VISIBLE SYNCHRONIZATION BASED CACHE COHERENCE

Krishna Kumar

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

April 1997

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Krishna Kumar**
Entitled: **Visible synchronization based cache coherence**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
______________________ Examiner
______________________ Examiner
______________________ Examiner
______________________ Supervisor

Approved ______________________
Chair of Department or Graduate Program Director

__________ 19 ____ ______________________
Dean of Faculty

Abstract

Visible synchronization based cache coherence

Krishna Kumar

In large scale machines, thousands of processor cycles, in other words, missed opportunities to issue floating point instructions, may be lost while waiting for a high latency synchronization or memory operation to complete, or a stall in an instruction pipeline to be dealt with. Latency is *avoided* by bringing data to a nearby locale for future reference(e.g., caching) while latency is *tolerated* by overlapping data movement with something useful. The issue of cache coherence arises whenever there are multiple copies of a shared datum in different caches of a shared-memory multiprocessor system. It is in order to maintain consistency between these multiple copies that cache coherence protocols are employed. The efficiency of latency avoidance methods is largely dependent upon the minimization of coherence traffic in the coherence protocol used to maintain cache coherency. Cache coherence protocols in general can be divided into two classes: *hardware implemented* ones and *compiler implemented* ones. Hardware implemented ones lead to large coherence traffic, and large state storage space. Conventional compiler implemented ones involve indiscriminate wasteful invalidation. There is also redundancy between synchronization operations and coherence operations. We seek to eliminate both weaknesses, by letting visible synchronization directly coordinate changes in the writability of shared data. We propose to add scalable compiler managed caches to a TERA-like multithreaded multiprocessor architecture, with user/compiler knowledge(i.e. alias analysis, dependence analysis and user directives) used to eliminate essentially all coherence traffic. To preserve scalability, we aim to use latency tolerance methods like switch-on-every-cycle multithreading, and augment this with simple, low-latency cache coherence protocols such as our visible synchronization based one.

To my father,
who shall remain an inspiration to me forever.

Acknowledgments

A fair number of people, both directly and indirectly have helped me during the course of this thesis. Since, at times I have a bit of a discursive mind, without my thesis supervisor Dr. Probst's constant efforts, I would not have had that much needed focus essential to research. My thinking has become a lot more streamlined, if you will, after discussions with him. Aside from his invaluable guidance, I would like to thank my mother who has had to put up with numerous long distance calls, mostly filled with complaints about the inclement Montreal weather and more infrequently about the long drawn-out process of thesis writing. Both she and my father, before he passed away, had instilled in me an appreciation for the subtleties of scientific research. I would also like to thank my aunt, uncle and cousins without whom I would probably still be wallowing in homesickness. Also, more importantly perhaps, I don't think I could have done without my aunt's none-too inconsiderable cooking skills. Thanks are due also to all of my friends, both here in Montreal, and elsewhere, whose dare I say, support, have helped me in more ways than one. Last, but by no means, least, the Chapters bookstore downtown has helped me from getting too bogged down, during some of those seemingly interminable winter days.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"When you can't solve a problem, manage it." - The Reverend Robert Schuller.

In today's computation intensive parallel processing world, shared memory - memories which are physically distributed but that share a single virtual address space - is arguably, the wisest architectural course to adopt[Bur]. Among the advantages of shared memory[M.D93] are processor accessibility to a uniform address space and referential transparency. Uniform shared memory space allows for construction of distributed data structures, which facilitate fine-grained sharing, and free programmers and compilers from having to put up with per-node resource limits. Referential transparency guarantees addresses and access primitives to be identical for both local and remote objects, thus freeing the programmer from worries about which node which datum lies on. The main issue that comes into focus when considering such shared memory systems [Fig. 1] is whether the available multiprocessor systems are scalable. Or to paraphrase this, whether they have the capability to (i) tolerate unpredictable fine-grained latency from any source and (ii) provide scalable long-range communication bandwidth in the interconnection network. Latency and bandwidth are the chief performance metrics that affect the scalability of a multiprocessor system.

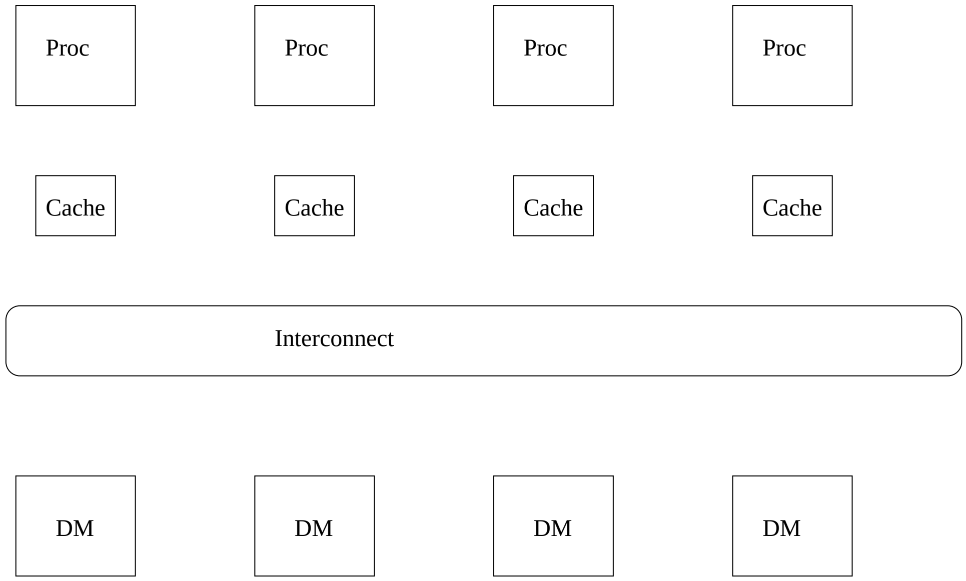

Figure 1: A distributed shared memory multiprocessor system with caches

1.1 Latency, scalability and cache coherence

In large scale machines, thousands of processor cycles, in other words, missed opportunities to issue floating point instructions, maybe lost while waiting for a high latency synchronization or memory operation to complete, or a stall in an instruction pipeline to be dealt with. If parallel languages do not keep pace with scalability requirements, i.e. do not produce enough locality to reduce bandwidth requirements as problem and machine sizes scale, we may choose to to provide scalable bisection bandwidth[D. 92a] in our machines by investing in high performance networks. Memory latency, our chief concern in the design of the cache coherence protocol, which is the focus of this thesis, on the other hand yields to a combination of the more traditional latency avoidance and the relatively non-traditional latency tolerance methods.

Latency is *avoided* by bringing data to a nearby locale for future reference while latency is *tolerated* by overlapping data movement with something useful. There are several ways to avoid memory latency. These include the caching of shared data, which allows the processor to keep a copy of frequently referenced data close by, thereby exploiting temporal and spatial locality in the data stream. The application can also be restructured to take advantage of locality (e.g., allocating data on the memory module closest to the processor most likely to use it). While all these methods improve the amount of computation a processor performs before requiring a long-latency remote memory access, often applications still end up spending a large portion of their time waiting on memory references. To deal with this remaining latency, several latency tolerating schemes have been proposed, including relaxed memory consistency models, prefetching, multiple context processors, instruction-level parallelism (which allows for more flexibility in re-ordering of instruction execution order), and vector pipelining.

The issue of cache coherence arises whenever there is the presence of multiple copies of a shared datum normally to which concurrent writes are allowed in the different caches of a shared memory system. Caches lose their integrity if the values of these different copies of the same datum are not consistent with each other. It is in order to maintain consistency between these multiple copies that cache coherency protocols are employed. The efficiency of latency avoidance methods is thus largely dependent upon the absence(as far as possible) of coherence traffic in the coherence protocol used to maintain cache coherency. Cache coherence protocols in general can be classified into two, *hardware-implemented* ones and *compiler-implemented* ones.

In the already existing hardware-based protocols, like the Stanford DASH and the MIT Alewife, distributed directories keep track of the states of cached copies of shared data, i.e., whether the copies are clean or dirty, and information as to which caches have copies of a memory location. But, the existence of distributed directories means high coherence traffic(although such distributed ones are a major improvement on the earlier centralized ones), simply because every time a shared variable is accessed, for its state to be known, the directories would have to be looked up and updated as required(not to mention the storage space taken up by the directory structure). This high coherence traffic *precludes* good scalability.

Then, there are the software-based protocols. The older ones in this class were too conservative, resulting in invalidation of the entire cache for instance, or they were slow in execution since they involved sequential invalidation of shared data structures. The newer ones like the fast selective invalidation one used at the University of Illinois[H. 88] pay very close attention to possible wasteful invalidations of non-stale cached copies of data, being very particular on high cache hit ratios.

But, our philosophy is different in the sense that we consider latency tolerance as a fundamental problem to be solved before directing our attention to latency avoidance as opposed to the afore-mentioned schemes where latency avoidance was the focus. Therefore, we think of latency avoidance as something that eases the strain, so to speak, on latency tolerance methods like multithreading and context switching. Hence, we are not obsessed with high hit ratios and are more concerned with (i) minimizing cache coherence traffic, and (ii) lessening bandwidth attenuation(in hardware-based protocols cache lines are long to include state information, leading to a decrease in available bandwidth).

1.2 Basis of our protocol

At the root of our protocol is the idea to let visible synchronization directly coordinate changes in the writeability of shared data[Pro94]. We propose to add scalable software managed caches to a TERA-like multithreaded multiprocessor architecture, with user/compiler knowledge(i.e. alias analysis, dependence analysis and user directives) used to eliminate essentially all coherence traffic. Thus, put in a nutshell, our idea is to use latency tolerance methods like deep pipelining and switch-on-every-cycle interleaved multithreading as our chief means to achieve scalable latency tolerance and augment this by latency avoidance with user/compiler controlled data caches for programs with explicit visible synchronization. Coherence actions thus become part of generated code unlike the hardware-based protocols which need hardware assistance in the form of directories and runtime bookkeeping to maintain coherence. Short cache lines result since we insist on uniword cache blocks. All of this results in graceful degradation of performance with increase in the number of processors. Visible synchronization makes coherence traffic almost entirely local, the invalidations restricted to being local operations. The global operations are memory loads and memory writebacks.

