprocessors** [Smith, Sohi]. These are *multiple-issue* machines which contain a number of micro-architectural features specifically designed to achieve concurrency by exploiting the parallelism contained in a program at the instruction level called *instruction-level parallelism (ILP)* [Jouppi, Wall]. ILP is the parallelism within single threads. A general outline of superscalar machines is given in Section 2.2.

The major drawback of the ILP processors is that they create *heavyweight* hardware threads leading to slow context-switching and poor fine-grained synchronization which often results in the disciplined avoidance of both features. The amount of parallelism that can be exploited depends on the controlflow and also on the dataflow inherent in the code. There are problems not only with context switching but also with frequent branches. Frequent branching, for example, limits the performance of a superscalar processor due to the maintenance of a large number of registers and cache footprints. We discuss this issue in detail in Section 2.4. Another major design issue is *out-of-order execution* in a superscalar machine which we discuss in Section 2.3.

As systems scale up, the data-locality requirements grow larger and so does the need to avoid and tolerate large-scale latency. Object-oriented or multithreaded applications strain cache capacity, adding to the bandwidth and I/O burden. Caches are essential to parallel superscalar processors and we discuss them briefly in Section2.1.

2.1 Caches

One of the major obstacles limiting application performance of superscalar machines is long latency of remote memory operations. Wide-issue multiprocessors issue multiple instructions per clock cycle. Instruction pipelines must be well fed in order both to avoid bubbles in the pipelines and to maximize performance. As the gap between processor and memory speeds continues to grow, there is need to develop and exploit techniques to avoid or tolerate such memory latencies, thereby improving processor performance.

Caching of data has been the most popular way to avoid memory latency. A cache is a small, higher speed memory system which stores the most recently used instructions or data from a larger but slower main memory system. The larger the cache, the more instructions and data it can store, and higher is the probability of finding the data or instruction in the cache. When a memory request does not find an address in the cache—a *cache miss* is incurred; whereas, if the search is successful a *cache hit* is supposed to have occurred. An nth level cache is $n-1$ levels away from the CPU. The *first level* (L1) or *primary cache* is the fastest in the memory hierarchy. It resides on the processor chip itself and runs extremely fast. Each time the processor requests information from memory, the cache controller on the chip uses special circuitry to first check if the memory data is already in the cache. If it is, then the system is saved from accessing the slow main memory. Most computers also use a *secondary* or *second level* (L2) cache, to catch some of the recently used data that does not fit in the smaller primary cache. The L2 cache is much larger but also much slower than the primary cache. Most modern processors, for example Pentium II processors actually have both L1 and L2 caches built into the processor chip. The rest of the $n-2$ higher level caches reside outside the processor chip. Caches each level away from the processor are typically larger and slower than the levels closer to the CPU.

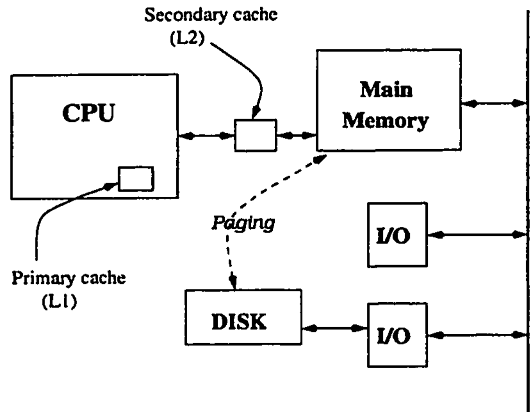

Figure 2.1: Memory hierarchy with primary (L_1) and secondary (L_2) caches

The first level cache is relatively small and has split instruction and data caches. It is usually direct-mapped or, has low associativity. L2 is much larger, is shared by both instructions and data, and usually has high associativity. L1 and L2 caches are located on-chip for fast access times. In case of a cache miss, the processor stalls until such time as the memory address is found in one of the cache levels. Figure 2.2 shows a flow diagram of a blocking cache, which stalls the CPU in case of a cache miss.

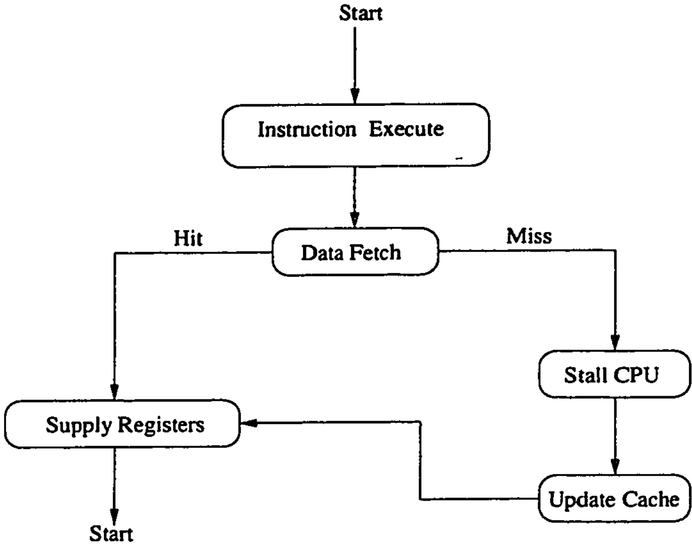

Figure 2.2: Flow Diagram of Lockup or Blocking Cache

As remedies *lockup-free* or *non-blocking* caches [Kroft] (see Figure 2.3) were designed to enable processing cache-accesses inspite of cache misses. Special registers

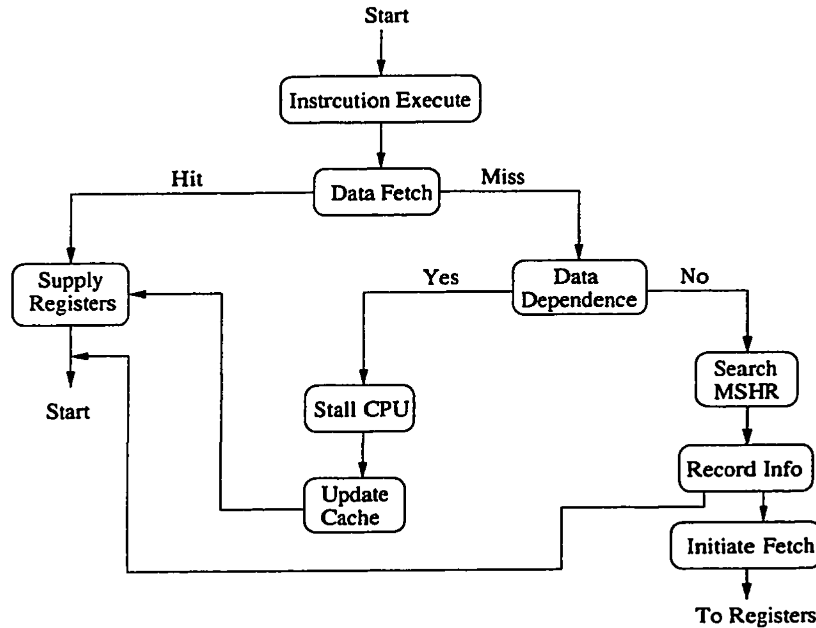

Figure 2.3: Flow Diagram of a Lockup-free or Nonblocking Cache

called *miss status holding registers* (MSHR's) along with the necessary control logic, contain adequate information to enable the processor to overlap processing of a cache miss with processing of subsequent instructions in the instruction cache. Nevertheless, with rise in the number of MSHR's, complexity and cost associated with non-blocking schemes rose quite rapidly.

In case of a cache miss there are overheads incurred in searching for the memory address in the cache. A single cache miss taking place is pipelined to higher and higher levels of caches until there is a cache hit—the processor remaining stalled until such a time. Thus, if the hit rate is too low, the overhead incurred as a result is very high and the processor stalls frequently.

Recently, integration of the two cache levels L1 and L2 into a single cache has been successful. Nevertheless, this has a major disadvantage in that the size of the cache footprint grows massively, which in turn causes context-switching to become increasingly more expensive.

The faster the processor, the more acute is the need to keep them fed with newer instructions and data. More and more instructions and data require more and more levels of cache in order to avoid *misses*. In events of higher context-switchings there

are overwhelmingly large cache-footprints making context-switching sluggish. Moreover, the problem of available memory bandwidth becomes all the more acute when systems scale up, given the memory overhead associated with building large cache-coherent systems. Caches are useful only to the extent that cache-lines are small and are managed by users and compilers rather than automatic cache-coherence protocols.

Thus, the state penalty of context-switching makes caches a bad implementation technique in avoiding latency tolerance.

2.2 Superscalar techniques to implement and exploit concurrency

Superscalar processing is the technique used to fetch, decode and execute multiple instructions in parallel *(out of order execution)* from an instruction stream executing on a single processor. Typically, the state of the instruction stream is stored in register files, various buffers and a single program counter. Indeed, we will use the term *instruction-level parallelism* (ILP) in the restricted sense of extracting and exploiting the parallelism in an instruction stream using a single program counter as architectural support. Though viewed by many as an extension of the Reduced Instruction Set Computer (RISC) [MIPS RISC] movement of the 1980's, superscalar implementations have forever been heading toward increasing complexity. These superscalar methods have been applied to a wide spectrum of instruction sets, ranging from the DEC Alpha [DEC Alpha], to the latest RISC instruction set, to non-RISC Intel $x86$ instruction set.

2.2.1 Historical perspective

Pipelining of instructions in order to exploit instruction-level parallelism has been in use for many decades. A pipeline acts like an assembly line with instructions being processed in phases, as they pass down the pipeline. With simple pipelining, only one instruction at a time is initiated into the pipeline, though multiple instructions may be in some phase of execution, concurrently. The CDC 6600 used some degree

of pipelining and achieved most of its instruction-level parallelism through parallel functional units. The 360/91 was heavily pipelined, and although it provided a dynamic instruction-issuing mechanism known as *Tomasulo's algorithm,* could sustain only a single instruction per cycle and was not a superscalar processor. The pipeline initiation rate of one instruction per cycle was perceived to be a serious practical bottleneck. There exist several implementation techniques for exploiting instruction-level parallelism, such as vector processing, multiprocessing and others. Superscalar processors succeeded in eliminating the *single-instruction-per-cycle* bottleneck by initiating more than one instruction per cycle. Ever since their invention in mid 1980s, superscalars became the standard method for implementing high-performance microprocessors.

2.2.2 Model of a superscalar processor

In the sequential execution model, a program counter is used to fetch a single instruction from memory. The instruction is then executed and the program counter incremented to fetch the next consecutive instruction from memory or a nonconsecutive instruction in case of a branch or jump instruction.

A superscalar machine eliminates much of the non-essential sequentiality (out-of-order execution) to turn the program into a parallel, high-performance version, yet retaining the outward appearance of sequential execution. Often, instructions are written in sequence in a program because of the language syntax with no implication that they need be executed in that order. The sequential program text, in effect masks the intended program order. A number of techniques including *dependence analysis*, attempt to recover the *intended* (essential for correctness) program order from the sequentially-written program order. Although this is certainly not an algorithm for uncovering it, formally we may say that the essential partial order among instructions (say, within a basic block) is conserved.

Parallel processing of instructions involves the following stages:

1. **Instruction fetch**
   Simultaneously fetching multiple instructions (which are partially predecoded) for later decode, using aggressive branch prediction (state explosion), and out-of-order issue (hardware complexity) and an instruction cache (conservative and

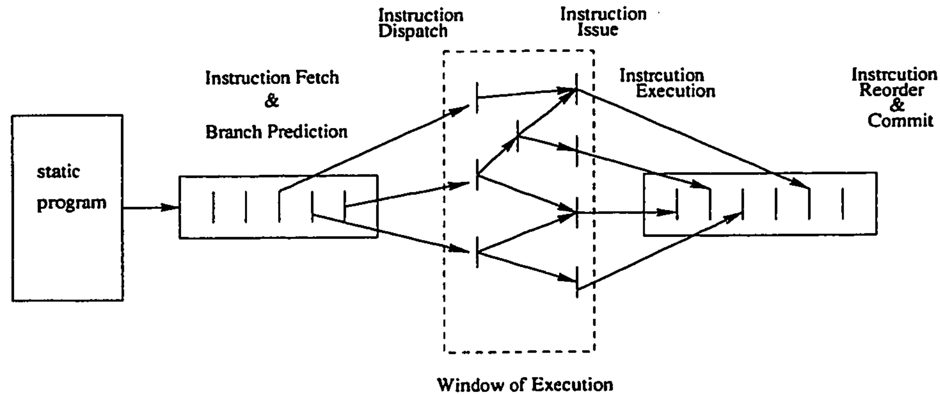

Figure 2.4: Stages of parallel processing in a superscalar execution.

well accepted).

2. **Instruction decode**
   Breaking apart instructions for determining true dependencies and other types of dependencies.

3. **Dispatch and schedule**
   Initiating or issuing multiple instructions in parallel.

4. **Execute**
   Contains the functional units (FUs).

5. **State update**
   Handling *(precise)* interrupts and instruction reordering.
   Maintaining the process state in a coherent sequential order and implementing precise interrupts in the presence of out-of-order execution and superscalar execution.

Figure 2.5 illustrates the microarchitecture, or hardware organization of a typical superscalar processor. A static program in essence describes a set of executions, each corresponding to a particular set of data that is given to the program. Implicit in the static program is the sequencing model, i.e., the order in which the instructions are to be executed.

As a static program executes with a specific set of input data, the sequence of executed instructions forms a dynamic instruction stream. As long as instructions

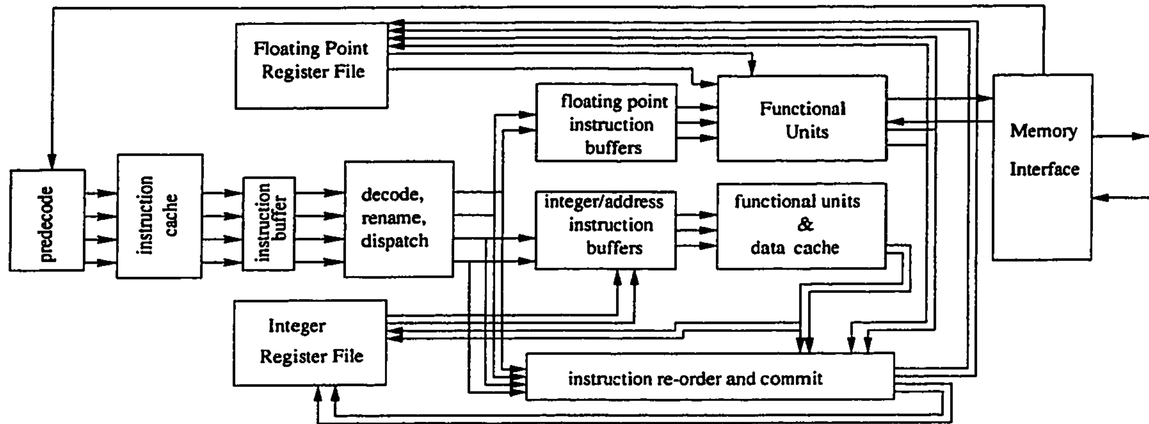

Figure 2.5: Organization of a Superscalar Processor.

to be executed are consecutive, static instructions can be entered into the dynamic sequence simply by incrementing the program counter, which points to the next instruction to be executed. When there is a conditional branch or jump instruction the program counter is updated to a non-consecutive address. An instruction is *control dependent* on its preceeding instruction(s) if the flow of program control must pass through the preceeding instructions first.

The most vital step in increasing instruction-level parallelism is to overcome control dependencies incurred due to incrementing or updating program counters. A static program can be taken as a collection of *basic blocks*—a contiguous block of instructions with a single entry point and a single exit point. Once a basic block has been entered by the instruction fetcher, all instructions in the basic block are executed eventually, i.e., any sequence of instructions in the basic block can be initiated into a conceptual *window of execution,* and they are free to execute in parallel subject merely to data dependence constraints.

Control dependencies due to updates of a program counter, especially due to conditional branches, must be overcome to get more parallelism. One way to do this is to *predict* the outcome of a conditional branch and *speculatively fetch and execute* instructions from the predicted path. Instructions from the predicted path are moved into the *window of execution.* If the prediction is later found to be correct, then the speculative status of the instructions is removed, and their effect on the state is the same as any other instruction. If the predicted path is later found to be incorrect, the speculative execution was wrong, and recovery actions must be initiated so that

process-state is not corrupted; all effects of the wrong speculative execution (misspeculation) are nullified.