To put it more lucidly, whenever an input synchronization(a P) primitive is executed the instructions(cached loads and stores) to copy the variables protected by that synchronization block from memory can then be executed. And, whenever an output synchronization primitive(a V) is detected, an uncopy(invalidate) instruction which has been inserted by the compiler will be executed(there are special case variations to this, which we shall describe in chapter 4). The inspiration behind this idea is the fact that the synchronization variable check is imperative, and hence if the invalidation is done along with it, it doesn't become a separate activity, thus eliminating the redundance of having a separate invalidate protocol. Indiscriminate invalidation is avoided by having uncopy instructions, which invalidate only those cached variables which are are *actually* guarded within the particular synchronization epoch. Protect consistency, which is the memory consistency model we base our protocol on[Pro93b], uses programming primitives to delimit synchronization scope, and this and other user directives tell the compiler which variables are protected in which synchronization block, as we shall see in chapter 5.

The rest of this thesis is organized as follows. In chapter 2, some of the existing cache coherence protocols will be dealt with, namely, the DASH protocol and the Alewife ones as representatives of the hardware-based genre and the Cheong-Veidenbaum fast selective invalidation one as a typical example of the software-based class. Then, in chapter 3, we shall describe multithreading and its significance in multiprocessing.

We shall go into the details of our visible synchronization based protocol in chapter 4 and bring out its strengths. We shall describe the base protocol, discuss the various cache state transitions – states are indicative of permission to copy a variable from memory to a thread's cache – that a variable in memory undergoes with respect to a single thread, and finally illustrate coordinated transitions with more than one thread. Comparisons of our protocol with other already existing coherence protocols shall also be made. Chapter 5 shall be devoted to the protect consistency memory model and the basic architectural support it requires. Chapter 6 shall serve as conclusion to our discussion.

1.3 Why we invented our protocol

Hardware-implemented protocols have the following drawbacks: There is attenuation of bandwidth due to high coherence traffic and long cache lines. Aside from being extremely complex the protocols are brittle: when data access patterns trash the cache, machine performance is destroyed. They depend on unit stride for performance. Directory state storage involves too much overhead and multiword cache lines reduce bandwidth when "working sets" are small.

The contribution of this thesis is to demonstrate a compiler-implemented cache-coherence protocol that overcomes these shortcomings in the context of a multi-threaded processor architecture.

Chapter 2

Existing cache coherence protocols

"The civilization of one epoch becomes the manure of the next."
Cyril Connolly.

2.1 The DASH cache coherence protocol

Being one of the first in the "assembly" of shared memory multiprocessing based cache coherence protocols, the DASH(Distributed Architecture for Shared Memory) protocol had a lot to offer in terms of improvement in latency avoidance and hence in scalability. The concept of a single virtual address space helped make significant gains over message passing multiprocessors in areas like data partitioning, dynamic load balancing and programmability. The distributed directory scheme to keep track of the states of cached copies of variables in memory also alleviated the inherent problem of serialization present in earlier centralized directory protocols, and made multiprocessing more scalable when compared to these centralized directory schemes. Also, the need for broadcasting every memory request to all of the caches as in snooping bus schemes ceased, as directories would now contain pointers as to which cache would have copies of which memory location and whether it would be a valid copy or not. Instead, it used point-to-point messages sent between processors and memories to keep caches consistent.

2.1.1 Organization

The concept of directory based cache coherence was first proposed in the late '70s[L. 78]. However, lack of scalability due to a centralized directory soon proved to be the bane of these early protocols. The DASH architecture circumvents this limitation by partitioning and distributing the directory and main memory, and by using a cache coherence protocol that could suitably exploit distributed directories. As shown in Fig. 2, the DASH high level architecture[D. 90b] [D. 92b] is made up of a number of processing nodes grouped together in clusters with memory and directories distributed among these clusters. Distributing memory is essential to exploit data locality, both spacial and temporal, thus lessening the latency which would be experienced in case of remote accesses. Cache coherence within a cluster is maintained via a snooping bus that connects the individual processors to the cluster memory. The DASH memory hierarchy is a four-level one, the levels being, the processor level, the local cluster level, the home cluster level and the remote cluster level. A memory location can be in either of the following three states[Fig 3]:

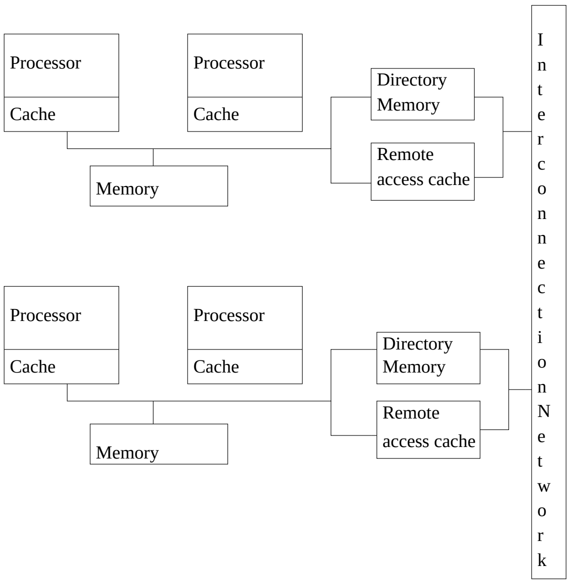

Figure 2: General architecture of DASH

Uncached – not cached by any cluster.
Shared – in an unmodified state in one or more clusters(i.e. in a read only state).
Dirty – modified(i.e. written into) in a single cache of some cluster.

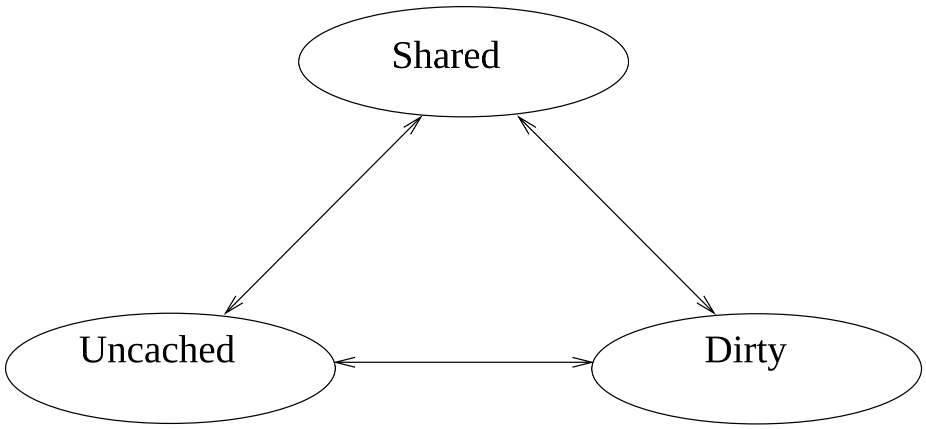

Figure 3: DASH - state diagram

The directory keeps the information for each memory block, specifying its state and the clusters that are caching it. Essentially, what happens is when there is a request which cannot be serviced by a processor's cache it is sent to the second level in the hierarchy, the *local cluster*. This level includes the other processors' caches within the requesting processor's cluster. If the data is locally cached, the request can be serviced from within the cluster. Else, the request is passed on to the *home cluster* level. The home level is made up of the cluster that contains the directory and physical memory for a given memory address. If the directory indicates that the copy in the cluster's memory is 'clean', then the request is dealt with directly, without having to go to other clusters. But, if the directory indicates the state of the copy to be 'dirty', then the request is passed on to the remote cluster which the directory indicates as having the up-to-date copy. Thus, the sequence of events that could happen on a processor read request is:

Processor level – If the requested location is present in the processor's cache, the cache supplies the data. Else, the request is relayed to the local cluster level.
Local cluster level – If the data is present in one of the caches within the local cluster, the data is supplied by that cache and no state change need be effected in the directory. If not found at this level, the request is passed on to the home cluster

level corresponding to that address.

Home cluster level – The directory state of the requested memory location is checked while simultaneously fetching the block from main memory. If the block is clean, the requesting processor's cache is supplied with the data and the directory state is updated to show sharing by the requesting cache. If the block is found to be dirty, the request is passed on to the remote cluster indicated by the directory.

Remote cluster level – The 'dirty' cluster responds with a shared copy of the data, which is sent directly to the requester, without being routed via the home cluster. In addition, a write-back message is sent to update main memory and change the directory state to indicate that the requesting cluster and the remote one have shared copies of the data.

Now, to the series of events that could occur on a write to a cache.

Processor level – If the location is 'dirty' in the writing processor's cache, the write can complete without delay. If not, a read-exclusive request is issued to obtain exclusive ownership of the line and retrieve the required data.

Local cluster level – If any of the caches within the local cluster has the up-to-date copy of the requested location, a cache-to-cache transfer of data is effected, without going to the home cluster level or the cluster directory. If the data is not present in the local cluster of caches, then a read-exclusive request is sent to the home cluster.

Home cluster level – If the directory indicates the location to be either in the uncached or shared state, the read-exclusive request can be immediately serviced, while invalidating the copies the other caches would have. If on the other hand, the directory indicates that the block is 'dirty', then the read-exclusive request is forwarded to the 'dirty' cluster. (The directory would have a pointer to tell the requester which cluster to go to).

Remote cluster level – If the directory had indicated that the memory block was shared, then an invalidation request is sent to the remote cluster to eliminate the shared copy. An 'acknowledge' message is then sent back to the requesting cluster. If the directory had shown the memory block to be 'dirty', then the 'dirty' remote cluster receives a read-exclusive request and a *direct* reply with the data is sent back to the requesting cluster. A dirty transfer message is also sent to the home cluster, so that the directory now indicates that the requesting cluster holds the block exclusively.

The other operation of note occurs when a 'dirty' cache line has to be replaced.

If 'home' was in the same cluster, then a simple write-back is effected while, if the home cluster of the block was remote, a message is sent to the remote 'home' and the directory is updated to indicate that ownership has changed hands to the remote cluster now.

2.1.2 Memory consistency model

The latency incurred by the system is directly related to the memory model it chooses to support. DASH implements the release consistency paradigm, the goal being to provide substantial flexibility in the ordering of memory requests, while still providing the user with a reasonably simple programming model.

At one extreme of the memory model spectrum is the very rigorous interpretation of the sequential consistency scheme which constrains every memory request to be completed before the next one is issued, thus causing high latencies on memory accesses. Albeit simple in concept, this puts too many restrictions on program order. It is useful only when any operation could be a synchronizing operation. For instance, supposing a data structure needs to be updated several times within a critical section, a sequentially consistent model forces invalidation of all other copies of the data structure on every write which is unnecessary once the synchronization points are identified since memory needs to be consistent only at those points. This in essence is the release consistency model which thus puts fewer restrictions on memory accesses, requiring as it does that the system need be consistent only at synchronization points.

The release consistency model furnishes the user with a reasonable programming model, since the programmer is now assured that all processes would have a consistent view of the updated data structure, once the critical sections are exited. Also, it permits reads to bypass writes and the invalidations of various write operations to overlap. Reads are stalling operations while writes are enqueued, allowing them to be non-blocking.

Although the use of release consistency does help in overcoming, i.e. hiding latency, the processor does still stall on reads, which can lead to considerable delays when programs exhibit poor cache behaviour(i.e. locality). Thus, to supplement release consistency, DASH provides a variety of prefetching and pipelining operations. Further, it furnishes the user with the 'update' primitive which aids in writing synchronization variable values to all processors requiring them. Then, there is the 'deliver' primitive which helps in passing on values of updated variables to processors caching them. Queue-based locks in which processor requests for the 'lock' are queued are also implemented. Once the lock is 'free' it is granted to one of the waiting clusters at random. A time-out mechanism is also provided in case the requesting process has migrated or has been swapped out.

2.1.3 Shortcomings

Having brought out the prinicipal features of the DASH design, we shall now very briefly touch on its shortcomings (more details of which will be described when we discuss the intricacies of our protocol), which we have attempted to rectify in the design of our visible synchronization based protocol. Despite having reduced serialization and coherence traffic vis-a-vis the earlier centralized directory based schemes, there still is the presence of a fair amount of latency due to *non-local* cache coherence traffic. Directories beget a considerable amount of communication, since, in order to maintain consistency between caches, there is a need for a significant amount of run-time bookkeeping. The other source of latency is the release consistency memory model which although having eliminated many of the redundant memory access constraints that were imposed by the sequential and weak consistency models, still causes some unnecessary data$-$ >synchronization precedence arcs. This shall be dealt with in detail in chapter 5, when we discuss protect consistency[Pro93b]. Also, since synchronization and the invalidating itself are two separate mechanisms, there is an amount of redundance because essentially they deal with the same cache block, and could be collapsed into practically the same procedure as we have done in our protocol. Indeed, hardware implemented invalidation harmlessly but redundantly redoes the work of synchronization.

2.2 The MIT Alewife

The Alewife architecture supports upto 512 processing nodes over a scalable cost-effective mesh network. The MIT Alewife machine, a prototype implementation of this architecture, is similar to DASH in the sense that it too uses distributed directories and interconnects the nodes via a mesh network. But, there are three differences of note between the two architectures[D. 92b]:

* Alewife does not support the notion of clusters – each node consists of a single processor.

* Alewife uses software to handle directory pointer overflow i.e. only memory locations which are shared by less than six processors are pointed to by directory hardware. For more widely-shared locations, the protocol issues a trap into software and the software does the write invalidations as required.

* Alewife applies block multithreading as its primary means to tolerate latency, whereas DASH banks principally on prefetching in its efforts to decrease latency. It should be noted however that Alewife does augment multithreading with prefetching.

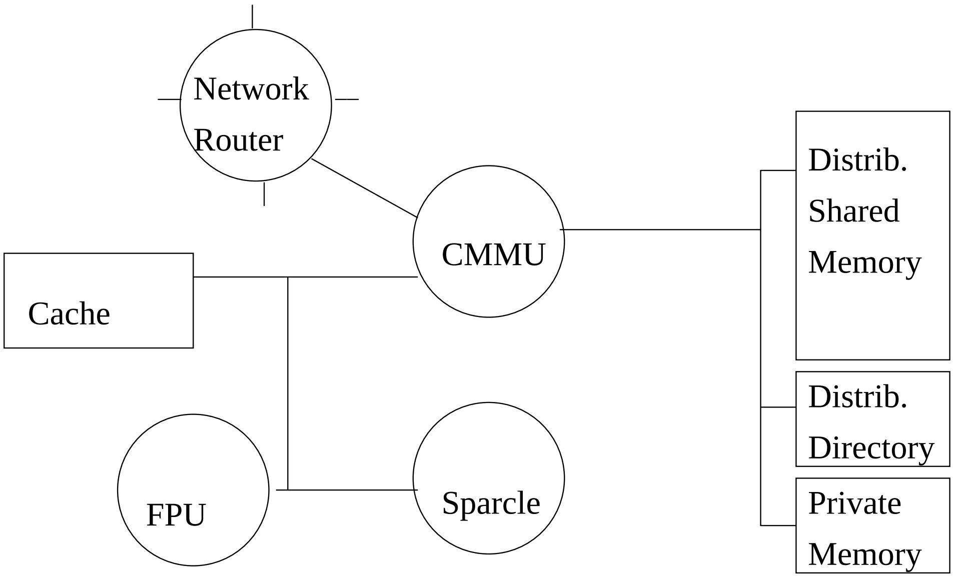

Figure 4: An Alewife node

The MIT Alewife employs four techniques[A. 95] as means to the end of scalability and programmability: *software-extended coherent shared memory* provides a global linear address space; *integrated message passing* affords compiler and operating system designers with efficient communication and synchronization; support for *fine-grain computation* allows many processors to cooperate on small problem sizes; and latency tolerance mechanisms – including *block multithreading* and *prefetching* – nullify to a large extent, delays due to communication.

Analysis[A. 95] shows integrated message passing with shared memory to be a cost-effective solution to the cache coherence problem and it provides the user with a rich set of programming primitives. Block multithreading and prefetching improve performance by upto 35 percent. Also, language constructs that provide programmers with a method to express fine-grain synchronization can enhance performance by over a factor of two.

2.2.1 Coherent shared memory

On each of Alewife's distributed nodes which constitute the abstraction of shared memory, a Communication and Memory Management Unit(CMMU) [Fig. 4] responds to memory requests from a Sparcle processor and determines in consultation with the distributed directory whether requests need access local or remote memory. When required, the CMMU synthesizes messages that transfer data from remote nodes. Data locality is implemented by the caching of both private and shared data on each node. A scalable software extended cache coherency scheme called LimitLESS which uses hardware to support data consistency of "common-case" memory accesses and traps into software for more widely shared memory locations, is employed.

2.2.2 Integrated message passing

While the programmer at a higher level of hierarchy enjoys the luxury of a shared memory programming paradigm, for performance reasons, much of the underlying software is implemented using message passing. Features in Sparcle and the CMMU blend to provide a streamlined interface for transmitting and receiving messages: both system and user code are empowered with the capability to describe and atomically launch message packets into the interconnection network; a fast interrupt mechanism that facilitates message reception and a Direct Memory Access(DMA) mechanism that allows data flow between network and memory whilst maintaining coherence between the data in messages and the data in local caches are other salient features.

2.2.3 Fine-grain computation

For a given data set the granularity of computation(the interval between events that require inter-processor communication) decreases as the number of processors in a multiprocessor system increases. Therefore, a system incapable of performing small tasks with a degree of efficacy, must resort to increasing synchronization and communication granularity artificially, which in turn, would lead to loss of locality. Hence, it is significant that Alewife's repertoire of fine-grain computation includes fast, user-level messages and support for full/empty bit synchronization, thereby preserving locality. Constructs that provide for implicit synchronization of threads upon every memory access are included in Alewife's programming languages, parallel C and Mul-T.

2.2.4 Latency tolerance

Block multithreading and prefetching contribute significantly to the Alewife scheme of reducing latency, when it is unavoidable otherwise. Multithreading which allows a processor to switch between computational threads on a latency incurring operation such as a cache miss or a failed synchronization attempt(thus getting useful computation done while waiting for the memory-to-cache data transfer or the waiting for a synchronization signal), requires a fast context switch mechanism which in turn is provided by the Sparcle processor's fast interrupt techniques. The other requirement of a multithreading environment is a solution to the "window of vulnerability" opened up by interleaved threads of execution[J. 92], thus leading to the possible invalidation or removal of the requested memory location before it is actually used by the processor. The CMMU's transaction buffer allows for outstanding cache requests to be buffered and thus provides the "antidote" to this ill, which can also come about as a result of fast message handling and the software-extended LimitLESS coherence scheme. Alewife implements lock-up free caches[D. 81] with input stacks to pipeline requests, to satisfy prefetching requirements. The transaction buffer doubles as storage unit for prefetched blocks.

2.2.5 Implementation and working of the above techniques

2.2.5.1 Coherent shared memory

Every 16 byte long memory line of the physically distributed shared address space is assigned a home node for data storage and a coherence directory. The CMMU of this home node directs all coherence operations of this memory location, be it hardware or software controlled. Every node accounts for the data and coherence directories of

4M bytes of shared memory.

Cache lines in Alewife are 16 bytes in length and coherence is maintained through a software-extended scheme called LimitLESS, which supports upto 5 readers per memory line in hardware, but for more widely-shared locations, traps into software. The Alewife message passing interface is made use of in handling remote requests which have been trapped into software. The queueing that comes along with the message passing interface aids in servicing pending coherence requests.

The Alewife directory format is as follows: Each entry is 64 bits long, with five 9-bit pointers which serve as the hardware pointers to the five different nodes(there being 512 nodes in all) which have copies of the particular location. There are 2 bits of state, a further 2 bits of meta-state and four full/empty bits(one for each 16 bit word in the line). The other bits serve the following purposes. There's a *local bit* which allows for an optimization that guarantees the local node the acquiring of a pointer. The *pointers in use* field indicates the number of pointers that are being actually used at that point in time.

The essence of the LimitLESS coherence(Limited Directory that is Locally Extended through Software Support) scheme is the assumption that only very few data structures are widely shared[D. 91]. That is, the working sets of shared data structures are assumed to be small, the assumption being backed up by observation. By working sets we mean the group of processors that share a particular memory block at a given point in execution time. And to handle hardware pointer overflow if the working set is greater than 5 the software facility is made use of. Once the number of processor caches that share a data structure goes beyond 5, a trap into software is issued and the software emulates the full directory. Also, the hardware pointers are flushed into memory resetting the pointers, the memory flush providing a means to keep track of the pointers so that cache reads can still be carried out via hardware until a software trap occurs again(i.e. for the next five read requests). But writes from then on are handled exclusively by software.