Instructions in the window of execution begin execution subject to ***data dependence*** constraints. The precedence requirements in executing some instructions is that operations can not be issued before their operands are available. Data dependencies occur among instructions due to the possibilities (hazards) of these instructions accessing the same storage location. Ideally, instructions can be executed subject only to ***true dependence*** constraints. These true dependencies appear as ***read-after-write(RAW)*** hazards, so that the consuming instruction can only *read* the value *after* the producing instruction has written it.

There is also a possibility of having ***artificial dependencies*** which have to be overcome during the execution of the program to increase the available level of parallelism. These artificial dependencies result from *write-after-read(WAR)*, and *write-after-write(WAW)* hazards.

After resolving these kinds of dependencies, instructions are issued and begin execution in parallel. In essence, the hardware forms a *parallel execution schedule* which takes into account necessary constraints, such as true dependencies and hardware resource constraints of the functional units and data paths.

A parallel execution schedule often implies that instructions complete in an order diffrent from that indicated by the sequential execution model. Thus, architectural storage cannot be updated immediately when instructions complete execution. Rather, the results of an instruction must be held in a temporary status until the architectural state can be updated. Eventually, when it is determined that the parallel model conformed to the sequential execution model, the temporary results are made permanent by updating the architectural state. This process is called *committing or retiring* the instruction.

2.2.3 Instruction fetching and branch prediction

The multiple instructions fetched in the instruction fetching phase feed the rest of the pipeline in a superscalar. A small memory called an *instruction cache* (different from data cache) containing the most-recently used instructions, reduces latency and

increases bandwidth of the instruction fetching process. The instruction cache is organized into *blocks* or *lines* containing several consecutive instructions. The program counter is used to search the cache contents associatively to determine if the instruction being fetched is present in one of the cache blocks; if present there is a cache hit, if not a cache miss and the instruction is fetched from the main memory.

The number of instructions thus fetched per cycle should be equal to at least the peak instruction decode and execution rate. The extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer than the maximum number of instructions can be fetched. There is often an *instruction buffer* designed to hold multiple fetched instructions in order to feed the processor pipeline during periods of instruction fetch restriction or processor stall.

The default mechanism of fetching, involves incrementing the program counter by the number of instructions fetched and fetching the next block of instructions [Lee, Smith]. In case of branch instructions which alter the flow of control, the fetch mechanism must be redirected to get instructions from the *branch target buffer(BTB)*.

Processing and handling of branch instructions can be broken down into the following parts:

- Recognizing a conditional branch

  Special instruction decode information is contained in some extra bits in the instruction cache, which assists in speeding up the process of identification of all instruction types not just branches. These extra bits allow very simple logic to identify the basic instruction types. For example, the *pre-decode logic* generates pre-decoded bits and stores them alongside the instructions as they are placed in the instruction cache.

- Determining the branch outcome (*taken* or *not-taken*)

  In case of data dependence between a fetched branch instruction and a preceeding uncompleted instruction, the outcome of a conditional branch can be predicted using one of the several types of *branch prediction techniques.* rather than waiting for the uncompleted instructions to terminate. There are broadly, two types of conventional branch prediction techniques, namely—

  - *Static branch prediction* using information from the binary which is put there by the compiler. Often the profiling information, or the program

execution statistics collected during previous runs of the program are used by the compiler as an aid for static branch prediction.

  - *Dynamic branch prediction* where, as program executes more information becomes available, which is used for further prediction. A *branch history table* or *branch prediction table* contains information regarding the past history of the branch outcomes. These tables are typically organized in a cache-like manner and accessed with the address of the branch instruction to be predicted.

- Computing the branch target

  Branch targets are computed relative to a program counter and using an offset value held in the instruction, in order to eliminate the need for read-registers. This requires a program counter and a number of register counters which slow down the process. A branch target buffer (BTB) that holds the target address that was used the last time a particular branch was executed, can be used to assist in finding the target address.

- Transferring control by redirecting fetch (taken branch)

  There can be bubbles in the instruction pipeline if there is a delay in recognizing the "taken branch," modifying the program counter, and fetching instructions from the target buffer. Usually, instruction buffers are used to minimize such delays and hence avoid bubbles in the pipeline.

2.2.4 Instruction decoding, renaming, and dispatch

In this phase of execution, instructions are removed from the instruction fetch buffers and examined. Control and data dependence linkages are then set up for the remaining pipeline phases. True data dependencies (RAW hazards) and resolution of other register hazards (WAW, WAR) are detected.

Instructions that have been placed in the window of execution may begin execution, subject to *data-dependence* constraints. Data dependencies occur among instructions because the instructions may access (read or write) the same storage (a register or memory) location. When instructions reference the same storage location, a *hazard* is said to exist— i.e., there is the possibility of incorrect operation unless

some steps are taken to make sure the storage location is accessed in the correct order. Ideally, instructions should only be ordered according to *true dependence* constraints. These true dependencies appear as *read-after-write (RAW)* hazards, because the consuming instruction can only *read* the value *after* the producing instruction has written it.

Artificial dependencies result from *write-after-read (WAR)* and *write-after-write (WAW)* hazards. A WAR hazard occurs when an instruction needs to write a new value into a storage location, but must wait until all preceding instructions needing to read the old value have done so. A WAW hazard occurs when multiple instructions update the same storage location; it must appear that these updates occur in proper sequence. Artificial dependencies can be caused in a number of ways, for example, by unoptimized code, by limited register storage, by the desire to economize on main-memory storage, and by loops where an instruction can cause a hazard with itself. For example, here is some dummy machine code:

```
r3 <--- r7              // move r7 to r3
r8 <--- ^r3             // load word at r3
r3 <--- r3 + 4          // increment r3
```

The *move* instruction produces a value in $r3$ that is used by both the *load* and *add* instructions. A dynamic execution must ensure that accesses to $r3$ made by instructions that occur after the *add*, access the value bound to $r3$ by the *add* instruction. Moreover, it must ensure that the value of $r3$ used in the *load* instruction is the value created by the *move* instruction.

Thus, there are mainly two types of instruction dependencies, namely, control-flow and dataflow dependencies. A possible change in control flow causes a control-flow dependency. For instance, a branch instruction that does not always go in the same direction causes a control-flow dependency on instructions that follow it, because the execution of the branch instruction determines whether these instructions execute. Control-flow and dataflow dependencies can be explained as shown in Figure 2.6.

The directed graph has nodes representing instructions and edges that connnect instructions that may follow one another. In Figure 2.6 the statements $X = 3$ and $X = 4$ are both control dependent on the statement $IF(X = 0)$. However, note that $Y = X$ is not control dependent on any of the other instructions because it executes

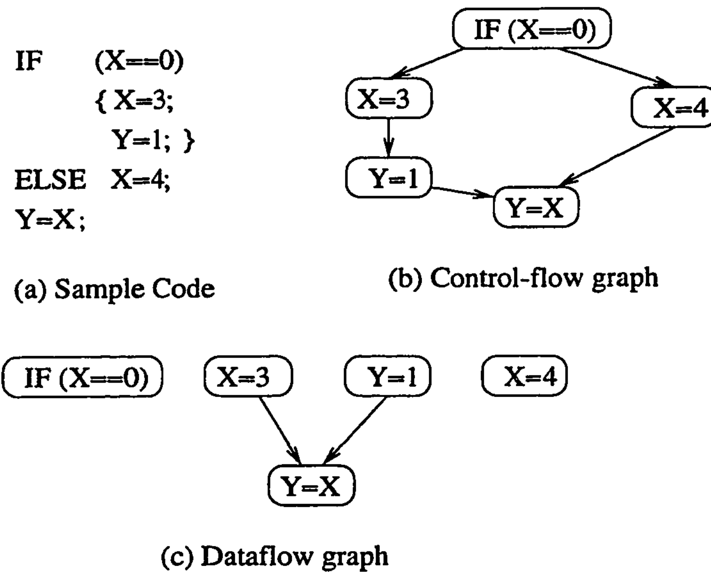

Figure 2.6: A Sample Program with corresponding Control-flow and Dataflow Graphs.

regardless of the path taken from the IF statement.

In the example, the statement $Y = X$ has a data dependency with both statements $X = 3$ and $X = 4$. True dependencies are typically viewed in a data flow or data dependency graph. Each node represents an instruction and each edge represents a true data dependency in Figure 2.6. In the example, the statement $X = 3$ causes an antidependency with the statement $IF(X == 0)$. The result of the second update should be seen by all instructions following the second instruction. In the example, the statement $Y = 1$ has an output dependency with the statement $Y = X$.

A parallel-execution schedule often means that instructions complete execution in an order different from that dictated by the sequential-execution model. Consequently, the architectural storage (the storage that is outwardly visible) can not be updated immediately after instructions complete execution. Rather, the results of an instruction must be held in a temporary status until the architectural state can be updated. Meanwhile, to maintain high performance, these results must be used by dependent instructions. Eventually, when it is determined that the sequential model would have executed an instruction, its temporary results are made permanent by updating the architectural state. This process is typically called *committing or retiring* the instruction. This instructions are dispatched into the window of execution, issued from the window of execution as allowed by dependences, complete, and are

finally reordered at the time they commit.

The decode phase sets up one or more tuples for each instruction. Each tuple consists of information like what operands to execute, the location where the operands can be found and where to store the results. These locations are generally memory locations or logical registers.

The *register renaming* methods are used to replace the logical register name in the instruction's execution tuple with the real name of the physical storage location. A method of register renaming uses a physical file of the same size as the logical register file and a one-to-one relationship is maintained. In addition, there is a buffer called the *reorder buffer* with one entry per active instruction (an instruction that has been dispatched but has not yet been committed). The reorder-buffer stores instructions that have been dispatched but have not yet been committed, is used to maintain proper instruction ordering for *precise interrupts.* As the number of instructions become large, the size of the reorder-buffer becomes increasingly large leading to larger state. Reorder-buffers are discussed in detail later in Section2.3.

2.2.5 Instruction issuing and parallel execution

An *execution tuple* consisting of opcode plus physical register storage locations is formed in the decode/rename/dispatch phase. Once these tuples have been created and buffered, they are examined to decide upon which ones to be *issued* for execution. Instruction issue is defined as runtime checking for the availability of data (operands) and resources such as execution units, interconnect, and register file (reorder buffer) ports. This is an area of the processing pipeline which contains the "window of execution" and is at the heart of many Superscalar implementations.

Instruction issue buffers can be organized in a number of ways in order of increasing complexity, namely—

— Single, Shared Queue: There is a single queue with no out-of-order issue and no register renaming. A register is reserved for any instruction that modifies the register after being issued. This register is cleared when such an instruction completes. Instructions may be issued as soon as operands are available.

— Multiple Queue Method (Out-of-order execution): Multiple queues (one per

instruction type) may issue out-of-order with respect to one another, though instructions from each queue are issued in order. The individual queues are organized according to their instruction type, for example, floating point instruction queue or integer instruction queue or a load/store instruction queue.

— Multiple Reservation Stations one per Instruction Type: The idea of reservation stations was first proposed in Tomasulo's algorithm wherein, instructions were supposed to be issued out-of-order, i.e., there was no strict FIFO ordering. Thus, all reservation stations simultaneously monitor their source operands for data availability. Reservation stations hold the operands or pointers to the register file where the operands may be found. When all operands for an instruction are ready in the reservation station then the instruction may be issued.

2.2.6 Handling memory operations

To reduce memory latencies, memory hierarchies are used so that most data requests will be serviced by data cache memories (Section 2.1) residing at the lower levels of the hierarchy. It is not possible to identify the memory locations that will be accessed by load/store instructions until after the issuing phase. Address calculation and address translation are required to generate the physical address of a memory location. A *translation lookaside buffer(TLB)* which is a cache of translation descriptors of recently accessed pages is used to speed up the addcss translation process. Once a valid memory address has been obtained, the load or store operation can be submitted to the memory.

In most fast Superscalars, the idea has been to overlap the address translation and memory access. The initial cache access is done in parallel with address translation and the translated address is then used to compare with the cache tags to determine if it was a cache hit. The key issue then is to allow memory operations to be overlapped, or to proceed out-of-order to ensure that hazards are properly resolved and that sequential execution semantics are preserved. *Store address buffers*, or *store buffers* are used to make sure that operations submitted to the memory hierarchy do not violate hazard conditions. The store buffers contain addresses of all pending store operations. Prior to issuing an instruction to memory, the store buffer is examined to see if there is a pending store to the same location.

2.2.7 Committing or retiring instructions

This is the final phase of the lifetime of an instruction where the effects of the instruction are allowed to modify the logical process state. This phase implements the appearance of a sequential execution model even though the actual execution is non-sequential due to out-of-order execution and speculative execution. The actions necesssary in this phase depend on the techniques used to recover a precise state, discussed in detail in Section 2.4.

The state of the machine is saved or *checkpointed* in history buffers. Instructions update the state of the machine as they execute and when a precise state is needed, it is recovered from the history buffer. Finally, in this phase the history buffer is discarded because it is no longer needed.

The state of the machine is separated into two parts, namely, the implemented *physical state* which is updated immediately as the operations complete, and a *logical state* which is updated in the sequential order as the speculative status of operations is cleared. The speculative state is maintained in the reorder buffer from where it is moved into the architectural register file or memory, and space is freed up for the reorder buffer.

In most superscalar implementations the reorder buffer is used as a method of register renaming, though it also holds useful control information used to move physical registers to the free list. In case of interrupts, the control information is used to adjust the logical-to-physical mapping table so that the mapping reflects the correct precise state.

2.3 Out-of-order issue/execution of instructions

The in-order-issue of instructions implies that instructions be issued sequentially in order as they appear in the program code. The problem with in-order-issue is that the processor stops decoding instructions whenever there is a dependency or the functional units are busy. Such wastage of CPU time can be avoided by just keeping decoding instructions and issuing them when dependencies have been resolved and busy functional units are clear. This is accomplished by simply adding an instruction window or buffer between the decode and execution stages. Instructions are issued

from the window whenever all dependencies are clear and a functional unit is available, regardless of instruction order as specified in the code.

The technique however introduces the problem of antidependencies or WAR hazards which happens when one instruction requires a register value for input, but a later instruction writes to this register. If the execution of these instructions are reversed then the first instruction reads a wrong value from its source register.

The key issue in allowing memory operations to be overlapped, or to proceed out-of-order is to ensure that hazards are properly resolved and that sequential execution semantics are preserved. Register renaming can be used to overcome both output and antidependencies. If register $R4$ is assigned a value, another register $R4'$ is allocated and all reads of register $R4$ in the future are directed to $R4'$. When a new assignment to register $R4$ is made another register $R4''$ is allocated and this process continues.

Out-of-order execution implies the extensive use of *reorder buffers* to maintain proper instruction ordering for precise interrupts. Should the execution of the program need to be interrupted and restarted later in case of external or internal interrupts, the state of the machine needs to be captured. The sequential execution model has led to the idea of precise state.

In case of an interrupt, the pipelined instruction may modify the process state in an order different from the sequential order. The state of the interrupted process is saved by the hardware or software or a combintion of both. The process states consists of a program counter, registers and memory. If the saved process state is consistent with the sequential architectural model than the interrupt is said to be a *precise interrupt.*

There can be two kinds of interrupts, the first one being called *program-interrupts or traps* which result from exception conditions detected during fetching/execution of the program. These are caused by an illegal opcode, overflow of arithmetic data or page faults. The other kind called *external interrupts* are caused by sources outside the executing process, for example, I/O or timer interrupts. At the time of an interrupt, a precise state of the machine is the state that would be present if the sequential execution model was strictly followed and processing was stopped precisely at the interrupted instruction. Restart can then be made by simply executing that interrupted instruction.

The reorder buffer can be imagined as a FIFO queue implemented in hardware

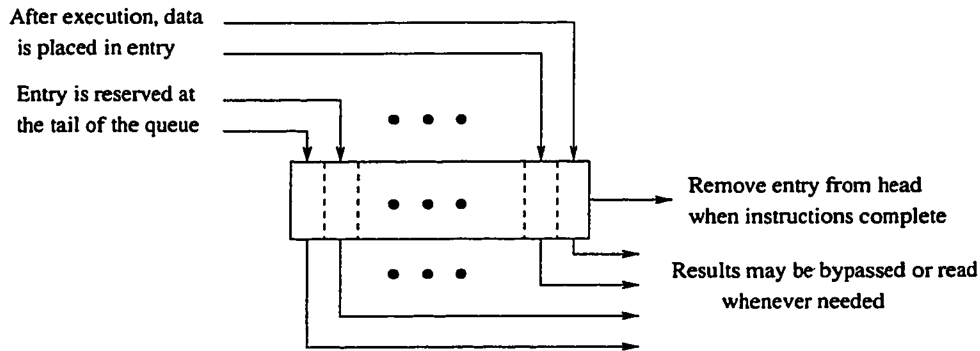

Figure 2.7: A reorder buffer.

as a circular buffer with head and tail pointers, and gets entries as instructions are dispatched according to sequential program order. As instructions complete execution, the results are inserted into their previously assigned entry, wherever it may happen to be in the reorder buffer. At the time an instruction reaches the head of the reorder buffer, if it has completed execution, its entry is removed from the buffer and its result value is placed in the register file. An incomplete instruction blocks at the head of the reoder buffer until its value arrives.