Now, to the chain of events that could occur on a read/write. First, assuming all writes and reads are being handled in hardware, i.e. the "normal" bit being set("normal" being one of the directory meta-states), if a read request occurs the following courses of action could be taken:

If the state bits in the directory indicate the read-only state and the pointers in it indicate which cache has a copy of the requested datum, the data is given to the requesting cache whilst updating the pointer list to include the requesting cache. The trivial cases are when the requesting cache does already have a copy and when no cache holds copies. In the latter case, the pointer list is updated to hold just the information that the requesting cache now contains the datum. If the directory state in memory indicates the read-write condition and a cache requests a copy of the datum under consideration, the memory goes into the read-transaction state, holding the read request while the cache which held the copy in read-write mode, updates memory and then, the requesting cache is supplied with the data and the memory goes into the read-only state.

Now, if a write request was sent by a processor to a cache, the following happens: If the directory indicates that all caches that held copies of the datum in question, the data is written into the cache which was the originator of the request and sends invalidation messages to the rest; the trivial case being when the requester cache already has a copy in read-write state, causing the write request to be serviced directly. In both of the above cases, the memory goes into the read-write state once the write request is granted. If the memory is already in the read-write mode, with a few other caches exclusively holding a copy of the datum, the write request is held up and the memory goes into the write-transaction state, while invalidating the previously cached copy. Following this, an update is sent to the requesting cache via memory, where the fresh value has just been deposited and the memory makes the transition to the read-write state again(except now, the exclusive ownership belongs to the requesting cache).

2.2.5.2 Message passing

Messages in Alewife are sent using a two phase process: (i) describe and (ii) launch. A message is described by writing directly by writing directly to an output descriptor array which essentially is 16 memory-mapped network registers in the CMMU. Once a message is described, it is launched using the *ipilaunch* instruction; the instruction being atomic and using up a single machine cycle. A message is comprised of a header – which contains the packet header, a dataword – that contains the dataword, an address – which points to the start of the data block and “length” – which gives the number of double words in the block.

Message reception entails either polling or interrupts. As soon as the message header is received, an interrupt is posted at the receiving end. Since one of the four hardware contexts in a Sparcle processor is reserved for message processing, there is no need to save registers or the need for restoring them. Message passing incurs more network traffic than shared memory but makes more effective use of available bandwidth.

2.2.5.3 Fine-grained synchronization

Alewife provides the programmer with fine-grain synchronization primitives, which allow for more instruction-level parallelism to be presented to the hardware for further hardware-level parallelism optimization. There is an appropriate mixture of software and hardware support for fine-grain synchronization. Hardware support consists of a full/empty bit for each 32 bit word and to access these bits, coloured load and store instructions are provided that perform full/empty test-and-set operations. Two Sparcle instructions bempty(branch on empty) and bfull(branch on full) can be used to examine the bit.

The system provides several language extensions for fine-grain synchronization in the form of J-structures and L-structures. J-structures support full/empty bit synchronization, for producer-consumer style synchronization on vector elements. A J-structure read waits for a vector element to be full before it is read whereas a J-structure write updates the element and sets it to full. An L-structure provides for mutual exclusion style synchronization on vector elements by means of a full/empty bit associated with each element; a locking read, an unblocking write and a non-locking read being the operations the programmer is furnished with.

2.2.5.4 Latency tolerance

Latency tolerance in Alewife is achieved chiefly by the following means: block multi-threading and non-binding software prefetching.

The following features in Alewife help implement and facilitate block multithreading: (i) The CMMU takes advantage of as much parallelism as possible when serving a remote cache miss, by generating a context-switch trap in parallel with message generation. (ii) Lockup free caches which enable outstanding cache requests to be serviced are implemented. (iii) A livelock avoidance technique is provided, to eliminate the possibilty of livelocks arising when cache-coherent shared memory is coupled with context-switching and LimitLESS.

Software prefetch is implemented as two different coloured load instructions, one for read prefetch and the other for write prefetch; the value returned from the prefetch instruction is ignored. Prefetched data is stored in the transaction buffer, which also serves as a mechanism to counter the "window of vulnerability" problem[J. 92] due to split-phase transactions.

2.2.6 Shortcomings of Alewife's software-extended scheme

Although the Alewife scheme has succeeded in reducing network traffic by employing a limited directory scheme supplemented by an extension of the directory in software, there is still the presence of a directory which does beget considerable traffic. Plus, there is the redundancy of having a separate cache coherence protocol as an activity distinct from doing a synchronization variable lookup, which can really be implemented together as we do in our visible synchronization based protocol – this saves a lot on coherence traffic. Also, instruction-level parallelism can be improved upon further by providing for a more relaxed consistency model than the release consistency model employed in most existing schemes. These are the areas where we focus our attention on, in our cache coherence protocol, as shall be seen later in Chapters 4 and 5. Additionally, block multithreading we feel is not the answer to latency tolerance because of too much context switch cost as compared to interleaved multithreading, as we shall see in the next chapter. These are the areas on which we focus our attention in our cache coherence protocol, as shall be seen later in chapters 4 and 5.

2.3 The Cheong-Veidenbaum fast selective invalidation protocol

The Cheong-Veidenbaum[H. 88] fast selective invalidation scheme is representative of a new generation of software-based protocols which answer to an extent, an oft-made criticism of software based schemes – that of indiscriminate invalidation – and can, in addition selectively invalidate cache items without extensive run-time book-keeping(which for instance, hardware-based protocols entail). The solution depends on a combination of compile-time reference tagging and reference-time invalidation, i.e. a cache line is invalidated only when referenced, although potential staleness is indicated by the tags at compile-time.

Hardware-based protocols require extensive bookkeeping whereas most earlier software-based ones were too conservative on the invalidation front. Either this, or as is in the case of the one-time identifier(OTI) approach proposed by [A.J85], sequential invalidation is effected and this runs up a heavy time penalty. To elaborate briefly on Smith's scheme, the principal idea behind it is to have an OTI associated with each translation lookaside buffer(TLB) entry and each cache line. A new OTI is generated for each new TLB entry and when this entry is loaded afresh into the cache, the OTI values of the cache line and the corresponding TLB entries become the same. The processor issues an invalidate TLB entry instruction for each page in a write-shared region and therefore the next time the cache entry is accessed, the run-time system finds a disparity between the OTI value of the TLB entry (the OTI value being invalidated) and that of the cache line. Hence, a cache miss is effected and the newly written value is reloaded from memory. Among the drawbacks of this scheme are (i) the fact that disparity in OTI values may be caused by page replacement other than just by invalidation and (ii) slow sequential invalidation of write-shared pages.

Various schemes such as the ones in [A.V86], [K.K87], [R.L87b] [R.L87a] all use indiscriminate invalidation methods which are at but slight variance with each other. Perusals of all the aforementioned protocols lead us to the conclusion that at the crux of the issue lie the following points: (i) when to invalidate and (ii) what to invalidate. The Cheong-Veidenbaum selective invalidation scheme hit upon the rather ingenious policy of compile-time analysis of where invalidate-cache statements should be inserted and which cache lines should be marked as cache reads/memory reads combined with reference-time invalidation, thus doing away with overinvalidation and the cumbersomely slow sequential invalidation, in one shot.

We shall further elucidate on the Cheong-Veidenbaum strategy with the aid of the following example and later delineate the general scheme. Every shared-memory reference is marked as either memory-read or cache-read. A read reference will be marked as a memory-read by the compiler if the cached copy may have become stale; that is, it marks potential staleness. Read accesses will be marked cache-read if cached copies are guaranteed to be up-to-date.

In the example in Fig. 5, the doall loop indicates a write-shared region, but

```
do serial     i = 1, n
          .
          .
          =    .  .  .  Y(i)              /*  Cache read */
          .
          .
          .
enddo
doall   j = 1, n
/*an invalidate-cache would be inserted here */
          .
          .
          =    .  .  .  Y(j)              /* Cache read */

          .
X(j)   =    .  .  .
          .
          .
          .
enddo
/*an invalidate-cache would be inserted here */
do serial   k = 1, n
          .
          .
          .
          =    .  .  .  X(k)

          .
          =    .  .  .  Y(k)              /* Cache-read */

          .
          =    .  .  .  X(k)              /* Memory-read */
          .
          .
 enddo
```

Figure 5: The Cheong-Veidenbaum protocol - a trivial example

since Y is only read, it is marked cache-read while X is marked memory-read in the subsequent serial loop because it has been written into by more than one processor in the doall loop.

Two invalidate cache statements are inserted at the beginning and end of the doall loop, the purpose of which we shall explain when we take up the general case. Also, the second reference to X(k) in the second doserial loop is tagged as a memory read. This is implemented by means of a change bit which is set to true by the invalidate cache instruction. Once a datum has been fetched into the cache, this bit is reset to false, a reference is treated as a hit, else a miss is recorded. Thus, a second (wasteful) memory read of X(k) is avoided because the change bit indicates the fact that it has just been loaded into the cache. The invalidate cache instruction in this case, being solely an indicator of potential staleness(and acting as a guide to the compiler) rather than of unequivocal non-validity.

Now to the general criteria for reference marking of potentially stale data. Reference markings are based on the location of invalidate-cache instructions. Since compile-time analysis is used, the ordering of read-write accesses is determined by flow analysis.

The reference marking scheme is applied to one subroutine at a time and it is assumed that a subroutine starts off with a clean cache(the clear- cache instruction being executed to this end). Variables which are only read within a subroutine are considered read-only and references to them are tagged cache-read. For read-write variables, all references that precede the first write to that variable are marked cache-read. The remaining class of references - that of reads following the first write access to a variable - are marked according to the following rules[H. 88]:

Case 1

A read reference is issued by the same processor that performed all the previous writes. This reference can be tagged as cache-read, because the processor can only read either unmodified data or data modified by itself, thus getting the up-to-date copy in both cases. A read reference is identified by the compiler as belonging to this case if the read reference and all writes preceding it(in run-time order) lie within a pair of consecutive invalidate- cache instructions.

Case 2

The read reference is to a variable which has either been modified by the current processor or for which no copy exists as yet in the processor's cache. Such references are also tagged cache-read. The characterizing features of this category are: (1) All preceding writes to the variable are enclosed within a pair of consecutive cache-invalidate statements, (2) there are no read references before the first of the cache-invalidate statements, so that the cache does not hold stale values from before and (3) the read reference under scrutiny appears after the second cache-invalidate statement. It is to be noted that between pairs of cache-invalidate statements, the reads of one processor do not depend upon the writes of another, the particular processor being guaranteed exclusive access to the variable. Hence, the read reference can be marked as cache-read.

Case 3

A read reference is made to a variable for which a stale copy might exist in the current processor's cache. A read reference belongs to this class if (1) the preceding writes are separated by at least one cache-invalidate instruction or (2) all preceding writes are within a pair of consecutive invalidate-cache instructions but one or more reads precede the first of the pair.

There exists a possible situation in which a line size of greater than one is in use; the problem arising when two processors access the same cache line but different words in it, thus resulting in the possible fetching of stale data, termed "false sharing". Both software and hardware solutions exist. One software solution is to enforce another restriction to the marking of accesses as cache reads in Case 2. Specifically, in addition to the restriction applied in Case 2, the following condition is added: The read reference can be tagged cache-read if there are no read references to the same variable that is enclosed within the two cache invalidation statements. Else, it is marked memory-read.

In the case of subroutine calls from within the current subroutine, if these subroutines can be expanded, they are, before the insertion of the invalidate-cache instructions. If they are not expandable, either all variables to which references are made within this "new" subroutine are conservatively marked memory-reads and a cache invalidation instruction inserted after the call, or a clear cache instruction is included right after the call and the reference marking algorithm is applied to the following code as described in the cases mentioned before.

2.3.1 Possible improvements

The Cheong-Veidenbaum scheme does thus counter the main criticism of software-based schemes, viz., their "conservatism" when it comes to invalidation. But, there is still room for improvement in areas such as instruction-level concurrency. This is where less restraining models like the protect consistency one we shall describe later play a most significant role. Another area which we look to improve upon is simplicity and elegance of implementation which admittedly although a trifle on the subjective side of things, is still a very important issue. We hope to achieve the same performance if not better it, with an implementation more directly linked to the consistency model chosen than the Cheong-Veidenbaum one. And, since the PRC consistency paradigm is a more relaxed one, we seem to be at a significant advantage, as we shall see in chapters 4 and 5.

In the next chapter we discuss the need for multithreading techniques, and interleaved cycle-by-cycle multithreading in detail, since we strongly believe that the accent in multiprocessor systems should be laid on latency tolerance which is to be then augmented by cache coherence protocols which entail minimum coherence traffic and relaxed memory consistency models such as protect consistency. We chose interleaved cycle-by-cycle multithreading because that view is more widely accepted as true[J. a]. We repeat the idea that multiprocessors are necessarily built using commodity microprocessors used in uniprocessors, although some people use this false idea to justify the proposition that, if a form of multithreading needs to be incorporated into multiprocessors, it needs to work well on uniprocessors too. Indeed, we do not care a whit for single-thread performance, only single-program performance.

Chapter 3

Multithreading

One thread to another. "An idle thread is the devil's workshop. So, hark comrades! Let us run, rush to its aid!"

3.1 Latency and how to deal with it

Subject as they are to heavy computational loads, scalability becomes the chief performance criterion by which to judge the efficiency of modern multiprocessor systems. The key to any multiprocessing system's effectiveness is thus whether latency destroys processor utilization, which in turn is the main factor in deciding system scalability. The three principal sources of latency are the following: Memory access latency - that caused by delay in accessing remote memory, synchronization latency - that which occurs on failed synchronization attempts, instruction latency - which results mainly due to pipeline stalls caused by data and functional dependencies among instructions. Of the above, memory latency is the most prominent.

Memory latency is incurred largely as the result of two related factors. Firstly, because multiprocessing is essentially cooperating processes executing on separate processors, cache misses occur due to data communication between the two processors. This very often overshadows initial uniprocessor cache misses and those that occur because of uniprocessor address collision. Then, there are the remote memory accesses that probably are the single most significant cause of latency.

Synchronization and instruction latency are relatively less dominant than memory latency, but still are significant in magnitudes. Synchronization latency includes time spent by a process waiting for another process, for instance, when a barrier construct is employed to synchronize all processes or when waiting to acquire and release locks. Instruction latency arises when there are "bubbles" in a pipeline due to either data or functional dependences between instructions in a pipeline.

A lot of the above latency can be nullified by coherent caching and block prefetching in the case of memory latency, hardware and software primitives for fast synchronization and fast functional units and pipeline bypassing with aggressive compiler scheduling(supported by relaxed memory consistency models)for alleviating instruction latency. However, despite all of these techniques, there still remains a fair amount of latency to be dealt with. This is where multithreading can play a most significant role.

Multithreaded or multiple-context processors tolerate latency by overlapping the long-latency operations of one computational thread with the execution of other thread(s) by switching contexts, both hardware and software, to the other thread(s), thus getting useful computational work done when otherwise it would have waited idly.

3.2 Multithreading techniques

As mentioned before, multithreaded processors share a single processor between several computational threads, switching to or opting for a different context either when encountering latency on operations involving one thread or automatically on every cycle. Context refers to the registers and memory hardware and the instructions involved in the execution of a portion of a thread. Certain prerequisites[J. b] have to be satisfied for this context switching to be efficient in its implementation, of which the following are of primary importance. (a)The context switching cost, or put more quantitatively, the number of cycles taken in context switching should be much lesser than the number of cycles the latency causing operation would consume. (b)And, there should be a fair amount of exploitable parallelism in the program to be run, although by stressing on good single program performance[J. a], the accent on this condition is lessened. By decreasing the number of contexts involved in latency tolerance, the range of workloads to which multithreading can be applied can be broadened. Some people believe this is of particular significance because they believe that most multiprocessors consist of commodity microprocessors and thus the small amounts of inherent parallelism present in many typical workstation applications for which most commodity uniprocessors are built have to be taken into consideration. The first condition is important because multiprocessor latency runs the whole gamut from the very low in the case of pipeline dependencies or primary cache misses, to the much higher on remote memory acccesses and interprocess synchronization.

Most existing multithreading schemes have been designed with the large-scale multiprocessor(with larger memory latency and more inherent parallelism) in mind, and thus are not tailored to the needs of commodity microprocessors which typically run jobs involving smaller latencies and lesser amounts of paralellism. But since commodity microprocessors like the ones used in workstations are those that make up multiprocessors these days, [J. a] it is believed that only if multiple contexts address the requirements of such processors, would they be incorporated into multiprocessor systems. Thus because of their emphasis on multiprocessors, already existing multi-context architectures require either too many contexts to fill the instruction pipeline and handle memory latencies, these in turn needing large inherent parallelism in programs, and suffering from poor single thread performance(and when there are significantly large serial portions in an application)or are unable to handle smaller latencies as a result of too big a context switch cost. Fine-grained multiple-context processors belong to the former class while blocked multiple-context processors are representatives of the latter one.

3.3 Existing multiple-context approaches

In this section, we shall first be describing the two aforementioned multiple-context (multithreading) solutions to the latency tolerance problem and later, we shall turn to the interleaved approach which is in a sense an amalgam of these two techniques and delineate the advantages of employing this method.

3.3.1 Fine-grained multiple-context processors

Multiple-context processor architectures made their entry into the multiprocessing world in the 1960's; among the earliest ones was the CDC 6600, the idea behind it being the time-sharing of CPU's and memory interconnects between a number of peripheral processors. The focus of these early architectures was on fine-grained multiple contexts, possibly because of the simplicity of context switch, the seminal representative of these being the Denelcor HEP made in the late 1970's. The HEP consisted of a small number of processors connected to each other by a packet-switched network. It used fine-grained multithreading to circumvent the problems of both pipeline stalls and memory latency. Pipeline stalls were dealt with by ensuring that an instruction would be issued only once every eight cycles (this being the pipeline depth), thus precluding pipeline dependency. This in turn frees hardware and compiler resources from being devoted to resolving pipeline data dependencies(except in the case of the divide unit, which could not sustain an issue rate of one divide per cycle). Also, HEP tolerated memory latency to a small extent by removing an instruction stream from the issue queue while memory was being accessed, a sort of primitive context switch. But, the absence of data caching in the HEP meant that a large number of contexts were still required to hide both pipeline and memory latencies, indeed it had up to 128 active contexts at a time.

Architecture-wise, the Horizon was a successor to the HEP. The major advancement of the Horizon which was also a fine-grained multiple-context processor was the inclusion of a 3 bit lookahead field. This lookahead field encoded inter-instruction dependencies, providing limited compiler-resolved pipeline interlocks, thus affording the capability for multiple instructions from the same context to use the pipeline simultaneously. But again, no architectural support for caches was provided which meant that memory access times were around 50-80 cycles, which in turn meant that a lookahead of only eight instructions would be not nearly large enough to fill the pipeline, albeit decreasing the number of required active contexts. Thus, the single thread performance was not quite good enough as yet.

The Tera design is to a large extent based on that of the Horizon. Support for thread management and synchronization is enhanced and this is a definite improvement over the Horizon. But, although, it requires only memory operations to be controlled by the lookahead field, on the flip side, it reverts to the HEP policy of respecting register dependencies by allowing only a single instruction per context; however, memory instructions can be issued once the lookahead condition is satisfied by all previous instructions. With an expected pipeline size of 11 to 13 stages, these many contexts are required at the very minimum, inspite of the lookahead provision.

Some other recent architectures have been developed based on the fine grained scheme including MASA, DART and the Stellar SPMP. However, due to their inability to handle limited parallelism(the limitations of fine-grained schemes for workstation loads are quite severe) - placing two onerous burdens on applications: first, a large number of threads are required to fully utilize the processor and hide memory latency and second, the single thread performance is extremely poor as one context can at best issue one new instruction every pipeline depth-cycles - a gradual progression of ideas led to the proposing of blocked multiple-context or blocked multithreaded processors.

3.3.2 Blocked multiple-context processors

Blocked multiple-context processors do not change contexts every cycle, waiting for long-latency operations to switch contexts or threads. Hardware is provided for a small number of threads, but at a particular point in time, only one thread will be executing. Several of the earlier blocked multiple- context architectures were made with specific high-latency operations in mind: The Xerox Alto personal computer provided microcode-level register sets for sharing the CPU between the instruction set interpreter and the I/O devices, the latter capable of incurring high latency. Thus, if the I/O process seemed like entailing high latency, a context switch between the process requiring the instruction set interpreter and the one requiring I/O is carried out. Another example of blocked context switch being implemented to the end of satisfying a particular operating system or user requirement was the Message-driven processor, which provided two hardware contexts, one for normal processor execution and the other for handling high-priority messages.

The first published exploration of shared memory multiprocessors that used multiple-contexts was performed by Weber and Gupta, the point at which they did the context switch was on every primary cache miss. They found significant performance enhancement even with relatively fast memory chips whose access times were 20-30 processor cycles; this with only a few number of contexts - two or four per processor. The Alewife multiprocessor based on the APRIL processor also uses the blocked multiple-context scheme - switching contexts on remote memory requests and failed synchronization attempts - supporting 4 contexts and switching via a fast processor trap in 11 cycles.