The disadvantage of using reorder buffers is that as more and more instructions are pumped into the instruction pipeline, the size of the reorder buffer grows substantially. The buffer must be able to store the results of all outstanding operations. Vendors regularly increase the size of their reorder buffers with successive processor generations. Think of the total register size as proportional to the amount of ILP being used.

2.4 Speculative Execution

When a branch is encountered in the instruction stream, it can cause a change in control flow. This change in control flow can cause branch penalties that can sigficantly degrade the performance of superscalar pipelined processsors. To overcome these penalties, these processors typically provide some form of control-flow prediction. Speculative execution means the execution of code at one or more destinations of a branch before the branch outcome is known [Calder et al.]. The basic purpose

of speculative execution is to speed up program execution by running some code segments before it is known whether or not they are actually taken. This technique is primarily used in trying to prevent branches from disrupting long execution pipelines. Speculative execution is an integral part of modern ILP processors, be they statically- or dynamically-scheduled designs. Speculation may take two forms, namely, control speculation and data speculation. Control speculation implies the execution of an instruction before the execution of a preceding instruction on which its control is dependent. Data speculation implies the execution of an instruction before the execution of a preceding instruction on which it may be or is data dependent.

Control flow speculation uses several branch prediction mechanisms to enhance performance of the superscalar machines. Opportunities for parallelism can be increased by performing *control-dependent analysis*, in which a program trace is analyzed to decide on exactly which branch each block is dependent. Speculative execution implies executing the code at one or more destinations of a branch before the branch outcome is known. The following steps are involved:

- Adding branch prediction to processor core.
- When branch is predicted, begin executing instructions until outcome is known.
- Mark these speculative instructions in the reorder buffer.
- If the branch is predicted Taken(NotTaken) and it is actually NotTaken(Taken), then:
  - Flush all instructions from branch to end of reorder buffer.
  - Resume execution at correct path of branch.

When there is a taken (or predicted taken) branch there is often at least a clock cycle delay in recognizing the branch, modifying the program counter, and fetching instructions from the target address. This delay results in pipeline bubbles unless remedied steps are taken. The most common solution is to have an "instruction buffer" with an abundance of instructions to mask the delay. Still complex buffers contain instructions from both the taken and not-taken paths of the branch.

In case of mispredictions the penalty is very high. All instructions along the wrong path and the modifications in the state made by them have to be squashed leading to

a wastage in processor cycles.The size of the Branch Translation Buffer (BTB) grows exponentially with the number of branch predictions made due to the combinatorial explosion of nested branches.

To expose parallelism that is hindered by ambiguous dependences, data dependence speculation may be used. In data dependence speculation the load is allowed to execute before a store on which it is ambiguosly dependent. If no true dependence is violated in the resulting execution, the speculation is successful. If, however a true dependence is violated, the speculation is erroneous (i.e, mis-speculation). In the latter case the effects of the speculation must be undone. Consequently some means required for detecting erroneous speculation and for ensuring correct behaviour. As window sizes grow larger, the mis-speculation becomes more frequent, and the cost of mis-speculations becomes very high.

2.5 Summary of the disadvantages of superscalar machines

Superscalar processors because of their ambitious attempts to exploit ILP have steadily raised the heaviness of threads to the point where context switching is now a high-cost operation. The process or thread state has grown incrementally with new superscalar techniques. At the bottom, large state is simply large number of registers and large cache-footprints. As other techniques, such as speculative execution are added, state includes new things like branch-target buffer foot-prints. The state mostly affects context switching, but superscalar features also have large negative impact on programs with frequent conditional branching. The result is that we may be shy about programming with either frequent context switching (i.e., fine-grained parallelism with fine-grained synchronization) or fine-grained conditional branching.

Chapter 3

Multithreading

"Everything comes to him who hustles while he waits."

— Thomas Edison

The problem that multithreading comes to solve is latency tolerance at all levels of granularity. Both SRAM and DRAM speeds are increasing much slowly than the processor speeds—in fact the gap between processor and memory speeds is growing exponentially. Small operations and large operations alike incur some non-zero latency. For example, memory references have to wait a finite amount of time before loads or stores complete. The longest latencies come from synchronization operations that do not succeed. Small but important latencies come from (for example) floating-point operations. We will discuss caches as a latency-avoiding technique below, as this is a long story. The waiting time for (remote) memory accessing is called *memory latency* and can be long enough to pose a major threat to the performance of any computer system. Approximately, every third instruction in a modern parallel program is a memory reference. The latency of memory operations is determined by the access time, which is growing exponentially larger than the processor clock cycle time. The numbers are dramatic when memory latency is expressed in clock cycles.

As is well known, when a user program blocks for disk I/O in a multiprogramming environment, the operating system blocks that program and merely switches to some other user program in the ready queue. This overlapping of disk latency with useful

work done by some other program is appropriate for the granularity of the context switching, which is a function of burst size and quantum size. Typically, many machine instructions are required to switch from one process to another. Although the execution time of the blocked process is not reduced in any way, the throughput of the system improves drastically.

Hardware-supported multithreading is the hardware analogue of multiprogramming (which tolerates disk latency). Ideally, in multithreading, independent parallel activities (threads) not only context-switch freely, but also frequently synchronize and communicate with each other. In a multithreaded processor, the essence of multiprogramming is implemented at a much smaller granularity of switching, with a much smaller switching cost, and cheaper and better inter-thread communication. In this way, the focus shifts from switching between jobs to switching between threads, which may well, belong to a single program. Moreover, memory latencies and latencies of functional units in a multithreaded processor are much smaller than disk latencies in multiprogramming; and are usually not visible to the operating system. As result, such latencies can not be dealt with by the operating system, where the cost of context switching that saves and restores processor-state is very high.

A multithreaded processor holds a large number of threads inside separate hardware contexts. A successful multithreaded processor can sustain a high issue rate of instructions from a parallel program. The large number of threads inside hardware contexts makes context switching have an essentially zero cost. In *simultaneous multithreading*, a multithreaded processor issues multiple instructions from multiple enabled threads, and then repeats with another group of enabled threads in round-robin fashion. The central idea, both in traditional fine-grained multithreading and in the newer work on simultaneous multithreading, is to emphasize per-program concurrency. Program concurrency is referred to as *thread-level parallelism (TLP)* while thread concurrency is called *instruction-level parallelism (ILP)*. Multithreading has a radically different philosophy of waiting involving *fast switching*. If context switching can be implemented to be essentially free, a whole New World of performance architectures opens to us. In multithreaded multiprocessors, waiting—for the completion of nonzero-latency processor operations that may not be concurrent with other operations in the program—is accepted as inevitable, and the main idea is to seek to minimize the cost of waiting, not eliminate it. In traditional multithreaded processors, performance is achieved from concurrency of memory accesses rather than from

locality of data, and from continual issue of memory references rather than continual re-placement (i.e., shifting) of data. The general philosophy of waiting is more abstract. It establishes a hierarchy of waiting places/costs; and distributes waiting parallel activities at the appropriate cost-level.

With multithreading there is some hope that we can actually achieve petaflops performance. According to Little's law, assuming average porcessor-operation as 10^{-9} second, with 100 outstanding operations per thread per cycle, we would need up to 10^6 outstanding processor operations in each clock cycle in order to sustain petaflops 10^{15} computing. A general purpose petaflops parallel computer needs a lot of parallelism, and must be equipped with latency-tolerant processors which can concurrently handle about a billion memory references per processor cycle. A high-bisection-bandwidth interconnection network needs to be provided along with a fast memory subsystem for a sustainable high delivery rate of operands to the processors. Today, the long-latency of remote memory operations is a major limiting factor for applications performance. Technology forecasts indicate that memory latencies will improve very slowly, but memory bandwidths will improve very rapidly. From the processor's point of view, the memory is getting slower at a very dramatic rate. The latency problem becomes increasingly serious with high-speed and large-scale multiprocessors, for three reasons—

(*i*) Distance between memory and processors increases,

(*ii*) Remote memory latencies only increase with increasing clock rates as technology advances,

(*iii*) Remote memory latencies also increase with decreasing spatial locality, which is unfortunately more and more prevalent in modern parallel programming.

Latency tolerance has a long history. The four classical techniques for achieving latency-tolerance are:

(*i*) Vector pipelining—requiring the inner loops in the program to be parallelizable and the resulting vectors to be long enough;

(*ii*) Dynamic scheduling—provided that there is instruction-level parallelism in the program, and that the architecture does not lead to excessively heavyweight threads;

(*iii*) Long cache lines—provided that there is good spatial locality that scales;

(*iv*) Simultaneous multithreading—in which many fine-grained and lightweight threads issue operations to multiple functional units in a superscalar fashion.

Vector processors perform operations on sequences of data elements, that is, a vector, rather than scalars. Vector operations support more parallelism within a single thread of control. Their primary benefit is to supply a steady stream of operands to the processor. In addition, as programmers moved along the learning curve, programming with vector processors became easier. Vector processors were implemented in very fast, expensive, high-power circuit technologies. Among the high-end parallel vector processors, three machines stood out,namely, Fujitsu's VPP-700, NEC's SX-4, and Cray's T-90, built from proprietary vector processors with peak performances of 2.2, 2.0 and 80 gigaflops per second, respectively. Parallel vector processors necessarily used large amounts of expensive shared memory in order to keep memory latency low and memory bandwidth up. Still, with vector pipelining, it was far more difficult to improve the scalar-processing rate than it was to improve the vector-processing rate, causing a mismatch between scalar and vector speeds. Most of the more modern scientific and engineering simulations have frequent conditional branches and short-vector or even scalar arithmetic. Clearly, on the average, we continue to find vector lengths that are not long enough to effectively use vector machines. Moreover, the vector systems were not scalable except as Single Instruction Multiple Data (SIMD) systems.

Some people hold the view that vector processors have, over time, been overtaken by superscalar processors (as discussed in Chapter 2), which issue many concurrent operations using a single program counter. As microprocessors are becoming faster, in order to avoid memory latency, the caches in such superscalar systems are becoming larger, and the cache hierarchies are becoming deeper. The size of register files has also been growing rapidly with each new processor generation, in part to hide the predictably increasing latency of the functional unit pipelines, and to support dynamic scheduling. Large size of register files and large cache-footprints lead to what are called *heavyweight threads.* The major disadvantage of superscalar processors is that the actual *architectural (visible)* and *non-architectural (invisible)* state size grow massively. By the architectural or visible state, we mean the processor registers, the program counter, the stack pointer and some per-process parts of the processor status

words. In contrast, by non-architectural or invisible state we mean the caches, the reorder buffer, the branch history buffer (BHB) and the branch translation buffer (BTB). We shall be talking more about heavyweight threads later in this chapter. Though the goal of superscalar processors is to exploit the instruction-level parallelism (ILP) in a thread, the synchronization costs force parallel activities in the programs with large amounts of fine-grained parallelism to communicate rarely.

The simplest solution to memory latency tolerance is the use of cache memory. To reduce the average latency experienced by the processor and to increase the bandwidth, people tried making a more effective use of the levels of memory hierarchy lying between the processor and the DRAM. As the size of memory increases, the access time increases due to address decoding, internal delays in driving long bit lines, selection logic, and the need to use a small amount of charge per bit. All microprocessors today have one, two or more levels of caches on chip. Primarily, caches exploit the spatial locality of data by the use of long cache lines. Though obviously caches have been useful, they are not a panacea. There has been a vast improvement with *lock-up free* or *non-blocking level-1 caches* and *pipelined level-2 caches*, but these linearly better solutions fail to keep pace with exponentially growing problems. Cache misses may still cause the processor to wait for a long time.

Dynamic pre-fetching is based on the idea of predicting the future use of data items and fetching them prior to their use, in order to prevent waiting. When instructions are involved, this is called *dynamic scheduling*. Instructions are scheduled dynamically and allowed to complete *out-of-order* so as to overlap the waiting time of a particular instruction encountering a cache miss, by processing other instructions ahead of the former in the instruction window, as long as they do not depend on the results of the former. More generally, out-of-order execution can be used to tolerate moderate latencies of various sorts. Relaxed memory consistency models are among several other latency-tolerating schemes that were designed. These models allow buffering and pipelining of memory references to loosen the various ordering constraints among memory operations. Despite of these techniques, there still remains a fair amount of latency to be dealt with.

Vector processors, caches, and dynamic pre-fetching techniques have failed to provide a satisfactory solution to the ever-growing latency problem. As chip density is growing rapidly leading to extremely large transistor budgets and high clock speeds,

the idea of exploiting instruction-level parallelism, with only a single thread of control in order to tolerate latency, raises serious doubts. This is where multithreading can play a vital role. The emphasis shifts from instruction-level parallelism (ILP) to thread-level parallelism (TLP). Note that this shift is common to all fine-grained multithreading approaches not just those that use simultaneous multithreading (SMT) (see Section 3.2). That is, we extract and exploit all the concurrency in a program, not just all the concurrency in each thread. Using lots of lightweight threads that issue lots of operations in every processor-clock cycle fulfills the prerequisite for exploiting all of the fine-grained parallelism in a program. For high processor-utilization, instructions should be issued from every slot in every processor, during every clock cycle. Also, it is now becoming necessary to have a good integration of processors and memory, and the interconnections between them, in order to achieve a balanced petaflops design.

3.1 Early multithreading

We now know that by exploiting techniques to avoid or tolerate long-latency memory operations, we can enhance application performance significantly. It is beyond the scope of the thesis to treat latency avoidance or indeed deep memory hierarchies (caches) in general. That is, we defer discussion of ideas for integrating caches with multithreading. We understand from the previous discussion, that vector processors, caches and dynamic pre-fetching techniques have failed to provide a satisfactory solution to the problem of memory latency and memory bandwidth, and that multithreading can play a vital role in providing a solution to this problem.

Multithreaded or multiple-context processors tolerate latency by overlapping the non-zero latency operations of one thread of computation with the execution of operations from other threads. The idea is to share a single processor among several threads of computation and tolerate the waiting time of one thread by switching contexts and executing another ready thread during that period. This kind of frequent context switching needs to be radically inexpensive which obviously requires hardware implementation.

The first multiple-context multiprocessors were used in CDC 6600, back in the early 1960's in order to time-share a CPU and memory interconnect between a number

of peripheral processors. Multiple-context processors such as Denelcor HEP used fine-grained multithreading and switched contexts every cycle, making the cost of context-switching essentially zero. Though this low switching cost allowed all forms of latency tolerance, the designs were limited to single-instruction pipelines. For full utilization of the model, a large number of threads were required to fill the pipeline and hide memory latency. To address pipeline stalls, a context was prevented from issuing a subsequent instruction earlier than eight cycles after the previous one. There were no pipeline dependences and hence the compiler and hardware were saved from resolving such dependency conflicts. The ability to tolerate the latency of a memory request was dealt with by removing an instruction from the issue queue while memory was being accessed. Data caches were not supported and hence a large number of threads were required to hide both the pipeline and memory latency. Single-thread performance was very poor because of the fact that each thread could issue a new instruction only at every pipeline-depth cycles at best. Some people (quoting Amdahl's law) have argued that even a small serial portion in the program could reduce performance significantly. Such multiple-context processors laid stress on fine-grained architectures, but still ended up sitting idle for a significant amount of time. The ultimate escape from these difficulties lay in tolerating latency by having multithreaded multiple-context processors share a single processor among several threads of computation, overlapping the latency encountered by one thread with useful work done by another thread. Before this solution was invented the emphasis switched from fine-grained multithreading to coarse-grained multithreading.

3.1.1 Coarse-grained or blocked multithreading

The later designs for multiple-context processors suggested by Gupta and Weber [Gupta et al.] share a single processor among several threads. In such a design, a single context utilizes all of the processor's resources until it reaches a long-latency operation (such as a cache-miss, a synchronization event, or a high-latency instruction such as a divide), whereby it switches to another *ready-active* thread. By ready-active, we mean an active processor-resident thread which has been assigned a hardware copy of its register file and program counter and is ready to issue instructions. *Ready (not active)* threads are those that are not stalled and are ready to run but are waiting to

be assigned the appropriate hardware resources. Such *blocked multiple-context processors* provide hardware for a small number of resident threads, but execute only a single thread at any given time. Long-latency operations are masked by switching to another thread. Blocked multiple-context processors are also called *coarse-grained multithreaded processors.* Several early blocked multiple-context processors were proposed for hiding specific long-latency operations or allowing expensive resources to be shared. Although such blocked multiple-context processors address both poor single-thread performance and the need for a large number of contexts of the fine-grained schemes, they do not attempt to reduce the context switch cost to zero. Consequently, they differ from *switch-on-every cycle multithreading*, which can be implemented to provide a better cost synchronization and communication, in addition to comparatively low-cost context switching.

Blocked multithreaded processors use caches for hiding memory latency and do not switch threads on cache hits, thus rendering a good single-thread performance. They also need only a small number of processor-resident threads in order to avoid pipeline stalls. However, the use of caches leads to large cache footprints and hence a large context-switch cost. Blocked multiple-context processors are unable to tolerate smaller latencies as result of the large context-switch cost. The high switch-cost limits the types of latencies that can be tolerated, and places an upper bound on the potential performance of the multiple-context processor. Moreover, the processor utilization increases linearly with the number of threads only up to a threshold, at which point it saturates.

In another attempt to reduce the high switching cost, a few blocked architectures have been proposed which replicate the pipeline registers. The hardware supports several active threads in that there is a hardware copy of each register file and support for other processor resources needed to hold the thread states. This means simply switching from one set of architectural state to another upodn a context switch. The context-switch cost falls to as low as a single cycle but pipeline-register replication results in a substantial implementation and performance cost. The state-size grows larger and hence the need for another technique to deal with the associated problems.

3.1.2 Fine-grained or switch-on-every-cycle multithreading

In fine-grained, or switch-on-every-cycle, multithreading, a new instruction is chosen every processor clock cycle from among different ready-active threads, enabling threads to be switched every cycle rather than only on long-latency events. A variant of this idea developed by Laudon [Laudon] is known as *interleaved multithreading*, which we discuss in the next section. A thread encountering a long-latency event is simply disabled or removed from the pool of ready threads until that event completes and the thread is labeled ready again. There is a cost-hierarchy on which threads *busy-wait*[2] for low-latency events and they become inactive, unready or unavailable, only when a certain threshold is crossed. The main advantage of interleaved multithreading is that there is no context switch overhead because the hardware state is not physically saved or restored. No event needs to be detected in order to trigger a context switch, since this is done every cycle anyway. Segmented or replicated register files are used to avoid the need to save and restore registers. Thus, if there are enough concurrent threads, in the best case, all latency will be hidden without any performance cost, and every cycle of the processor will perform useful work. Nevertheless, the corollary is that fine-grained multithreading is incapable of providing benefits, if implemented with a small, hardware-context replication count.

Fine-grained multithreading has undergone a fair amount of evolution. The early techniques severely restricted the number and type of instructions from a given thread that could be in the processor's pipeline at a time. This reduced the need for hardware pipeline interlocks and simplified processor design, but had negative implications for the performance of single-threaded programs. Interleaved multithreading can be broadly classified in to two categories on basis of whether or not, they use caches to reduce memory latency. The Tera and Horizon do not use caches, whereas the interleaved scheme suggested by Laudon uses caches for hiding memory latency. Next, we discuss each kind in brief.

3.1.2.1 Fine-grained multithreading with full single-thread support and caching

The interleaved scheme of Laudon suggests the possibility of having fine-grained multithreading using caches, and supporting a full single-thread pipeline. In contrast to

[2]Note: This is a programmer specified retry limit which does not imply instruction re-issue.

the blocked approach, the interleaved scheme has a lower context switch cost overhead. Every cycle, the processor selects instructions from a thread from among a set of ready threads, in round-robin fashion. Each thread has its own set of registers and status words. A thread upon encountering a long latency event, becomes unready and hence, unavailable. The pipelines used in this scheme are the same as those used in standard superscalar microprocessors, and have the capability of issuing instructions from the same thread in consecutive cycles without any minimum delay. In the best case, with no pipeline bubbles due to dependences, a $k-$deep pipeline may contain k instructions from a single thread.

The use of caches in the interleaved scheme to reduce memory latency implies a reduction in the number of long-memory latency events, which in turn implies that a given thread is ready and available for a larger fraction of time. The number of threads required to hide latency is thus kept small. This also poses a disadvantage for the scheme in that, there might be many dependent instructions from a single thread in the pipeline as there are not many threads available. On a cache miss, all dependent instructions may have to be squashed from the pipeline making them unready. Typically, the cost associated with squashing dependent threads is smaller than the cost of context switching in the blocked approach. Nevertheless, in order to refrain from squashing dependent instructions in events of cache misses, the number of ready threads may be increased in order to accommodate instructions from different available threads in the pipeline—which in turn leads to a large degree of state replication of the available threads.

The interleaved scheme requires a complex hardware support in order to switch contexts every clock cycle. The processor-state is altered every cycle and the instruction issue unit is capable of issuing instructions from multiple active threads. The advantages of interleaved scheme are found to be the greatest in case of applications that incur a lot of short-memory latencies, such as awaiting results of floating point operations. The most obvious and potential disadvantage of the blocked and interleaved schemes (both use caches), is that multiple threads share the same cache, TLB and branch prediction unit leading to interference between them. In addition, the pipeline squashes significantly increase the effective context switch time.

3.1.2.2 Fine-grained multithreading without caches

The Tera [Tera] and the Horizon multiprocessors do not use caches to reduce latency, relying completely on the latency tolerance provided by multithreading, for all memory references. They have also alleviated the restrictions of allowing only a single memory operation to be issued from a single thread in the pipeline at a time. The current design allows multiple memory operations from a thread to be in the pipeline and be executed in order to achieve peak performance.

Tera supports 128 active hardware-resident threads. It replicates all processor state (i.e., program counter, status word and registers) 128 times, resulting in a total of 4096 64–bit general-purpose registers and 1024 branch target registers. The processor is designed taking locality management away from software, and with the primary focus on supporting latency tolerance through hardware. The Tera system issues three operations per cycle—memory, arithmetic and control (MAC) operations. The packing of operations into wider-instructions is done by the compiler. The hardware chooses a three-operation instruction from a single thread every cycle. Memory operation (M-op) dependences in Tera's Multithreaded Architecture (MTA) are handled using the technique called *explicit-dependence lookahead*—wherein each instruction contains a three-bit look-ahead field that explicitly specifies the number of immediately following instructions within that thread which are independent of that particular memory operation. The analysis of dependences is left to the compiler whereby every instruction is tagged with the three-bit lookahead field. With a three-bit lookahead field, seven is the maximum possible number of instructions that can be explicitly declared as being independent of a particular memory instruction (i.e., simultaneously outstanding from a given thread). This implies that at most eight instructions and twenty-four operations (Tera issues three operations per cycle) can be concurrently in execution from each thread. A particular thread becomes ready to issue a new instruction when all instructions with look-ahead values referring to the new instruction have completed. Thus, with explicit-dependence lookahead, when a long latency event is encountered, the thread does not immediately become unready but can issue more instructions, which are independent of that memory operation, before it becomes unready. Lookahead across one or more branch operations can be done by explicitly specifying the minimum of all distances involved. The major benefit of lookahead is that it allows memory latency to be tolerated within a thread.

It also allows instructions from the same thread to be co-resident in the pipeline, handles register dependence on loads, and handles any sort of dependence between memory operations.

A single-issue thread usually does not have enough instruction-level parallelism to fill in the available issue-slots every cycle, especially in high-width superscalar architectures. Since many threads are available anyway, an alternative is to let available operations from different threads to be scheduled in the same cycle, thus filling the instruction slots more effectively. This is called simultaneous multithreading which is discussed in detail in the next section.

3.2 Simultaneous multithreading

In simultaneous multithreading (SMT) the processor's functional units are fed by multiple instructions issued from multiple independent threads (or programs) each cycle. Unlike conventional multithreaded architectures, which depend on fast context switching to share processor-execution resources, all hardware contexts in an SMT processor are active simultaneously, competing each cycle for all available resources. At least, this is feasible when the number of contexts is small. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking the two major impediments to processor utilization—long latencies and limited per-thread parallelism.

We observe that there are at least two possible interpretations of SMT. The first examines integrating multithreading with conventional superscalar design. This means adopting superscalar ideas, such as caching and speculative execution in a complete fashion *(superscalar-SMT or SMT1)* (see Section 3.2.1). Whereas, the other interpretation proposes to integrate SMT with fine-grained multithreading with modest instruction-issue width *(SMT without heavyweight threads or SMT2)* (see Section 3.2.2). In fairness, there may well be some slight alleviation of problems associated with expensive context switching in conventional superscalar design when SMT is added. Nevertheless, the first school of thought has its own limitations which we shall discuss in the next section. We believe that the integration of multithreading with the gentler forms of the superscalar techniques, which are aggressive precisely when they are directed at extracting maximal single-thread parallelism, would be the

suitable architectural set up of future simultaneous multithreaded microprocessors. Here by "gentleness" we mean, having a moderate amount of ILP—since we aim at compensating for the need of high ILP, by making available more instructions from multiple threads. Hence, the essential idea in future multithreaded processors would be to enable utilization of high degree of context replication in order to make it unnecessary to harvest large amounts of ILP in a thread. Symbolically, we seek only to maximize the product $ILP \times TLP$, not ILP itself.

3.2.1 Simultaneous multithreading with heavyweight threads (SMT1)

Simultaneous multithreading with heavyweight threads and multiple-issue multiprocessors (SMT1) [Lo, Eggers], permits multiple independent threads (or programs) to issue multiple instructions each cycle to the multiprocessor's functional units. SMT1 combines the multiple-issue features of modern superscalars with the latency-hiding ability of multithreaded architectures. Unlike conventional multithreaded architectures, which depend on fast context switching to share processor-execution resources, all hardware contexts in an SMT1 processor are active simultaneously, competing each cycle for all available resources. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking the two major impediments to processor utilization namely, long latencies and limited per-thread parallelism. SMT1 does not compromise single-thread performance.

Whereas wide-issue superscalar processors exploit ILP by executing multiple instructions from a single program every cycle, multithreaded multiprocessors exploit thread-level parallelism (TLP) by executing different threads (or programs) in parallel on different processors. An SMT1 processor accommodates variations in ILP and TLP. That is, when a program has only a single thread (lack of TLP), all of the SMT1 processor's resources can be dedicated to that single thread. Whereas the existence of more TLP in the program can compensate for lack of per-thread ILP. Simple multiple-issue processors suffer from two inefficiencies. First, due to limited ability to find and exploit instruction-level parallelism, not all slots are filled in a cycle. Second, due to long-latency instructions, many cycles go wasted without any

useful work done. Simple multithreading though addresses both long- and short-memory latency problems it fails to provide a concrete solution to the first problem. An SMT1 processor is capable of exploiting either kind of available parallelism and utilizes the functional units more effectively, hence achieving better throughput and higher speedup.

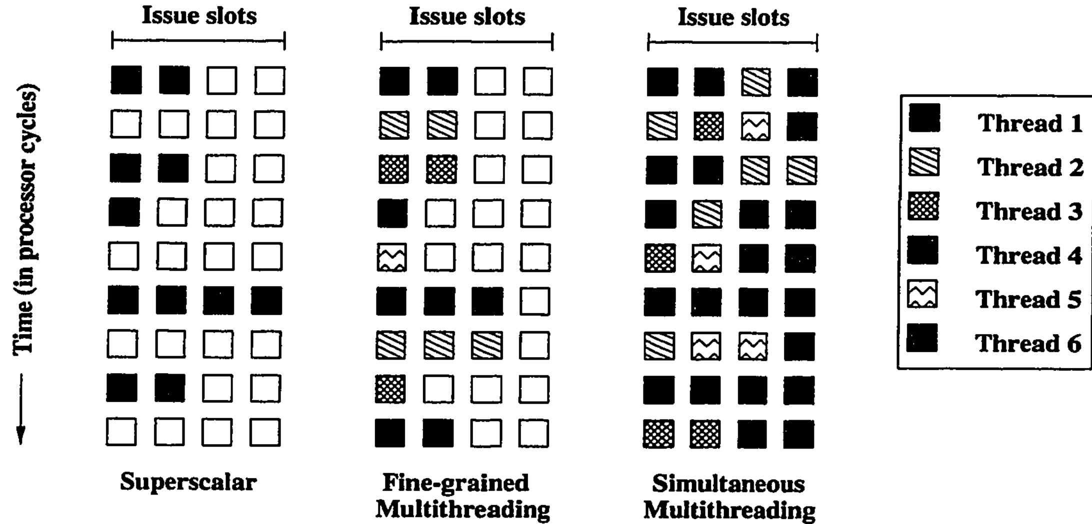

Figure 3.1: Comparison of issue slot partitioning in various architectures.

Figure 3.1 compares the issue-slot (functional unit) utilization in various architectures. Each square corresponds to an issue slot of the multiprocessor and the white squares are symbolic of the unutilized slots. Lack of utilization or ineffective utilization of available parallelism in a thread, leads to poor hardware-resource utilization. A superscalar processor relies on the amount of ILP in a thread, for its performance and thus, suffers from low hardware-resource utilization due to low or in-exploitable ILP in its single thread. Multiple-instruction-issue has the potential to increase performance, but is ultimately limited by instruction dependences (i.e., available parallelism) and long-latency operations within the single executing thread. The effects of these are shown as horizontal- and vertical-waste in Figure 3.2.

Empty issue slots discussed above can be classified as either vertical- or horizontal-waste. Vertical-waste comes as result of no instruction issue in a cycle. Horizontal-waste is introduced when not all issue slots in a cycle, can be filled. Superscalar execution introduces horizontal-waste and increases the amount of vertical-waste. Traditional multithreading starts losing its ability to utilize processor resources with increase in instruction issue-width. SMT1 attacks both horizontal- and vertical-wastes.

Classical multithreaded multiprocessors exploit TLP for performance but we observe that TLP is low in serial portions of programs due to inadequate compiler technology and expensive inter-thread synchronization. SMT1 allows multiple threads to compete for all resources in the same cycle, and avoids hardware-resource partitioning among different threads of execution. SMT1 can cope with varying levels of ILP and TLP; consequently, utilization is higher and performance is better.

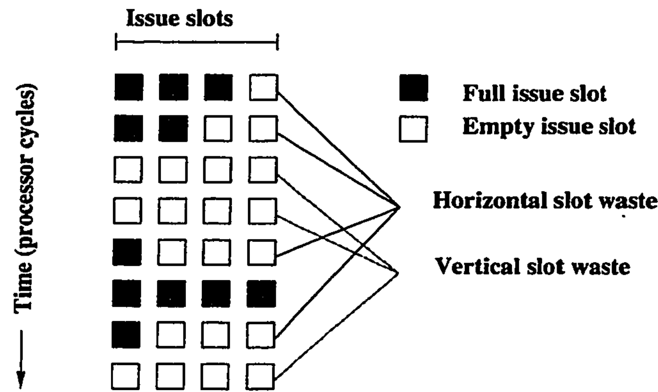

Figure 3.2: Empty issue slots as vertical or horizontal wastes

Usually, issue-slot waste is introduced by wrong-path instructions (branching) and optimistically issued instructions. In case of cache misses or data conflicts, all optimistically issued instructions need to be squashed to save issue-slots from being wasted. In SMT1, right-path instructions are intermingled in the instruction queues with wrong-path instructions. Thus, SMT1 is more tolerant of branch mispredictions, because it is less dependent on techniques that expose single-thread parallelism (for example, speculative fetching and speculative execution based on branch prediction) due to its ability to exploit inter-thread parallelism. SMT1 differs from SMT2 primarily in that it is multithreading of lesser degree.

3.2.2 Simultaneous multithreading with lightweight threads (SMT2)

We now, turn our discussion to simultaneous multithreading with lightweight threads, i.e., essentially using simultaneous multithreading to refine Tera's fine-grained multi-threaded architecture (MTA). Our goals mainly are to achieve both:

(*i*) fine-grained multithreading in order to sustain the concurrent processor operations every clock cycle as required by Little's law; and

(*ii*) fine-grained synchronization in order to permit frequent inter-thread communications.