Blocked multiple-context processors thus get rid of the over-dependence on a large number of threads/contexts that the fine-grained multiple-context processor had. But, in ironing out this fault, another creeps in. Due to the fact that the processor waits for a long latency operation to switch contexts, the context-switching overhead is large and puts an upper bound on processor utilization, thus restricting the range of latencies that can be tolerated - shorter latencies like those caused by pipeline dependencies cannot be overcome now. Assuming that memory latency will be completely tolerated, the maximum processor utilization is given by Ra/(Ra+Rs+C) where Ra is the number of active cycles(time spent doing useful work), Rs the time spent waiting due to pipeline stalls and C the context switch time. Taking a typical example, a 20 cycle average active interval (Ra=20), a total pipeline stall time of 10 cycles for this Ra and a context switch time of 10 cycles would mean an effective processor utilization of 50 percent which is too small for most applications.

Pipeline register replication was seen as a solution to this large context switch cost. Using pipeline register replication, context switch cost was theoretically reduceable to one cycle(bypassing the pushing of register information into stacks on each context switch) - the one cycle being the time required to broadcast the switch information so that it could be made use of by the translation lookaside buffer, pipeline forwarding logic etc. But, the practical implementation side of things were very different as replication needed latches to hold the state information about the contexts and then these needed to be multiplexed when control shifted between various pipeline stages, and all these added to the total pipeline area leading to delays - the context switch cost hence not being significantly lessened. Thus, more subtle methods were seen to be required to decrease this context switch cost; to be more specific, a review of the existing fine-grained scheme was in order.

3.3.3 The interleaved proposal

The interleaved proposal attempts to correct the two shortcomings of the fine-grained context-switch scheme rather than do away with it altogether by proposing modifications to the blocked context-switch one, whose philosophy of approach is entirely different. These shortcomings, it is felt[J. a] are not inherent ones which come along with every fine-grained scheme but are offshoots of two basic ideas which were incorporated into the early fine-grained schemes: the absence of data caching and the restriction of instruction issue to one every pipeline-depth cycle, both of which can be done away with because of improvements in caching and pipelining respectively.

With caching every data access is no longer a long latency operation and there is no more the need for a large number of contexts to tolerate memory latencies, thereby alleviating the over-dependence on a large number of contexts(one major drawback of fine-grained multithreading) to an extent. Thus, better support for single contexts is achieved. Also, more than one instruction is issued per context every pipeline-depth cycle, decreasing the instruction latency to much less that what it was with the conventional very conservative fine-grained scheme.

The working of the interleaved scheme can be best described with an example. Assuming that each processor has four active contexts as shown in Fig. 6, again as shown in the figure, every processor keeps switching between these four contexts every cycle in a round robin fashion. Each time a long latency operation is encountered, the particular context which caused the operation is taken out of the ready context set(i.e. it is made temporarily unavailable), and the round robin switching is continued with the existing available contexts. Once the long latency operation is complete(i.e. if it was a memory request, the data was obtained), the context is added to the end of the round robin pool. Taking the example of the following contexts:

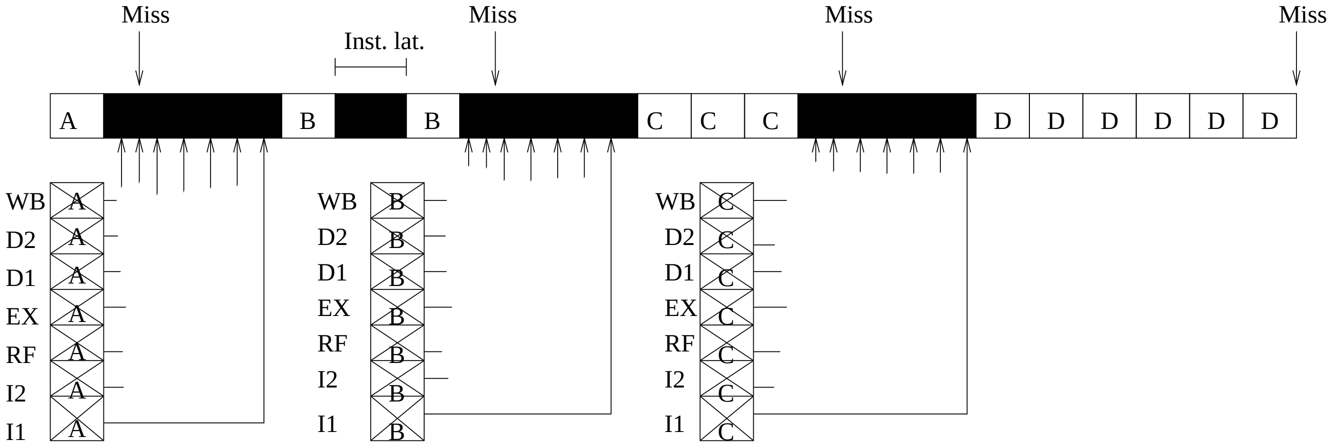

Figure 6: Blocked scheme

Context A - issues two instructions, the second of which causes a cache miss.

Context B - issues one instruction, followed by a two cycle pipeline dependency

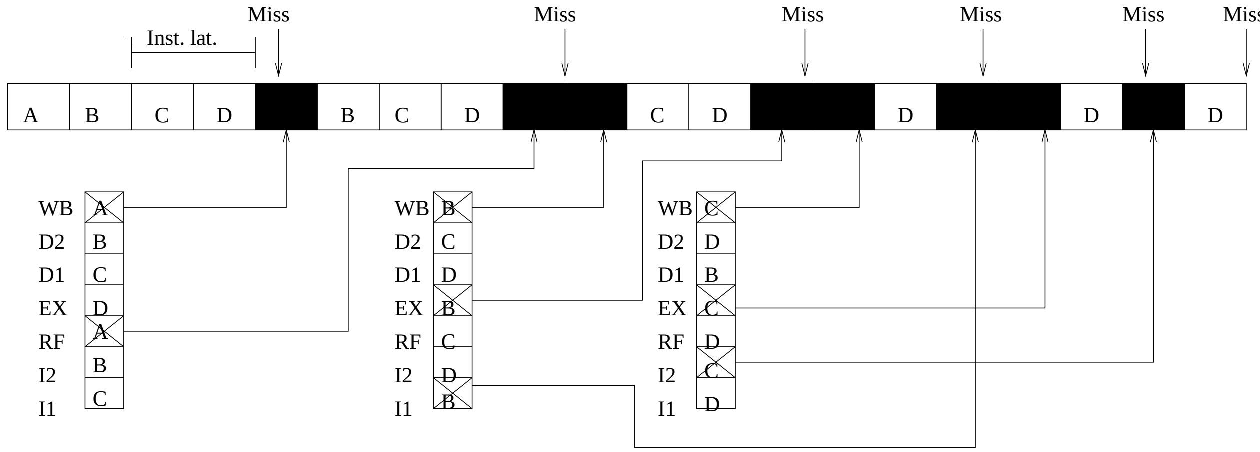

Figure 7: Interleaved scheme

stall, followed by two more instructions, the last of which causes a cache miss.

Context C - issues four instructions, the fourth resulting in a cache miss.

Context D - issues six instructions, the last one causing a cache miss.

Outlining what would have happened if we were to adopt the blocked context switch case will make clearer what benefits the interleaved scheme holds. As shown in Fig. 6 the blocked scheme works as follows. Context A starts execution and issues two instructions and once at the WriteBack stage, the second instruction causes a cache miss, as a result of which the entire pipeline has to be flushed before the context switch is made - the context switch cost thus being extremely high, since the switch decision is reached so late in the pipeline(the WriteBack is the last stage in the pipeline). Same is the case with context B. It executes once the switch from A is made and issues one instruction, then has to wait two cycles because of a pipeline dependency. After the pipeline stall, it issues a further two instructions and the last of these causes a cache miss in the WriteBack stage. Again, a wasteful flush of the entire pipeline has to be effected. Plus, when the earlier pipeline stall occurred, you couldn't turn to a context switch as a means to tolerate the instruction latency due to the stall, because the stall takes up only two cycles whereas the context switch cost would be larger. Compare this to the interleaved scheme which works like the following.

The interleaved scheme starts off with four interleaved contexts. As can be seen in

Fig. 7, interleaving causes sufficient spacing between the interdependent instructions in context B so that the pipeline stall caused by this dependence is completely tolerated. The context switch cost is also much smaller than is the case with the blocked context switch method, since instructions of only the particular context that causes the cache miss during writeback need be "squashed". In one pipeline-deep phase, as shown in Fig. 7, there are only two cycles which have stages from context A, and only these need be squashed when the cache miss on the writeback occurs, thus resulting in the loss of a maximum of only two cycles as compared to the seven cycle loss for the same cache miss in the blocked context switch case. Similar comparisons can be made for contexts B and C. Both these contexts lose 3 cycles each, still much lesser than the 7 cycle loss suffered by the blocked scheme. Each time a cache miss occurs, the "culprit" context is made unavailable(very much like the blocked scheme in that sense) until finally it is only context D that is left and is interleaved with itself. A significantly lower context switch cost and better pipeline stall tolerance thus make interleaved context switching much the better choice as compared to the blocked scheme.

3.3.3.1 Implementation details of interleaved scheme

Much like the blocked scheme or any multithreading strategy in general, the interleaved scheme too requires each context to have its copy of the program counter, register file and any miscellaneous process-specific state from the processor status word. Only the motivating factor for such a requirement differs: In the interleaved scheme, state replication is a requisite to allow multiple contexts to be simultaneously active in the context window rather than to lower context switch costs as is the case with the blocked scheme.

The instruction issue unit needs to be more sophisticated than in the blocked scheme because of the requirement of having to issue instructions from multiple *active* contexts. Also, sharing a pipeline among multiple contexts means having to keep track of which context issues which instruction stream because in the event of long latency operations, only the instructions of the particular context which is the "perpetrator" of the long latency operation need be squashed.

The other implementation requirements are mechanisms to make contexts unavailable in cases of (i) long latency operations (ii) synchronization latency and (iii) instruction latency.

One transaction buffer is provided per context to keep track of the outstanding request of the corresponding context. In the event of a cache miss, the address of the memory operation which caused the miss is entered in the buffer, all instructions in the pipeline from this context are squashed and further instructions from the same context are prevented from issuing. When the cache miss is serviced, the transaction buffer is marked invalid and the context restarts by re-issuing its memory request.