We start with a brief description of Tera's MTA which is the soul of SMT2. Then we go on to show the advantages that SMT2 has over SMT1 and over superscalar processors and other multithreaded architectures. We also describe the qualitative comparisons between SMT1 and SMT2.

3.2.2.1 Architectural overview of SMT2

The Tera MTA implements a physically shared-memory multiprocessor with multi-stream processors and interleaved memory units interconnected by a packet-switched network [Tera]. The MTA does not use caches for latency avoidance, instead it relies completely on latency tolerance provided by multithreading, for all memory references. Each processor of Tera supports 16 protection domains and 128 streams. A protection domain implements a memory map with an independent set of registers maintaining stream resource limits and accounting information. Hence, 16 distinct applications maybe run in parallel on a processor. The 128 active contexts, or threads (or streams) supported simultaneously, enable concurrent issue of instructions. Each stream possesses its own register set and is hardware scheduled. Threads are switched every clock cycle, irrespective of whether they block. The hardware provision of 128 active threads per processor is more than necessary to keep the processor busy at any moment of time. The extra threads enable a processor to remain saturated during periods of higher latency such as synchronization waits and memory contention.

- Stream-state and Branch target registers
  Each stream has a Stream State consisting of the following three:

a) 1 64−bit Stream Status Word (SSW),

b) 32 64−bit General Registers ($R0 - R31$),

c) 8 64−bit Target Registers ($T0 - T7$).

Context switching is rapid (every cycle) and there are 128 $SSW's$, 4096 general registers, and 1024 target registers. Tera replicates all processor-states 128 times and relies completely on latency tolerance rather than latency avoidance. The large number of threads brings in many advantages. First, the fact that some instructions might not have much lookahead and can always be replaced by instructions from other threads. Second, some memory references might face contention in the network, resulting in long memory latencies. This event can be well tolerated by bringing in instructions from many other available threads. Third, the synchronization waits, which can be long enough due to load imbalances and resource-contention, can be tolerated if there are plenty of threads available.

The program addresses are 32 bits long. There are multiple program counters and each stream has its program counter stored in the lower half of its SSW. The target registers are used as branch targets. Determination of branch target addresses is independent of the decision to branch, which proves beneficial as the hardware prefetches instructions at the branch targets, thus avoiding delay when branch decisions are made. Using target registers also makes branch operations smaller, resulting in tighter loops. Skip operations enable to alleviate the need to set targets for short forward branches. The target register $T0$ always points to the trap handler and this makes trap handling extremely lightweight and independent of the operating system.

- Horizontal Instructions

  A horizontal instruction consists of several operations specified together within a single instruction. Tera instructions are mildly horizontal and a typical instruction consists of three operations: a memory reference-, an arithmetic- and a control-operation (MAC). Such horizontal instructions enable processing of vectorizable loops, at a nominal vector rate of one flop per cycle.

- Explicit-Dependence Lookahead

Memory-operation dependences in Tera's MTA are handled, using the technique called explicit-dependence lookahead (EDL)—wherein each instruction contains a three-bit lookahead field that explicitly specifies the number of immediately following instructions within that thread which are independent of that particular memory operation. Every instruction is tagged with the *three*—bit lookahead field, by the compiler. With the three-bit lookahead field, seven is the maximum possible number of instructions that can be explicitly declared as being independent of a particular memory instruction (i.e., simultaneously outstanding from a given thread). This implies that at most eight instructions and twenty-four operations (Tera issues three operations per cycle) can be concurrently in execution from each thread. SMT2 has a very high instruction issue-width and better fine-grained multithreading due to EDL and can actually hope to achieve the large concurrency required by Little's law for gigaflops computing performance.

- Efficient Synchronization

  Fine-grained synchronization is essential to enable frequent inter-thread communication. SMT2 or Tera uses a *full/empty* bit with each memory word to achieve synchronization. When a memory access for synchronization is made and the full/empty bit is in the empty state, the request returns to the processor with a failed status. The hardware keeps the request in a retry queue and keeps retrying for a sufficient number of times. These retry requests are interleaved with new memory requests, in order to avoid congestion on the network. While the hardware retries memory synchronization operations, the processor continues to issue instructions in the normal fashion. The retry count is maintained with each protection domain and is incremented with each retry or failed status of its threads. On reaching, a retry-count limit, the stream traps using the lightweight trap handling facility. The value of the retry-count is set in a way, to effectively balance the cost of saving and restoring the state of the blocked or swapped thread.

3.2.3 Qualitative and Quantitative Comparisons between SMT1 and SMT2

1. Zero-cost context switch time

   The major advantage of SMT2 is that it provides zero-cost context switch time. The large number of available threads and rapid context switching every clock-cycle enables cheap context switching between processor-resident threads. The two primary causes for the zero-cost context switching are that:

   - The state of the switched-out thread need not be saved or restored (later) due to the fact that, a thread does not lose its state when it is switched out. Each thread has a program counter, which keeps track of the last issued instruction from the thread, while the multiple registers belonging to each thread keep track of the rest of the thread-state. Moreover, the thread-state of a switched-out thread needs no rebuilding upon its return to the ready-active state, as it does not lose its state after the context-switch.
   - Threads do not become inactive quite frequently. In cases when a thread fails to get memory access for a synchronization operation, the hardware retry mechanism retries the request a sufficient number of times, before switching the thread out of the processor. Such switching out of a processor-resident thread to memory, and switching in a memory-resident thread to the processor is what we call a context-swap. Context-swaps are few in SMT2, and when they do occur, they are not expensive as compared to the context-swaps in superscalar processors and SMT1.

   In SMT1, the context-switch cost is high because each context-switch is actually a context-swap. Every time that a thread is switched out, its state has to be saved; and every time that a thread is switched in, its state has to be restored. In order for a switched-out thread in SMT1, to regain most of its former speed, its thread-state has to be rebuilt. Such rebuilding of thread-state is expensive and relates to the nonzero context-switch cost of the superscalars and SMT1 processors.

2. Effectively small processor-state of SMT1 as compared to SMT2

Zero-cost context switches are the main architectural precondition for fine-grained synchronization. We need a theory that relates context-switch cost to the architectural features. Our primary notion is that of an *effective processor state*, which we propose to be a weighted sum of the architectural and non-architectural state. The architectural state as mentioned before, includes the processor registers, the program counter, the stack-pointer and some per-process parts of the processor status words. The nonarchitectural state of the system includes the cache footprint, the reorder buffer and the branch translation buffer (BTB), which if abandoned (or not saved) incurs enormous performance consequences. In traditional superscalar processors or in SMT1, there is a very simple relation between processor-state and context switch time; namely, twice as much state implies twice as much context-switch time. In multithreaded processors, the relation between state (and space) and time is not quite as direct. In fact, we need to articulate a notion of an effective processor-state. We have done this in two stages. First, we proposed an effective context-switch time for superscalar or SMT1 processors. Now, we propose a corresponding (suggestive) notion of effective processor-state. Our goal here is to be as quantitative as possible.

We suggest here a the notion of effective processor-state that captures the effective context-switch time in a semi-quantitative manner. We propose to measure effective processor-state $psmt_1$ of a superscalar or SMT1 processor as,

$$psmt_1 \; = a + n/k,$$

where a is the architectural state and n the non-architectural state. What is a good value for k? One line of reasoning suggests that k should be equal to 2. The assumption here is that if 50% of the non-architectural state is recovered, the performance will be an appreciable fraction of the pre-switch performance. If more non-architectural state recovery is required to regain acceptable performance, we would consider decreasing k.

In superscalars, when a thread encounters a nonzero-latency operation, it is switched out, and another ready thread is switched in; the processor saves or restores the architectural state a of the switched-out thread. When the switched-out thread returns after the memory latency has been resolved, it

must rebuild its nonarchitectural state to re-attain its former speed. The value $n/2$ indicates the amount of nonarchitectural state that a thread has to rebuild in order to regain at least half of its former speed. The above formula, though merely suggestive, clearly has some strong correspondence to context-switch time. In traditional superscalar multiprocessors, the architectural and non-architectural state size has grown massively over the years. This implies that the effective processor state $psmt_1$ has been growing enormously.

MTA has less effective processor-state consisting of only its architectural-state a' as compared to the traditional non-multithreaded processors. MTA does not have any nonarchitectural state because it does not support caches, reorder buffers, branch history buffers (BHB), and branch translation buffer (BTB). The need for nonarchitectural-state does not arise due to its reliance on high concurrency, rather than spatial locality. As we have seen earlier, in superscalars the size of register files increases to squeeze out the last drop of ILP—the higher the issue-rate of instructions, the larger grow the reorder buffer and the branch translation buffer. This massive growth of non-architectural state can be avoided by SMT2, wherein concurrency brings in performance. The effective processor state-size for SMT2 can be quantified as,

$$psmt_2 = a' << a$$

This smaller effective state size $psmt_2$ enables an essentially zero-cost context switch. It is obvious that $psmt_1 > psmt_2$ and hence, it is emphatically more beneficial to integrate Tera's MTA with simultaneous multithreading (SMT2) which can generate zero-cost, frequent context switching in a high performance supercomputer.

Hence, we conclude three points of vital importance from the above discussion.

- First, we observe that SMT2 provides a zero-cost context switching when compared to SMT1, because the state of a processor-resident thread need not be saved or restored, during or after a context-switch, respectively. Moreover, there is no rebuilding of thread-state to be done when a thread returns to the processor from the memory, as it carries its own state.

- Second, we observe that the effective processor-state in SMT2 consists of only the actual architectural state a', and no nonarchitectural state.
- Third, we notice that the large context-switch cost in case of SMT1 with superscalar processors (or heavyweight threads) is mainly due to their massive non-architectural state n. The effective state $psmt_1 = a + n/2$ gives SMT1 machines a high context-switch cost.

Chapter 4

Shared Responsibility for Parallelism

"The real problem is not whether machines think—but whether men do."

— B. F. Skinner

This chapter explores the programmer's role in massively concurrent parallel programming, and proposes a reasonable division of labor between the computer system (i.e., the hardware architecture, the system software including a parallelizing compiler), and the programmer. Our recommendation is that the computer system should be responsible for discovering, extracting and exploiting 85% of all the parallelism in a computation, while the programmer should be responsible for the remaining 15%. Throughout this chapter, we assume a shared-memory programming model, which should be viewed as a performance model. There are two performance models, viz., *uniform memory architecture (UMA)* and *non-uniform memory architecture (NUMA)*. As will be seen, we favor the uniform-memory model and see great promise in massively concurrent parallel programming of uniform shared-memory multiprocessors.

In the earliest shared-memory architectures, the processors were connected to a single shared memory by a shared bus. There was a uniform access time from each

processor to the single main memory. Data decomposition was irrelevant because all data was stored in a single shared memory. This type of centralized shared-memory architecture was a popular organization. Later, a type of multiprocessor called SMP achieved essentially this shared-memory performance model using interconnections that were not buses. However, not all SMPs were as uniform as they should have been when they grew in scale. Bus-based multiprocessors do not scale. For this reason, all large multiprocessors have physically-distributed shared memory. The interesting question for any machine with physically-distributed shared memory is: what is the degree of nonuniformity in their shared memory performance model. Of course, the physically separate memories are addressed as one logically shared address space, in the sense that a memory reference can be made by any processor to any memory location. When the logical non uniformity is noticeable it is called a *distributed shared-memory* machine *(DSM)*. Such machines are also called *non-uniform memory architectures* (NUMA)s. In these machines the access time depends on the location of a data word in memory.

The significance of UMA as compared to NUMA shared-memory performance model lies in the effects on control and data decomposition in parallel programming. In UMA machines, data and control decomposition are decoupled because data decomposition does not really affect performance. On the other hand, in NUMA machines, especially as the degree of non-uniformity increases, the problems of data and control decomposition become tightly coupled, in fact. This makes parallel programming hard. Thus, we primarily address the uniform shared-memory architecture wherein we discuss the shared responsibilities of the compiler and the programmer for control decomposition in the program.

Before proceeding, it might be useful to recall some of the major taxonomies for parallel programming. One distinguishes *state-based* from *functional programming.* Another distinguishes *explicit* from *implicit parallel programming.* In implicit parallel programming the programmer writes a Fortran or C program without any parallel constructs, and a parallelizing compiler breaks apart loops and loop nests into threads that run in parallel. The programmer may provide clues to the compiler, either about control decomposition or data decomposition, which can be very useful to compiler decisions. In explicit parallel programming, the programmer does some of this work upfront using parallel constructs. As stated above, if a shared-memory multiprocessor has a non-uniform memory, then both the programmer and the compiler are

burdened with the tasks of both data and control decomposition. Such tasks tend to become more tightly coupled as non-uniformity increases. On the other hand, if the shared-memory multiprocessor has a uniform memory, then the problem of data decomposition largely goes away, and the programmer and compiler may concentrate on control decomposition without being troubled for data decomposition. There is an enormous difference when neither the programmer nor the compiler needs to worry about data decomposition. The distinction between explicit and implicit parallel programming is sharp, perhaps and reduces to helping with loops (implicit) as well as showing control flow explicitly (explicit).

Of course, we may have parallel programming methodologies that are partly explicit and partly implicit. For example, we might use a single assignment variables at coarse granularity but leave all loop processing to the compiler. Indeed, this is quite close to our proposed work-load sharing between the compiler and programmer as 85% and 15% respectively. This is a goal. These numbers represent intuitions of designers of parallel computers with vast experience. We may have to change any number of things to achieve our goals, including our programming languages.

4.1 Explicit Programmer Control Decomposition

4.1.1 The CC++ Programming Model (Explicit Control Constructs)

A good example of an explicitly parallel programming language is *Compositional C++ (CC++)*. It is a distinct programming language that contains C++ and an added set of explicitly parallel constructs. The programmer may explicitly parallelize code through the use of these special parallel constructs. First, we discuss the three constructs for *parallel composition* that are available in CC++ and then we go on to discuss the synchronization constructs [CC++].

(i) The *par* Block

This is the most elementary way of specifying parallel composition in CC++. The

syntax of a **par** block is that of the compound statement in C or C++ with the keyword **par** preceding the block. A **par** block can lexically contain any CC++ statement with the exception of variable declarations and statements that result in nonlocal changes in the flow of control. The statements in the **par** block can contain simple statements, sequential blocks or, even nested **par** blocks. A simple example of a **par** block is given below.

```
{
   int a, b, d;
   c = 1;
   par {
       a = 2;
       b = c + 3;
   }
   d = b + 1;
}
```

An example, of a nested **par** block follows.

```
par {
   par {
       S1;
       S2;
   }
   {
       S3;
       par {
           S4;
           S5;
       }
       S6;
   }
   S7;
}
```

The semantics in both the examples is what you would expect in a sequential program. A new thread of control is created for each top level statement in a **par**

block. A **par** block terminates after all its statements have terminated. As defined by fair, interleaved execution of the top level statements in the block, every executable statement in a **par** block will eventually execute, even though the block might not terminate.

A **par** is useful when there is a need to create a fixed number of threads and that number is known at compile time. Recursion within a **par** block can be used to create an arbitray number of concurrent threads, although iteration using the **parfor** statement is often a better way of expressing such computations.

(ii) The *parfor* Statement

This is a parallel loop construct in which the iterations are executed in parallel with each other. The body of each iteration is executed sequentially and there is an implicit barrier at the end of a **parfor**, in that it completes only after all its iterations have completed. The syntax of a **parfor** statement is exactly like the **for statement** in C++. The loop body can be a simple statement, a sequential block, or a parblock. A easy example of a **parfor** could be:

```
{
    int A[N]; B[N]; C[N];
    parfor (int index = 0; index > N; index++ )
      {
         A(index);
         B(index);
         C(index);
      }
}
```

Each iteration of a **parfor** creates a new thread which executes in parallel with all other iteration bodies. The threads have the same interleaved execution semantics as the **par** block. The **parfor** terminates when all the loop bodies have terminated. Each thread has a local copy of the value of the index variables at the time that the thread was created. In the above example, each loop body will have a local variable called *index* with a value set to the value of *index* in the **parfor** loop at the time of creation of the thread for the loop body.

In the above examples of parallel composition, there was only one mechanism of synchronization between concurrently executing threads of control viz., the implicit barrier at the ends of **par** blocks and **parfor** statements. CC++ has a powerful complementary mechanism for explicit parallel programming. This new mechanism expresses arbitrary synchronizations between concurrent threads of control, which we discuss next.