For the purpose of synchronization latency tolerance, a backoff instruction is supplied to make the context in question unavailable for a period of time specified by the backoff value. Instruction latency problems are much lesser in the interleaved scheme as compared to the blocked scheme due to the temporal spreading apart of instructions from the same context. In addition, the backoff instruction can again be put to good use - the instruction either being manually inserted by the programmer/compiler next to dependent instructions that need stalling or by a hardware scoreboard that keeps track of instruction dependences.

Having thus described the form of multithreading we will use in order to tolerate latency, we shall now go onto the intricacies of our cache coherence protocol used to maintain cache consistency between the distributed caches used in a multiprocessor system; the caches being employed to ease the burden on the latency tolerance techniques used, the interleaved context-switching scheme in our case. We propose the simple minded visible synchronization based protocol which while being not so particular on being perfect with respect to invalidations, saves a lot on network traffic and given an intelligent parallel compiler, is easy to implement. The saving on invalidation traffic also means lesser latency, and naturally more scalability. Scalability being the overriding concern in multiprocessor systems nowadays, we strongly believe that the amalgam of interleaved multithreading and visible synchronization based cache coherency would be just the optimal blend of low latency multiprocessing that is so much in demand in the multiprocessing systems of today.

Chapter 4

A software managed cache coherence protocol

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth."

Sir Arthur Conan Doyle.

"The beauty of truth lies in its elegance and simplicity."

Anonymous.

4.1 Underlying principles

Hardware-implemented cache coherence protocols insist that runtime information be kept about which cached copies of variables need to be invalidated(caches are largely invisible to the compiler) – a centralized directory controller records which processors have copies of a location, and always invalidates an outstanding writeable copy when another request arrives. We, however, rely solely on user directives and machine instructions inserted by the compiler to coordinate changes in the writeability of shared data; the pertinent point being that our software-managed protocol does *not* allow copies of concurrently writeable data. We follow the dictum: If some other parallel activity might write it, do not copy it. In a properly synchronized (data-race-free) program with visible synchronization, one never needs explicitly to keep track of who has copies of which data. This is done *only* at compile time, and usually only implicitly. Coherence actions are thus, a part of generated code rather than a result of a hardware implemented protocol.

As mentioned brifely in the introduction to this thesis, the idea to let visible synchronization direct changes in writeability of shared data stems from the following fact which is a general feature of most already existing protocols. Non-adaptive hardware-based protocols caused each shared variable write-access to be preceded by a write-invalidate protocol. Some adaptive hardware protocols did lessen this superfluity[Pro94] by identifying producer/consumer relations with more clarity with respect to 'write' regions. But, the redundancy of having a distinct write-invalidate protocol that resulted in coherence traffic on top of the already present synchronization traffic still remained. The significant point being the fact that synchronization in traditional cache coherence protocols is not used to direct changes in writeability; instead of which, it merely guarantees that invalidations entail no harm acting only as guards of critical regions rather than actual perpetrators of invalidating action. In our protocol, such redundancy is done away with by letting synchronization trigger *local* invalidation, eliminating non-local coherence traffic and the need for a separate coherence protocol in the process.

Thus, to sum up, the objectives of our protocol are:(i) to ensure that there are *never* any copies of concurrently writeable data, and (ii)to make sure *all* coherence actions are local. As means to this end, we do the following: User directives and compiler analysis let visible synchronization tell the runtime system when to load copies of memory locations into caches and when to invalidate these copies. This makes sure that condition(i) is met. Since all invalidation actions are *local*, (ii) is satisfied as well. In the following section, we describe our protocol in greater detail.

4.2 What are our problems?

Two significant problems in large-scale multiprocessors are: (i) Memory bandwidth – there is too little of it and, (ii) Memory latency – there is too much of it. As solutions to these problems, the following views have been aired: True believers in caches are obsessed with locality and true believers in multithreading normally stay away from both caches and locality. We on the other hand opt for a "middle" way. In our approach, caches are not the principal solution to either problem. We believe that memory bandwidth problems are solved by high bandwidth networks and concurrent memory systems, and that memory latency problems are solved by switch-on-every-cycle multithreading. The philosophy we adopt is that caches ease the strain on latency tolerance mechanisms. Therefore, they are allowed to fail in terms of more misses or unlucky invalidations some of the time. One result is that our protcol is simple.

The conventional wisdom is that cache-coherence protocols *must* be implemented in hardware. If we allow shared writeable data, this may well bye true. If however, we decide to disallow shared writeable data, this conventional assumption becomes optional. That is to say, if there are no shared and writeable data, we are not *required* to implement in hardware, and we may choose instead to experiment with a compiler-implemented cache coherence protocol.

4.3 Assumptions

As the first step in the formulation of our protocol we assume that the programmer and compiler have produced a data-race-free program in which synchronization has resolved all dependence conflicts. The correctness of the protocol depends on this. Compiler researchers have long investigated the possibility of automatic detection of data race anomalies, but this area is not yet fully mature. Further, we assume that the compiler detects state transitions by looking at visible synchronization and the code itself. In particular, all coherence actions are compiler-implemented and appear in the generated object code, i.e. there are no run-time decisions.

4.4 Visible synchronization based protocol

The central theme of our protocol is to copy after input synchronization points and to uncopy before output synchronization points[Fig. 8] (essentially, to avoid having concurrently writeable copies of the same datum) with variations based on the type of state transition the variable in question undergoes. The compiler "knows" of the state transitions by a combination of compiler techniques like dependence analysis and alias analysis and user directives. The base protocol thus, is as follows:

(i) After reaching an input synchronization point, the processor may make copies

```
P(s)        //obtain permission to copy $a$
  copy $a$ into cache
  ..
  ..
  self-invalidate $a$ in cache
V(t)        // signal that the copy of $a$ in cache
               has been invalidated
```

Figure 8: Base protocol

in the cache of all shared variables that are protected by the synchronization operation. These copies will be loaded as cached loads and stores occur.

(ii) Before reaching an output synchronization point, the processor discards all copies in the cache of shared variables that are protected by the synchronization operation, and – if there has been a store to x – possibly writes back the last store to x in the block.

Before discussing the modifications to the base protocol, we shall clearly define the cache states that a datum goes through, from a single thread's point of view. Once we do that, we will be in a better position to describe these modifications based on the type of cache state transition that happens.

4.4.1 State definitions

A datum will be in one of the following two cache superstates :(i)cacheable, and (ii) uncacheable. Cacheable means a memory reference to a datum automatically results in a copy from memory to cache. Uncacheable means even if memory references are made, the datum cannot be copied into the cache. The former can be divided into the following substates: (i) private - exclusive read/write, one thread has permission to make a copy, and may modify it. The datum is invisible to all other threads. (ii) shareable - read-only, several threads have permission to make a copy, but not to modify it. Other threads may, of course, request permission to make copies.

If a cache has a copy of a datum, it is thus either (i) shared - the copy of the

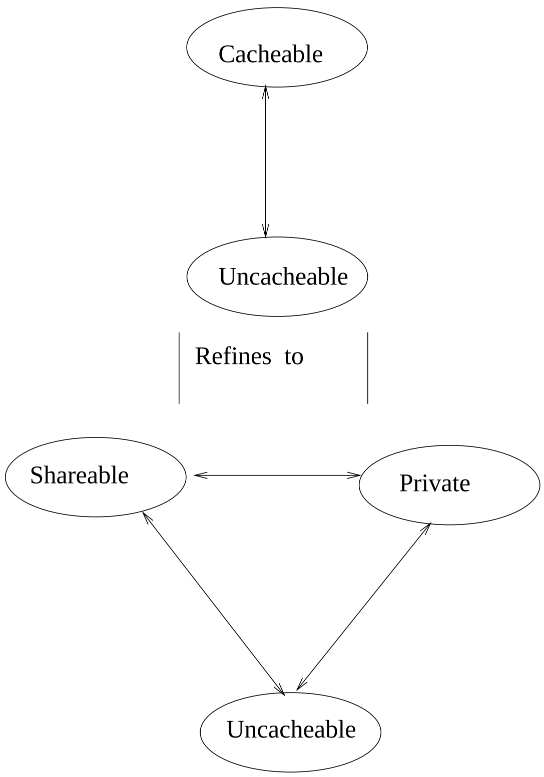

Figure 9: Cache state diagram

datum can only be read or (ii)private- the cache has an exclusive read/write copy.

The one additional machine instruction – aside from cached and uncached loads and stores – we need as part of the instruction set architecture in order to guide the cache correctly through these states and hence to implement cache coherence is an uncopy, to invalidate all copies made during the last synchronization epoch. Thus, an uncopy basically undoes what the corresponding copy did in that synchronization epoch.

There is no need for an explicit copy instruction for the same set of variables since any operating system running a cache when performing cached loads/stores would copy data from memory, in case of misses; the above uncopy would ensure that these data when later referenced would cause misses and hence data to be reloaded from memory. So, whenever we say copy, we mean load from memory.

4.4.2 High-level "economic" administrator

To reiterate, the fundamental idea of our protocol is to forbid the presence of shared and writeable data. We have a high-level protocol based on the idea that: Data cycles between two states: (i) Copyable – a thread has permission to copy a datum somewhere. (ii) Uncopyable – a thread has no persmission to copy that datum.

Based on cost/benefit analysis - which indicates whether caching is economical - this high-level protocol into two: (i) A register-coherence protocol and (ii) A cache-coehrence protocol. More precisely, when a datum is copyable and the cost/benefit analysis about reuse passes a certain threshold, we use the cache-coherence protocol, in which a datum cycles between two states: (i) Cacheable – a thread has permission to do cached loads and stores and, (ii) Uncacheable – we use uncached loads and stores.

If the cost/benefit analysis indicates that the level of reuse has not reached a certain threshold, the register-coherence protocol is applied. A datum now cycles between two states: (i) Copyable – a thread has permission to make copies in registers and (ii) Uncopyable – no permission to make copies in registers.

4.4.3 State transitions

1) Uncacheable to private: Once past input synchronization, the thread has permission to copy from memory to its cache. And since the variable was uncacheable with respect to the thread's cache before the input synchronization, a copy from memory to cache is executed after input synchronization.

2) Private to uncacheable: When a variable goes from being private to a thread to being uncacheable with respect to it, an uncopy instruction is inserted by the compiler to self-invalidate the thread's copy, and a writeback is executed to write back the up-to-date value to memory.

3) Shareable to private: No action need be carried out, since what was shared and read-only becomes private to the thread, which means if the thread had a copy before input synchronization, it's up-to-date.