(iii) Single-assignment Variables

CC++ introduces a new type of variable — a single-assignment variable denoted by the keyword **sync**. We show below how single-assignment variables can simplify parallel programming. The single-assignement variables are very much like the *future* variable in MultiLisp. A **sync** variable in CC++ is treated *exactly* like a **const** variable in plain CC++ with two exceptions:

- Unlike a constant **const**, the initial value of a **sync** variable need not be defined when it is declared. Initialization can be delayed. Nevertheless, once defined, there is no difference between a single-assignment variable and a constant. The value of a defined single-assignment variable cannot be modified. Attempting to modify the value of a defined single-assignment variable is a run-time error.
- Any process that attempts to read an undefined **sync** variable suspends, until such a time when the variable becomes defined.

Thus, single-assignment variables like **sync** provide a cheap means for synchronization because they support the following rule: *if a thread of control attempts to read a single-assignment variable that has not yet been defined, that thread suspends execution until that single-assignment variable has been defined.* This allows the threads of control that share access to a single-assignment variable to use that variable as a synchronization element.

We mention some examples of how **sync** variables can be declared.

```
sync int a;
sync int b[10];
sync char *p;
char *sync q;
sync UserDefinedClass u;
```

We, now provide a simple example of data dependency and flow control using **sync**.

```
{
    sync int a,b,c,d;
    par {
        a = b+c;        // the first  thread of control
        c= 2;           // the second thread of control
        b= 3;           // the third  thread of control
        d= a+b;         // the fourth thread of control
    }                   // here,  a=5,  b=3,  c=2,  d=8
}
```

The data dependencies in the above example dictate the flow of control. The first thread of execution, that defines the **sync** variable a will not proceed until both b and c have been defined. The second and third threads of control will proceed immediately after b and c have been defined. The fourth thread defines d after a and c have been defined. Another interesting example could be the problem of calculating all the powers of 2 from 0 to N–1. We know the fact that the ith power of 2 can be calculated as $2 * 2^{i-1}$.

```
{
sync int P[N];
P[0] = 1;
parfor (int i = 1; i < N; i++)
    P[i] = 2 * P[i-1];
}
```

In the above example, the order in which the threads of execution created by the **parfor** statement, are interleaved or executed, is uncertain. However, the **sync** variable *P[N]* ensures that a value will not be assigned to *P[i]* until a value has been assigned to *P[i-1]*.

Single-assignment variables can also be used as function arguments. The *pass-by-value* semantics of function invocation in C and C++ guarantees that the single-assignment variable can be copied (and hence has been defined) before the function begins execution. For example:

```
void SomeFunction(sync int i)
                        //Suspends here until i is defined
{
                       // At this point, we can
                // assert that i has been defined
    ...
}
```

Similarly, a function can return a single-assignment type. The single-assignment nature of a **sync** variable cannot be cast away, either implicitly or explicitly. This guarantees that a **sync** variable cannot be misused (i.e., modified) in a thread of control. For example, we consider the following:

```
{
void f(int *);
sync int a;
int b;
 b = a;          // This is correct and O.K
 b = (int)a;     // This is also correct and O.K
(int)a = 3;      // This is an ERROR !
f((int *)&a);    // This is also an ERROR !
}
```

The concept of **sync** variables can also be used in higher levels of the memory hierarchy. Thus, we see that in a continuum of sharing possibilities, depending on the degree of use of explicit parallel constructs, what remains unchanged is the fact that managing the fine granularity is always left to the compiler.

4.1.2 The OpenMP Programming Model (Control Constructs as Compiler Directives)

The OpenMP [OpenMP] is a proposed industry-standard application-program interface (API) for explicit parallel programming of shared-memory multiprocessors. OpenMP provides moderate scalability of applications performance to programs with

moderate amounts of interthread synchronization and communication that run on true SMPs. The OpenMP API uses a fork-join model of parallel execution which is somewhat simpler than the CC++ model. When a parallel construct is encountered, the master thread creates a *team* of threads, and the master thread becomes the master of the team. Upon completion of the parallel construct, the threads in the team synchronize, and only the master thread continues execution.

In the OpenMP Fortran Application Program Interface, the directives are special Fortran comments that are identified with a unique *sentinel*. The directive sentinels are structured so that the directives are treated as Fortran comments and not as executable statements. Let us consider a few interesting examples from the OpenMP Fortran API.

(i) The parallel region construct

The **PARALLEL** and **END PARALLEL** directives define a *parallel region*—a parallel region being a block of code that has to be executed by multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts parallel execution. It is basically the same as **par** blocks in CC++. The **PARALLEL** directive can be used in coarse-grained parallel programs. When a thread in execution encounters a parallel region, it creates a team of threads, and itself becomes the master of the team with its thread number as 0, within the team.

The parallel block denotes a structured block of Fortran statements. Attempting to branch out of the block is an error, as it is in CC++. The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads. The **END PARALLEL** directive denotes the end of the parallel region and there is an implied barrier at this point. The only thread that continues execution at the end of the parallel region is the master thread. The parallel regions can also be nested within one another. We, thus find that the **PARALLEL** and **END PARALLEL** directives in OpenMP Fortran API are almost equivalent to the **par** block in CC++.

In the following example, each thread in the parallel region decides what part of the global array X to work on based on the thread number:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X, NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
```

```
        NP = OMP_GET_NUM_THREADS()
        IPOINTS = NPOINTS/NP
        CALL SUBDOMAIN(X, IAM, IPOINTS)
!$OMP END PARALLEL
```

(ii) The DO directive

The **DO** directive specifies that the iterations of the immediately following **DO** loop must be executed in parallel. The iterations of the **DO** loop are distributed across the already existing threads.

```
!$OMP DO
        DO I = 1, N
             A[I] = 0
             ...
        END DO  // the NOWAIT clause can be included here
```

If the *NOWAIT* clause is not specified with the *END DO* directive, the threads do not synchronize at the end of the parallel loop. Threads that finish early, proceed straight to the instructions following the loop without waiting for the other members of the team to finish the *DO* directive.

(iii) The SECTIONS directive

This is a non-iterative work-sharing construct that specifies that the enclosed sections of code are to be divided among threads in the team and that each section be executed once by a thread in the team. The code enclosed in a **SECTIONS/END SECTIONS** directive pair must be a structured block. In addition, each constituent section must also be a structured block. It is illegal to branch into or out of the constituent section blocks. It is similar to the **par** block construct in CC++.

```
!$OMP SECTIONS [clause,...]
        [!$OMP SECTION]

             block
```

```
[!$OMP SECTION]

      block

   ...
!$OMP END SECTIONS [NOWAIT]
```

(iv) The BARRIER directive

The **BARRIER** directive synchronizes all the threads in a team. When encountered, each thread waits until all of the others in that team have reached this point. Such a barrier is implicit in CC++ wherein, all threads of execution within a loop wait until a time when all have finished execution. In the following example, the call from *MAIN* to *SUB2* is legal because the *BARRIER* (in *SUB3*) binds to the *PARALLEL* region in *SUB2*. Similarly, the call from *MAIN* to *SUB1* is legal.

```
      PROGRAM MAIN
      CALL SUB1(2)
      CALL SUB2(2)
      END

      SUBROUTINE SUB1(N)
!$OMP PARALLEL PRIVATE(I) SHARED(N)
!$OMP DO
      DO I = 1, N
      CALL SUB2(I)
      END DO
      END PARALLEL
      END

      SUBROUTINE SUB2(K)
!$OMP PARALLEL SHARED(K)
      CALL SUB3(K)
      END PARALLEL
      END
```

```
      SUBROUTINE SUB3(N)
      CALL WORK(N)
!$OMP BARRIER
      CALL WORK(N)
      END
```

(v) The ATOMIC directive

The **ATOMIC** directive avoids race conditions by protecting all simultaneous updates of a location, by multiple threads. It ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The **ATOMIC** directive in OpenMP has an equivalent **atomic** construct in CC++.

```
!$OMP PARALLEL DO
      DO I = 1, N
           CALL WORK(XLOCAL, YLOCAL)
!$OMP ATOMIC
       X(INDEX(I)) = X(INDEX(I)) + XLOCAL
      END DO
```

(vi) The FLUSH directive

This directive identifies synchronization points at which the implementation is required to provide a consistent view of memory. Thread-visible variables are written back to memory at the point at which this directive appears. **FLUSH** does not have any matching or equivalent directives in CC++.

```
      ...
!$OMP FLUSH(ISYNC)
      DO WHILE (ISYNC(NEIGH). EQ. 0)
                          // C WAIT UNTIL NEIGHBOR IS DONE
!$OMP FLUSH(ISYNC)
      END DO
      ...
```

4.1.3 Tera Loop Directives

We have already seen how the parallel constructs and directives when added to a program can influence its translation by the compiler in CC++ and the OpenMP programming model. Now, we discuss the Tera loop directives many of which explicitly guide compilation by a parallelizing compiler without altering the semantics of the program. Tera directives are grouped into five general categories viz.,

- *compilation directives,*
- *parallelization directives,*
- *semantic assertions,*
- *implementation hints,*
- *language extension directives.*

We describe each of them in brief below. The Fortran version of these directives is used in the following descriptions.

1. Compilation directives

A *compilation directive* is a command to compile a program in a particular way. These directives apply to whatever follows them textually, and they apply until the end of the file or until another directive of the same kind is encountered. Of the many different compilation directives, we just mention a few interesting loop control directives. The two directives shown below control the choices made by the compiler while parallelizing a particular loop. We also note that the compiler makes reasonable decisions on its own without any directives.

- **C$TERA block schedule**
  When this directive appears before a loop that the compiler parallelizes, each thread assigned to the execution of the loop, performs a contiguous subset of the total iterations. For example, if 100 itereations are performed by 20 threads, the first thread executes the first 5 iterations of the loop, the second thread executes the next 5 iterations, and so on.

- **C$TERA interleave schedule**

  Unlike the block schedule, this scheduling interleaves threads while distributing them their share of load. For example, if 100 iterations are to be done by 20 threads, the first thread executes iteration 1, iteration 21, iteration 41, etc. This scheduling leads to a better load balancing especially for triangular loops. For example:

```
C$TERA INTERLEAVE SCHEDULE
      DO I = 1, N
          SUM = 0.0
          DO J= 1, I
              SUM = SUM + X(I,J)*Y(J)
          END DO
          Z(I) = SUM
      END DO
```

2. Parallelization directives

These tell the compiler how to parallelize various sections of a program. We discuss briefly the various types of parallelizations that are allowed.

* *Single processor parallelism*—This form of parallelism, also known as *fray parallelism*, has low overhead. However, the code is parallelized in such a way that it can only take advantage of the 128 threads on the Tera processor on which the code is running.

⋆ *Multi processor parallelism*—This form of parallelism, also known as *crew parallelism*, has higher overhead than single-processor parallelism. The number of threads available is bounded by the size of the entire machine rather than the size of a single processor.

- **C$TERA parallel [on | off | default | single processor | multiprocessor]** — this directive enables/disables automatic generation of parallel code for a section of the program as well as choosing the form of parallelism to use viz., single processor or multiprocessor. The off flag turns off parallelism until it is

turned back on or until the end of file. By default, automatic generation of parallel code is enabled and single-procesor parallelism is used in Fortran.

- **C$TERA restructure [on |off |default]** — this directive enables/disables loop restructuring and loop transformations. By default, this directive is on in areas in which parallelization is being performed and off in areas in which parallelization is turned off.

3. Semantic assertions

These provide information to the compiler that could be proved true about the program even though that proof may be beyond the capabilities of the compiler. Asserting semantics often yields more effective compilation.

- **C$TERA assert parallel**—This directive can appear before a loop constuct and asserts that the separate iterations of the loop may be executed concurrently without synchronization. The use of this loop does not ensure that the loop is parallelized, but it does strongly suggest so. This directive affects the loop following it.

- **C$TERA assert no dependence** *variable-list*—The directive can appear before a loop construct and asserts that if a word of memory is accessed during execution of the loop through any load or store derived from a variable in *variable-list*, then the word is accessed from exactly one iteration of the loop. The word *nodep* may be used in place of *no dependence*. For example:

```
C$TERA ASSERT NO DEPENDENCE IA
       DO I = 1, N
          IA(I,1) = IA(I, INDEX(I))
       END DO
```

4. Implementation hints

These directives intend to provide guidance to the compiler for effective optimization about the expected behaviour of the program.

- **C$TERA expect [true|false]**—This directive can appear before a logical IF and specify the expected value of the associated predicate. This is also used for

branch prediction and choosing the best parallel implementation of a containing loop depending on sparse versus dense branching.

5. **Extension directives**

The language extension directives allow the programmer to specify special parallel semantics for variables and statements without compromising the portability of their programs. Some of them are:

- **C$TERA volatile** *variable-list*—This directive specifies that the variables in variable-list should be marked volatile-qualified, similar to the C volatile type declaration.
- **C$TERA future** *variable-list*—This directive specifies that the variables in variable-list should be marked future-qualified, similar to the **FUTURE** type qualifier in MultiLisp.
- **C$TERA sync** *variable-list*—This directive specifies that the variables in the variable-list should be marked sync-qualified as in the **SYNC** type qualifier in CC++.

4.2 Examples of Explicit and Implicit Parallelism

In Section 4.1, we discussed various flavours of explicit control decomposition by the programmer. Because this was so straightforward, we have actually demonstrated that occasionally parallel programming can be quite easy indeed. In this section, we continue to examine explicit and implicit parallelism. We explore our recommendation that the computer system should be responsible for discovering, extracting and exploiting 85% of the total parallelism in a single program or application, while the programmer should be responsible for the remaining 15%. An appropriate division of labour between the programmer and compiler will enhance exploitation of parallelism within programs given the relative "clerical" strengths of these two entities. In Section 4.2.1, we illustrate examples of explicit and implicit parallelism so that we can better understand both the cost issues and the ease-of-use issues associated with massively concurrent parallel programs. In Section 4.2.2, we provide various examples to show that implicit parallellism works especially best when utterly cheap threads—for

switching and synchronizing—are available because the load is distributed appropriately to entities that can bear it. Our compilers are actually good at extracting fine-grained parallelism and it is wise to have cheap threads that will free the compiler to exploit as much fine-grained parallelism as it can find—a lot actually!. The following kinds of parallelism demand fine-grained synchronization:

- scans (prefix computations) and reductions
- DAG parallelism within large basic blocks
- parallel, possibly conflicting memory updates
- parallelism in inner loops exceeding one processor's needs

Another powerful technique is to uncover large amounts of both instruction-level parallelism (ILP) and thread-level parallelism (TLP). The amount of parallelism that can be pulled from ILP $\times$ TLP is vastly greater than which can be extracted alone from ILP. ILP processors are certainly limited in generating parallelism. In the next section, we provide several examples of explicit as well as implicit parallel programming techniques.

4.2.1 Examples of exlicit and implicit parallelism in loops and vectors

To begin with we illustrate a simple example of computing the rank of each number in a given vector.

1. **Parallelizing loops in a simple counting sort example**

   Most people are familiar with "counting sort" and so we will not discuss the algorithm in detail, but instead we will just look at the sequential and parallel implementation. To start with, we first accumulate, in $vector[n]$, the number of keys with value n. The next step would be to compute a starting location, $start[n]$, for each possible key value n. Thus, all $0's$ would come first, starting at location 0, then all the $1's$, follwed by all the $2's$ and so forth. Finally, the rank of each key value would be computed and would lie between $start[n]$ and $start[n+1]-1$ inclusive. The algorithm could be expressed in C as follows:

```
for (i = 0; i < key_bound; i++)                  \\ Loop 1
    vector[i] = 0;

for (i = 0; i < nkeys; i++)                      \\ Loop 2
    vector[key[i]]++;

start[0] = 0;                                    \\ Loop 3
for (i = 1; i < key_bound; i++)
    start[i] = start[i - 1] + vector[i - 1];

for (i = 0; i < nkeys; i++)                      \\ Loop 4
    rank[i] = start[key[i]]++;
```

The above sequential code could be parallelized using a parallelizing compiler, as follows:

```
for (i = 0; i < key_bound; i++)                  \\ Loop 1
    vector[i] = 0;

for (i = 0; i < nkeys; i++)                      \\ Loop 2
    vector[key[i]] = vector[key[i]] + 1;

start[0] = 0;                                    \\ Loop 3
for (i = 1; i < key_bound; i++)
    start[i] = start[i - 1] + vector[i - 1];

start$ = (sync int *) start;
#pragma tera assert parallel
for (i = 0; i < nkeys; i++)                      \\ Loop 4
    rank[i] = start$[key[i]]++;
```

The single-assignment variable **sync** has already been explained in sufficient detail in Section 4.1.1. The compiler generates parallel code for each of the above loops and distributes each iteration in each loop to a thread chosen from a set of acquired threads. The compiler recognizes *Loop* 1 as trivially parallel and distributes iterations among threads on all P processors, so that computation progresses at the rate of P iterations per clock cycle. On a single processor, the iterations are distributed across multiple threads, with each thread executing a single instruction per iteration.

Loop 2 is not trivially parallel as there is a loop-carried dependence caused by the increment of *vector[key[i]]* which becomes unproblematic if we can perform atomic updates of the memory locations. The atomic update can be done by a simple *fetch-and-add instruction.*

Loop 3 is a *linear recurrence* which the compiler automatically recognizes and generates parallel code to solve it using the technique called *cyclic reduction.* We will discuss more about recurrences and cyclic reduction in examples 2 and 3, respectively.

In *Loop* 4, the programmer has to explicitly inform the compiler that the loop is parallel, using the *assert parallel* statement. Normally, the compiler tries to maintain the semantics of the original code. However, in this example, the programmer does not care about the order in which the different $0's$ or $1's$ appear. Thus, the clustering of the keys can be done in parallel. The variable *start$* is declared as a pointer to **sync** *int*, implying that each integer in the vector should be treated as a **sync** variable by the compiler.

2. **Parallelizing recurrences**

   A recurrence refers to a loop where values computed in one iteration are used in subsequent iterations. The subsequent use of a value computed in a previous iteration is called *loop-carried dependence* [Wolfe]. Such loop-carried dependences usually prevent parallelization. Some really aggressive compilers are able to generate parallel code for a special class of recurrences called *linear recurrence relations.* A linear recurrence relation is of the form $x_i = \sum_{k=1}^{m} a_{i,k}x_k + c_i$, where m is the order of recurrence, $a_{i,k}$ is called the recurrence coefficient, and c_i is a constant term. A *first-order linear recurrence* is simply of the form $x_i = a_ix_{i-1} + c_i$

which can be rewritten as $x_i = a_i(a_{i-1}x_{i-2} + c_{i-1}) + c_i$ making it independent of x_{i-1}. This process of substitution changes the order of arithmetic computation. For example, in the following loop:

```
DO I = 1, N
    X(I) = X(I-1) + Y
END DO
```

the computation of $X(I)$ depends upon the previous iteration value $X(I-1)$. This particular case is easy to parallelize if the code is rewritten without loop-carried dependences as follows:

```
DO I = 1, N
    X(I) = X(0) + I*Y
END DO
```

All iterations can be done in parallel in the above rewritten code. However, some other types of linear recurrences may not be parallelized as easily as shown in the above example. These more difficult loops can be rewritten using a technique called **cyclic reduction** [Callahan] and thereafter easily parallelized. In general, recurrences of the form

```
X(I) = X(I-K) * F(I) + G(I)
```

are identified by the compiler as linear recurrences and are parallelized if the following two conditions hold:

- The operators $*$ and $+$ are sufficiently like multiplication and addition (e.g., bitwise **and** and **or**).
- All loop-carried dependences are simple and cross a small constant number of iterations (e.g.,K iterations).

We now discuss in more detail a specific linear recurrence and its implementation by *cyclic reduction*. The loop

```
X(K) = X(K-1) + Y(K)
```

is a fairly common structure, which we may assimilate to perform *radix* or *bucket sort*. We will translate this into the following threads (described here in SPMD

Fortran).

```
        sync real MSG$ (log2T,T)
        me = 1
        ...
        hop = 1
        DO 100 step = 1, log2T
            IF (me + hop  ≤  T) MSG$ (step, me + hop) = sum
            IF (me − hop  ≥  T) sum = sum + MSG$ (step, me)
            hop = hop * 2
    100 continue
```

Here, *sum* is thread local and is initialized to $Y[me]$ for each thread. Its final value is $X[me]$, for the loop body $X(k) = X(k-1) + Y(k)$. *me* is a thread index very much like the PID used to index arrays of process control blocks, i.e., it is just the thread name.

The **cyclic reduction** algorithm was derived from the cascade sum method to solve a particular case of a general first-order recurrence. We provide below, a situation in which the first N elements of an array are to be added together in parallel. We will use the cyclic reduction algorithm to rewrite and parallelize the sequential code for the addition.

We assume, we have T threads that execute the parallel code. Each worker thread is permitted to accumulate a private partial sum for the iterations assigned to it. The algorithm computes the partial sums $Sum_i = N_0 \oplus N_1 \oplus ... \oplus N_i$, for $0 \leq i \leq N-1$ assuming one input per thread (or processor as the case may be). The final global sum is then accumulated from the partial sums computed by the worker threads. Initially, the vector value N_i is loaded into thread T_i. Each thread T_i has a local variable Sum_i which is the data to be transferred in every step of the computation. The maximum number of steps that the algorithm goes through is $\lceil log_2 T \rceil$ or $\lceil 3.32(log_{10} T) \rceil$. The cyclic reduction algorithm can be explained with the aid of *Figure* 4 shown below.

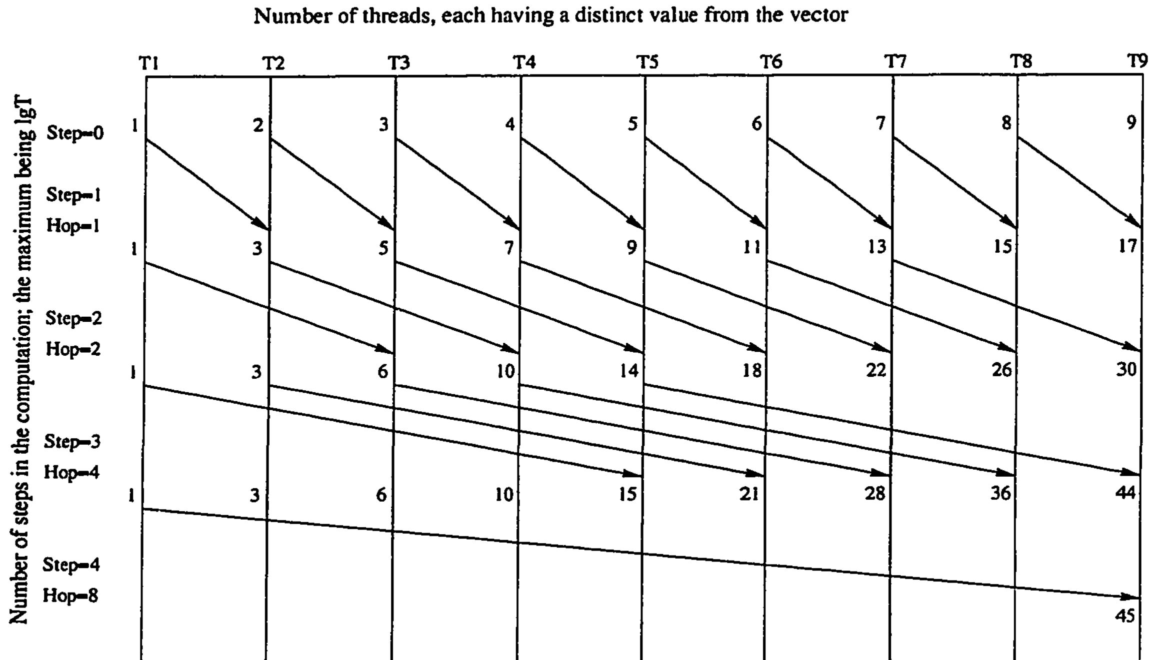

Figure 4.1: An example of *cyclic reduction* using 9 threads to compute the sum of the first 9 array elements starting from 1. The maximum number of steps is $\lceil\ log_2 9\ \rceil$.

Each thread or processor contains a value from the given vector. In this case, the goal is to compute the sum $1 + 2 + \cdots + 8 + 9$. The maximum number of *steps* in the computation will be $\lceil\ log_2 9\ \rceil = \lceil\ (3.32 * log_{10} 9)\ \rceil = \lceil\ 3.168\ \rceil = 4$. The threads execute in parallel and use the **sync** variable $MSG\$$ (,) to synchronize. This aids in synchronization of concurrent addition of the various local sum values. Use of **sync** variables has been discussed in Section 4.1.1.

Hop values increment in powers of two, starting from 2^0, running through 2^1, 2^2, and so on, up to 2^{n-1} where n is the value of the final n^{th} step or, $\lceil\ log_2 T\ \rceil$.

3. **Reductions**

Reduction operations involve reducing a large vector of scalar values to a single scalar value. As discussed in the above example, reductions in a concurrent loop can be efficiently done when the reduction is associative and commutative. An extremely common reduction is accumulating the sum of a vector or an array. For example,

```
SUM = 0.0
DO I = 1, N
   SUM = SUM + X(I)
END DO
```

computes a sum of the first N elements of the vector array X. Another example from linear algebra is the calculation of the dot product of two vectors:

```
SUM = 0.0
DO I = 1, N
   SUM = SUM + X(I)*Y(I)
END DO
```

In general, reductions are easier to solve in parallel than recurrences, and usually involve less overhead.

4.2.2 Parallelization versus vectorization

Parallelization and vectorization are closely related, though not identical. Parallel compilers are able to parallelize code that cannot be vectorized. This is because vector machines are able to take advantage of only inner-loop parallelism, but a parallel machine can take advantage of parallelism at any loop level.

Loops with output dependences are impossible to parallelize. For example,

```
DO I = 1, N
    Z(G(I)) = A * X(I) + Y(I)
END DO
```

Here, the parallelizing compiler is unaware of the values of $G(I)$ and hence is not sure that every iteration writes a different location in Z. Such loop-carried output dependences cannot be parallelized.

Since vectorization applies only to inner loops, outer loops can be parallelized in order to minimize parallel overhead. For example, the following can be parallelized:

```
DO I = 1, NROWS
    DO J = 1, NCOLS
    Z(I) = Z(I) + A(I, J) * X(J)
    END DO
END DO
```

The I loop can be easily parallelized but the J loop has a reduction. Thus, the parallelizing compiler parallelizes the outer loop and compiles the inner loops in a sequential fashion. There is a set of parallel startup overheads with the outer loop.

Whereas, the vectorizing compiler will interchange the two loops as follows:

```
DO I = 1, NCOLS
    DO J = 1, NROWS
        Z(I) = Z(I) + A(I, J) * X(J)
    END DO
END DO
```

The vectorizing compiler will vectorize the new inner loop while compiling the outer loop sequentially. Here, there are multiple vector startup overheads, one for each iteration of the J loop.

Our proposition is that, if threads that synchronize frequently are cheap, the overhead of parallelization would be much less. Parallelization of loops and vectors using cheap threads and implicit/explicit parallel programming techniques can be implemented easily and in a cost-effective manner using multithreaded hardware. The job of the parallelizing compiler becomes simpler and cost-effective with multiple inexpensive threads executing in parallel. The kind of architectural support that we have proposed in Chapter 3, best assists the compiler in exploiting parallelism for massively concurrent parallel programs. The compiler runs iterations of each parallel loop with multiple hardware threads. These threads communicate frequently among themselves at negligible synchronization cost. Our multithreaded hardware architecture supports cheap synchronization among threads, and also allows easy and cost-effective implicit and explicit parallel programming in order to exploit parallelism in loops and vectors.

4.3 Wait-free or lock-free transactions

Continuing with our proposition of shared responsibilities between the compiler and programmer, we examine and propose a method of implementing lock-free transactions for multiword shared variables, which will greatly facilitate synchronization and consistent updates by the compiler, without the use of conventional 'locks'. The primary motivation is to allow both the programmer and the compiler not to worry about deadlocks when it is difficult or inconvenient to define an ordering on a set of locks. Wait-free techniques are a standard solution to this problem. Ultimately, we will demonstrate an efficient dynamic hardware implementation.

The most common technique for updating shared variables is through the use of control variables called *semaphores* or *locks*. Many synchronization operations involve an atomic *Read/Modify/Write* operation of some form. Some of the methods implemented for synchronizing processes are:

- *Test-and-Set* which operates on a single bit,
- *Increment-Decrement* which produces sums and differences,
- *Compare-and-Swap* with high consensus number,
- *Fetch-and-Add* for fast atomic updates.

In this thesis we do not discuss each of them in detail. Instead we attempt to give an overall view of the methods mentioned above. *Lock* and *Unlock* statements have many drawbacks, including tending to serialize processors' requests. This leads to deadlocks and convoys for example, *Read/Modify/Write* operations are implemented with locks, where only one request of a set of multiple concurrent requests gets past the *Lock* to update the shared variable. When a process holding the *Lock* gets blocked for some reason, many useful cycles are simply wasted. *Spin-locks* [Anderson] were introduced in order to avoid processor cycles from being wasted. Spin-locks let processors do *busy wait*, but instead end up wasting processor cycles in an effort to repeatedly testing a semaphore, and cause memory contention at the semaphore. In fact, when multiple processes are waiting at a semaphore, the contention causes additional delay while a process is attempting to release a *Lock*. All the methods namely, test-and-set, fetch-and-add and increment-decrement use locks and critical sections

that lead to performance degradation. A single process either busy or blocked inside a critical section can keep multiple processes waiting at the entrance of the critical section.

The *Compare-and-Swap (CAS)* reduces locked regions of a program to a single compare-and-swap instruction [Herlihy, Moss 91]. The shared variable is locked at the start of the instruction, updated during the execution of the instruction, and unlocked at the end. In contrast to the other methods of synchronization, *CAS* does *not* create a critical section. It has been successfully implemented on the *IBM* 370 architecture, among others. A shared variable is read into a local register and a newly computed value of the shared variable is stored in another local register; the *CAS* statement compares the current value of the variable in memory to the old value stored in the register. If the values are equal, the instruction completes by writing the updated local copy back to the main memory. The *CAS* assures the atomicity of the Read/Modify/Write sequence because the shared variable is updated only after the old counter value is found to be the same as the current value, thus ensuring that the shared variable has not been updated by another process in the meantime. However, the Compare-and-Swap suffers from the *A-B-A* problem and does not completely serve the purpose. The A-B-A problem occurs when the original value of the shared variable is read as A, and while a process tries to evaluate a new value for the shared variable, it has already been changed from A to B and back to A by different processes. This can cause inconsistencies in results and thus Compare-and-Swap fails to provide consistent updates.

The more interesting problem with *CAS* is that most implementations are restricted to a single-word comparison. We suggest a *Multiword Compare-and-Swap (CASn)* for a multithreaded multiprocessor. The *Compare-and-Swap-Two (CAS2)* instruction has been successfully implemented in the M68000 architecture.

[Stone, Stone] discusses a viable solution to the problem using the multiple reservation approach which allows atomic updates of multiple shared variables, and simplifies concurrent nonblocking codes for managing queues and linked lists. The disadvantage of the proposed implementation of Stone's multiple reservation scheme is that it is based upon cache-coherence protocols.

We explain in stages our multiword atomic Read/Modify/Write operation, which are notorious for their bandwidth attenuation. In all cases, we assume a special machine instruction for atomic increment of indexed fields of multiword sync variables

with *full/empty bits* [Tera, SC97]. The stages are that we will first explain the use of this *Multiword Compare-and-Swap* operation where it is assumed that all addresses of variables in the transaction set, and their correspondence to fields in the multiword sync variable, are known at compile time. The algorithm holds for *commutative transactions* and alike. Later, we generalize this to allow dynamic runtime memory disambiguation leading to the same power. Again, the real motivation is to enable complex "database-like" programming without having to worry about deadlocks. We propose a hardware-managed technique which is faster and cheaper to implement. Our transaction algorithm is memory based and is for "lock-free" transactions with optimistic concurrency control. It does not depend on any invalidation-based cache-coherence protocol. The algorithm is provided below, where:

— the multiword sync variable $nc = <nc[a],\ nc[b],\ nc[c],\ nc[d]>$ indicates the total number of commits for each of the words that indexes a wraparound counter in nc,

— a, b, c, d are shared variables,

— D is the sector data cache,

— $old = <old[a],\ old[b],\ old[c],\ old[d]>$ and $new = <new[a],\ new[b],\ new[c],\ new[d]>$ are multiword thread local values.

A *sector cache* is a D_cache that allows wholesale (single-instruction) flushing of a set of tagged dirty values. The benefit is that if n dirty words need to be flushed, this can be done using a single machine instruction rather than n separate instructions. The *full-empty* bit is used for lightweight synchronization. There are two modes of interaction with the *full/empty bits*: (i) *Future variables*—read and write the memory word only when the bit is full. Leave the bit full; (ii) *Synchronized variables*—read only when the bit is full and set the bit to empty, and write only when the bit is empty and set the bit to full.

Circular counters (i.e., wrap around to 0 rather than overflow) are used for keeping count of the number of commits for each of the transaction variables. They are simply implemented as in any $Modulo - N$ counter. The value of N here is large enough to safely wrap around, for example $N = 2^{32}$.

The simplest way of explaining our transaction algorithm is to suppose that there are two commutative transactions executed by two threads with transaction sets a, b and b, c, respectively. *Thread* 1's transaction is on a and b, that is, its t_set is a, b, and its code reads as follows:

```
trans (a,b) =
loop
   start_trans ( )                                ; cache.color := next ( )

   [record number of commits for a,b]

   <wait/f, set/f: load (old := nc)>              ; atomic multiword read of nc
   loadc (a)                                      ; read t_set into D_cache
   loadc (b)

   [local computation]

   a' := f(a,b)                                   ; compute locally, store in  D_cache
   b' := g(a,b)

   [check if commit enabled]

   <wait/f, set/e: load (new := nc)>              ; atomic multiword RMW of nc

   t := < new[a], new[b],> = <old[a], old[b] >
   if t then
     tnc := < new[a]+1) mod N, (new[b]+1) mod N, new[c], new[d] >
     store (nc := tnc)
     sector_flush  ( )                            ; write back t_set from D_cache
   fi
   <set/f: nc>

   [test for successful commit]

   if t then
     exit                                         ; then exit loop
   fi
forever
```

Figure 4.2: Thread 1's transaction on variables a and b.

The first thread *Thread* 1 runs concurrently with another thread *Thread* 2 whose *t_set* is not disjoint. The transaction algorithm works on the other two shared variables b and c:

```
trans (b,c) =
loop
   start_trans ( )                              ; cache.color := next ( )

   [record number of commits for b,c ]

   <wait/f, set/f: load (old := nc)>            ; atomic multiword read of nc
   loadc (b)                                    ; read t_set into D_cache
   loadc (c)

   [local computation]

   a' := f(b,c)                                 ; compute locally, store in D_cache
   b' := g(b,c)

   [check if commit enabled]

   <wait/f, set/e: load (new := nc)>            ; atomic multiword RMW of nc

   t := < new[b], new[c],> = <old[b], old[c] >
   if t then
     tnc := < new[a], (new[b]+1) mod N, (new[c]+1) mod N, new[d] >
     store (nc := tnc)
     sector_flush ( )                           ; write back t_set from D_cache
   fi
   <set/f: nc>

   [test for successful commit]

   if t then
     exit                                       ; then exit loop
   fi
forever
```

Figure 4.3: Thread 2's transaction on variables b and c.

Let us discuss the action of the algorithm. This is optimistic concurrency control. If there is no interference to $T(a, b)$ by $T(b, c)$, there is no retry of $T(a, b)$. Only if $T(b, c)$ actually interferes with $T(a, b)$, is it necessary for $T(a, b)$ to retry the loop. We throw more light on the second phase of the exposition. The two critical issues are:

- dynamic runtime memory disambiguation, and
- runtime association of shared variables to counters.

The above algorithm as proposed by us admits an efficient hardware implementation. It is not difficult to bind nc to $a, b, ...$ at run time. We need an array of address registers. The underlying implementation resembles Stone's *Oklahoma Update* [Stone, Stone]. The goal here is to implement $CASn(a, b, ...)$, where the parameter addresses are known at run time.

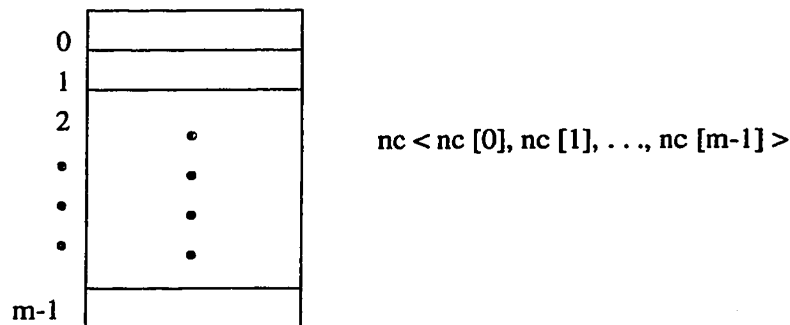

Figure 4.4: The simple address register AR[ ] required to increment the multiword sync variable nc.

The conclusion is that we do need to set aside dedicated multiword sync variables, and to implement the standard "synch ops" on them. However, the nc variables are few and known at compile time, and the atomic component increments are trivial from a hardware point of view. Thus, we have a very simple and efficient implementation of Compare-and-Swap2 [Herlihy, Moss 91] without any of the downsides of the proposed implementation.

Chapter 5

Conclusions and future work

"To stay ahead, you must have your next idea waiting in the wings."

— Rosabeth Moss Kanter

We want to repeat a point. We did not invent RISC superscalar processors and we did not invent multithreaded processors. What we did was to bring out an analytical point that has not been properly appreciated. Essentially, we showed that high-end computing requires massive concurrency, and tht this concurrency is available from multithreaded architectures, but not from RISC superscalar architectures. This is the central thesis of this thesis. The specific conclusions of this thesis are:

- Massive effective processor state is a consequence of single program counters, and makes context switching unreasonably expensive.
- Multithreaded processors both exploit ILP and TLP, and distribute processor state across multiple program counters making context switching extremely cheap.
- Programmers and compilers should cooperate, not compete—although the compiler must do the bulk of the parallelization work.
- We hope that transactions will be the new parallel-programming methodology and that *multiword compare-and-swap* will be generally useful in parallel programming.

Broadly speaking, this thesis has looked at some of the architectural trends that will affect how computer architecture will play out in the next fifty years. We have used conventional RISC superscalar design as the archetype of the current conventional wisdom, both because there are a lot of RISC and because alternatives such as Intel's Merced chip are closely held proprietary secrets. Taking an abstract view, however, we may argue generically about ILP chips (ones that focus exclusively on instruction-level parallelism) and ILP/TLP chips (ones that trade off dynamically between instruction-level and thread-level parallelism), which—for all practical purposes—means multithreading. Both conventional RISC superscalar designs and the Merced design (but perhaps *not* the Deschutes design) are ILP processors in that they are hardware unithreaded processors that use various techniques to expose and exploit the parallelism in individual threads (i.e., ILP).

As clock speeds increase, there is a steady upward pressure on processor state. Of course, the precise details depend on how much of the state is architectural (example, registers with fixed names) and how much is nonarchitectural (example, data caches and branch-target buffers). If you have any nonarchitectural state at all—even a simple data cache—it will necessarily grow as clock speeds scale up, primarily because you need to keep filling multiple functional units and multiple pipelines, and caches help avoid processor stalls. But, this means that context-switch times will scale up to match the growing processor state. We know that data and other caches build up state to enable high performance of threads. When this state is lost, performance suffers. The correct way to view context-switch time is not merely as the time to save and restore registers, but rather as the sum of the times to 1) save thread *t's* architectural state when *t* is suspended, 2) restore thread *t's* architectural state when *t* is resumed, and 3) rebuild thread *t's* nonarchitectural state sufficiently so that thread *t* regains the bulk of its previous performance.

This can only be called massive effective processor state. Normally, it makes context switching unreasonably expensive. But, there is a way out. To some extent, large processor state is inevitable. But there is a fundamental alternative:

- do you load the whole processor state onto a single program counter? If so, you can afford to switch but rarely.
- do you distribute the processor state across multiple program counters? If so, you can easily implement low-cost context switching, and switch whenever you

want.

Thus, there is almost a binary choice between pure-ILP processors which do not distribute state and therefore cannot switch, and ILP/TLP processors which do distribute state and therefore can easily switch. The latter, but not the former, provide appropriate architectural support for massive concurrency. This is the thesis of this thesis. The rest is details. Now, we explain some of these ideas.

The focus of this thesis is to show clearly how, as conventional RISC superscalar processors become increasingly aggressive about extracting ILP from threads, they become increasingly unable to benefit from TLP. Since ILP is limited, so is the sustained performance of multiprocessors built from conventional processors.

In this thesis we have spent considerable time analysing the consequences of single-program-counter design, and show that as one attempts to extract more ILP, threads become heavyweight, making it impossible to benefit from TLP. Although we do not discuss Merced, which uses an alternative—viz., predication—to speculative branch prediction to get more ILP, and which moves out-of-order control from the hardware to the compiler, there are reasons to believe that, because it looks to ILP alone for parallelism, it too has no fundamental solution to providing massive concurrency of processor operations. We leave to future work the demonstration that Merced is intrinsically unable to profit from TLP, albeit possibly for very different reasons than the ones that make RISC processors unscalable. Both probably are unable to tolerate memory-reference latency for much the same reasons.

There are still multiple open problems related to this work:

- To what extent does our criticism of RISC superscalars carry over to Merced? to any processor with an exclusive focus on ILP? What governs thread weight in general? What governs latency tolerance in general? What governs context switching in general?

- Can complex hardware multithreaded systems be radically simplified? What happens to the network as the number of processors grows? What happens to the cost of the network? its latency? its bandwidth? Could a network with a high processor count still be relatively immune to "hotspots" (i.e., memory and network contention)? Is high bandwidth all we can provide in the hardware to reduce memory contention? What memory management is appropriate for

parallel computers? Do we want hardware address translation without faults? How should locality of various kinds be integrated with multithreading? Is there an absolute distinction between control and data locality? Can dataflow ideas obviate this distinction? How should latency avoidance of various kinds be integrated with multithreading? How should memory-bank interleaving be set up in large scale parallel architectures? Is it scalable?

- Can our wait-free transaction solution be generalized? What does this imply for cache architectures? How can it be turned into a compiler algorithm that parallelizes loops? In the context of fine-grained synchronization, is ours the first cost-effective wait-free primitive implemented (on paper, anyway)? How can it be tried out on other difficult programming problems? Can one write a simulator? What is the right mix between explicit (programmer-directed) and implicit (compiler-directed) transactional programming?

- Finally, is it really true that practically no reprogramming is required to port codes from parallel vector processors or mid-range SMPs to a flat-memory, cacheless, multithreaded machine?

This list reads more like a list of open problems in parallel computing than it does a set of minimal extensions to this thesis. However, in computer architecture, think about one thing and you soon are thinking about a thousand things. Equivalently, the ideas explained in this thesis point the way to fresh thinking about many aspects of computer architecture. It is a modest contribution to the reinvention of computer architecture.

Bibliography

[Anderson] T.E. Anderson. "The Performance of Spin-Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 1, pp. 6-16, Jan. 1990.

[Calder et al.] Brad Calder, Dirk Grunwald, and Joel Emer. "A System Level Perspective on Branch Architecture Performance," *28th Int'l. Symp. on Microarchitecture*, Jun. 1995.

[Callahan] David Callahan. "Recognizing and Parallelizing Bounded Recurrences," Tera Computer Company, Seattle, WA, Aug. 1991.

[CC++] http://www.cs.caltech.edu/~paolo/ccpp/tutorial/tutorial.html. *A Tutorial for CC++,* First Edition, 1994.

[Culler et al.] David Culler, Jaswinder Pal Singh, with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach,* Morgan Kaufmann Publishers, 1998.

[DEC Alpha] R. Sites. ed., *DEC ALpha Architecture,* Digital Press, Burlington, Mass., 1992.

[Eggers et al.] Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm and Dean M. Tullsen. "Simultaneous Multithreading: A Platform for Next-generation Processors," *IEEE Micro,* Vol. 17, No. 5, Sep/Oct. 1997.

[Gupta et al.] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. "Comparative Evaluation of

Latency Reducing and Tolerating Techniques," *Proc. 18th Int'l Symp. Computer Architecture,* pp. 250-263, May 1991.

[Herlihy 90] M.P. Herlihy. "A Methodology for Implementing Highly Concurrent Data Structures," *Proc. Second ACM SIGPlan Symp. Principles and Practice of Parallel Programming, SIGPlan Notices,* Vol. 25, No. 3, pp. 197-206, Mar. 1990.

[Herlihy, Moss 91] M.P. Herlihy and J.E.B. Moss. "Lock-free Garbage Collection for Multiprocessors," *Proc. 3rd Annual ACM Symp. Parallel Algorithms and Architectures (SPAA '91),* pp. 229-236, 1991.

[Herlihy, Moss 93] M.P. Herlihy and J.E.B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures," *Proc. 20th Int'l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., pp. 289-300, 1993.

[Hwu, Patt] W. Hwu and Y. N. Patt. "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Trans. Comput.,* Vol. C-36, pp. 1496-1514, Dec. 1987.

[Jouppi, Wall] N.P. Jouppi and D.W. Wall. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III),* Boston, MA, Apr. 1989.

[Kroft] D. Kroft. "Lockup-free instruction fetch/prefetch cache organization," *Proc. 8th Annual Int'l Symp. Computer Architecture,* pp 81-85, 1981.

[Kruskal et al.] C. Kruskal, L. Rudolph, and M. Snir. "The Power of Parallel Prefix," *Proc. 1985 Int'l. Conference on Parallel Processing,* pp. 180-183, Aug. 1985.

[Laudon] J. Laudon. *Architectural and Implementation Tradeoffs in Design of Multiple-Context Processors,* Ph.D Dissertation, May 1994.

[Lee, Smith] J.K.F Lee and A.J. Smith. "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer,* Vol. 17, pp. 6-22, Jan. 1984.

[Lo, Eggers] J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Trans. on Computer Systems,* Aug. 1997.

[MIPS RISC] G. Kane and J.Heinrich. *MIPS RISC Architecture,* Prentice Hall, Englewood Cliffs, N.J., 1992.

[OpenMP] http://www.openMP.org/. *Fortran Language Specification,* version 1.0, Oct. 1997.

[Perleberg, Smith] Chris H. Perleberg and Alan Jay Smith. "Branch Target Buffer Design and Optimization," *IEEE Trans. Computers,* Vol. 42, No. 4, Apr. 1993.

[Rau, Fisher] B. Ramakrishna Rau, Joseph A. Fisher. "Instruction-Level Parallel Processing: History, Overview and Perspective," *Journal of Supercomputing,* Vol. 7, No. 1, pp. 9-50, Jan. 1993.

[Smith] J. E. Smith. "A Study of Branch Prediction Strategies," *Proc. 8th Annual Int'l. Symp. Computer Architecture,* May 1981.

[Smith, Sohi] J. E. Smith and Gurindar S. Sohi. "The Microarchitecture of Superscalar Processors," *Proc. IEEE,* Dec. 1995.

[Smith, Pleszkun] J.E. Smith and A.R. Pleszkun. "Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. Computers,* Vol. 37, pp. 562-573, May 1988.

[Sohi] Gurindar S. Sohi. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. Computers,* Vol.39, No. 3, Mar. 1990.

[Stone, 93] H.S. Stone. *High-Performance Computer Architecture,* third edition, Addison-Wesley, Reading, Mass., 1993.

[Stone, Stone] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. "Multiple Reservations and the Oklahoma Update," IBM T.J. Watson Research Center, *IEEE Parallel and Distributed Technology,* 1993.

[Tera] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Prterfield, and Burton Smith. *The Tera Computer System,* in Proc. 1990 ACM Int'l. Conference on Supercomputing, pp. 1-6, Jun. 1990.

[Tera, 1993] *TERA Principles of Operation,* Tera Computer Company, Seattle, WA, May 1993.

[Tera, SC97] Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, Richard Korry. *Tera Hardware-Software Cooperation,* SC97, Tera Computer Company, WA, 1997.

[Theobald et al.] Kevin B. Theobald, Guang R. Gao, Laurie J. Hendren. "Specualtive Execution and Branch Prediction on Parallel Machines," *Conference Proceedings, International Conference on Supercomputing (ICS'93)*, Tokyo, Japan, pp. 77-86, ACM, Jul. 1993.

[Tullsen] D.M. Tullsen, S.J. Eggers, and H.M. Levy. "Simultaneous Multithreading: Maximumizing On-Chip Parallelism," *Int'l. Symp. Computer Architecture,* Jun. 1995.

[Wall] D.W. Wall. "Limits of Instruction-Level Parallelism," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* pp. 176-188, Apr. 1991.

[Wolfe] Michael Wolfe. *High Performance Compilers for Parallel Computing,* Addison-Wesley Publishing Co., 1995.

[Yeh, Patt] Tse-Yu Yeh and Yale N. Patt. "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," *25th Annual Int'l. Symp. Microarchitecture,* Portland, Oregon, pp. 129-139, Dec. 1992.