4) Private to shareable: What was private – in exclusive read/write mode – to the thread, becomes shareable and read-only after output synchronization. Thus, memory needs to be made up-to-date by executing a writeback.

5) Shareable to uncacheable: What was shareable, that is, if the thread had a copy in its cache, it'd have been up-to-date, becomes uncacheable. Therefore, the copy in the cache has to be invalidated for which an uncopy instruction is inserted.

6) Uncacheable to shareable: Before the transition, the variable could not be copied into the thread's cache, hence, when it becomes shareable, the thread can request for a copy of the value in memory. Hence, a copy is executed.

It is to be noted that synchronization variables are uncacheable always. They may only be brought as high as nearby memory modules as they are shared and writeable always. Variables of type future, synch and volatile can be declared in parallel programs[D. 90a]. None of these variables are *ever* cacheable. In other words, synchronized data accesses may be cacheable at certain times, but asynchronous data accesses are never cacheable.

Until now, we were discussing state transitions with just one thread under consideration. It is more useful however to consider the "real-world" case of transitions with several threads. We give sample fragments of code with two threads each, when the transition in one thread has to be coordinated with that in another; coordinated transitions with two threads and copies of a single variable serve to give a clear idea of state transition in a multiprocessor system with a large number of threads and is

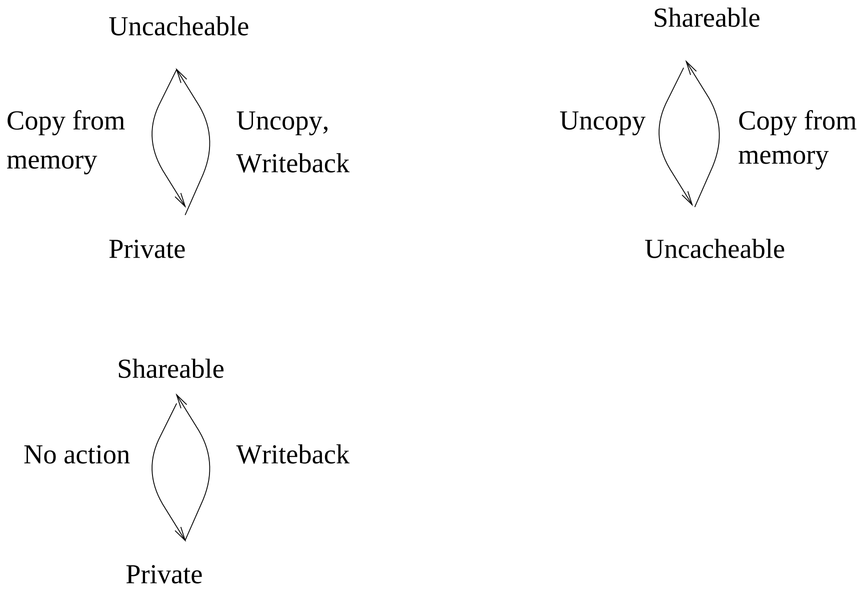

Figure 10: Cache state transitions

representative of the bigger picture.

4.4.4 Illustrations of coordinated state transitions

In the examples below, shared(me) means thread 1 and 2 have copies of the datum in the shared state, that is, copies in read-only state. The "me" in all cases refers to thread 1. nP may be thought of as a sequence of n fetch-and-decrement operations on an atomic integer variable. Shared(not me) means thread 1's cache does not have a copy, and thread 2's cache has a copy in the shared state, that is, in read-only state. Private(me) denotes thread 1's cache having a copy in the private state, that is, it has an exclusive read/write copy. Private(not me) means thread 1's cache does not have a copy, and thread 2's cache has a copy in the private state - an exclusive read/write copy.

Shared(not me) to private(me)

Before the transition, a set of threads other than thread 1 has copies of A in read-only mode, and after it, thread 1 has an exclusive read/write copy of A. The user adds an nP as shown in the figure below so that thread 1 waits for all n threads

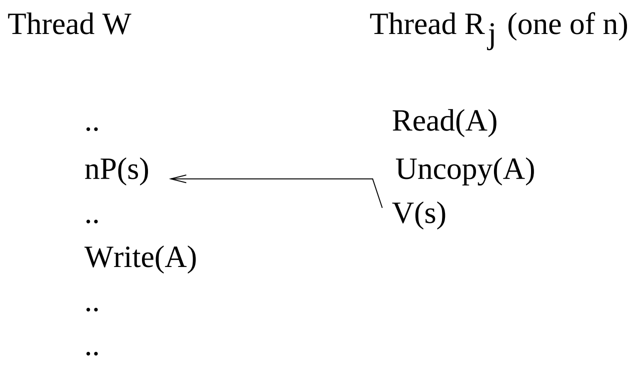

Figure 11: Shared(not me) to private(me)

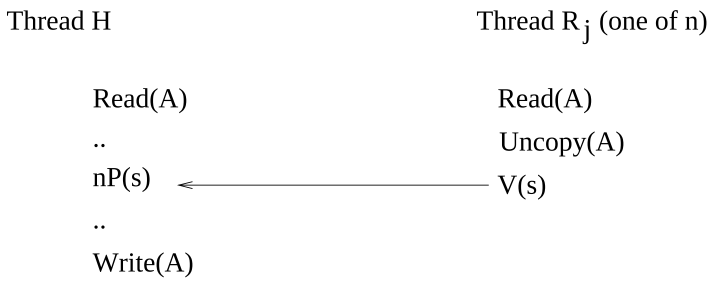

Figure 12: Shared(me) to private(me)

read-sharing variable A to uncopy.nP may be thought of as a sequence of n fetch-and-decrement operations on an atomic integer variable. And after thread 1 goes through the P, the write reference to A causes a miss and load of A from memory to get the up-to-date value. Thread 2 is representative of a set of threads which have A in read-only state.

Shared(me) to private(me)

Shared(me) means a set of threads including thread 1 has A in read-only mode, and after the transition, thread 1 will have an exclusive read/write copy of A. In this case, since thread 1 already has an up-to-date copy of A, all that has to be done now, is to wait for the other threads that have cached copies to execute uncopies, which is what the nP and the V together effect.

Private(me) to shared(me)

Thread 1 has an exclusive read/write copy of A, and after the transition, a set of

Thread W	Thread R_j (one of n)
P(s)	..
Write(A)	..
Writeback(A)	..
nV(s) $\longrightarrow$	P(s)
Read(A)	..
..	Read(A)

Figure 13: Private(me) to shared(me)

threads including thread 1 will have copies of A in read-only mode. Here, the compiler inserts a write-back. An nV ensures that each of the P's allows the respective thread to go through. And, the other threads seeing a P, execute a fresh load of the variable they require from memory.

Shared(not me) to private(not me)

This is a trivial case, no statements are inserted.

Shared(me) to private(not me)

A set of threads inclusive of thread 1 has copies of A in read-only mode, and the transition results in a thread other than thread 1 having an exclusive read/write copy of A(in this case, thread 2). Transition 1 is reversed, in the sense that thread 1 takes the place of thread 2 and vice versa. Thread 1 and other threads execute a V primitive and since the programmer would have already inserted an nP before the private(not me) epoch which is the private(me) epoch of thread 2, thread 2 would wait for all n threads to uncopy, before it does a load from memory.

Private(not me) to private(me)

Before the transition, a thread other than thread 1(thread 2 in this case) has an exclusive read/write copy of A and after it, thread 1 has an exclusive read/write copy.

Therefore, a load from memory is carried out and A is copied in. And supposing A was in the private(me) state in thread 2, a uncopy with writeback is inserted before the V that the programmer would have inserted to protect the A in thread 2.

Private(me) to private(not me)

Thread 1 has an exclusive read/write copy of A, and after the transition, thread 2

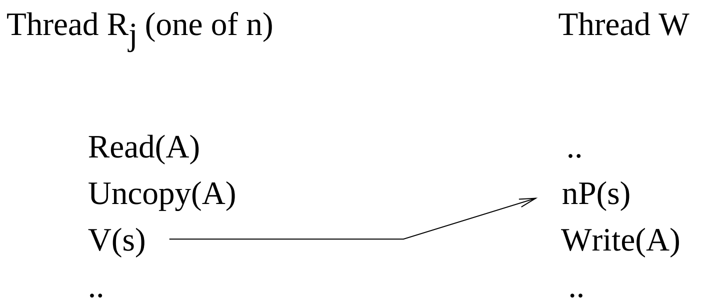

Figure 14: Shared(me) to private(not me)

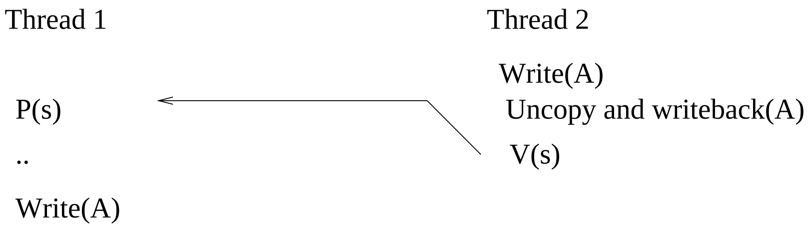

Figure 15: Private(not me) to private(me)

Thread 1		Thread 2
P(s)		..
..		..
Write(A)		..
Uncopy and writeback(A)		..
V(s)	$\longrightarrow$	P(s)
..		Write(A)

Figure 16: Private(me) to private(not me)

has an exclusive read/write copy. An uncopy with writeback is inserted and executed. Thread 2 meanwhile would execute a load of A from memory after the P that marks the beginning of its read/write region.

4.4.5 Performance advantages

The uncopy instruction is implemented in hardware by tagging each variable that is copied into the cache from memory during a particular synchronization epoch with a "colour", the colour being indicative of which synchronization epoch the variable was loaded into the cache, so as to identify them at the time of uncopy. Without tags, the cost of the uncopy goes up linearly with the size of the data to be uncopied. Before output synchronization, a parallel search is done for tags of the particular colour, and the cache entries with that particular tag are invalidated. Thus, with "coloured" uncopies we eliminate the time penalty that sequential invalidation incurs.

Cache coherence protocols lose their meaning if they do not scale, i.e. if they lose too much processing speed as problem and machine sizes increase. Broadly, scalability is achieved by two means: (i) by tolerating unpredictable fine-grained latency from any source and (ii) by providing long-range communication bandwidth in the interconnection network. Focussing our energies on the first technique, we note that that latency tolerance is comparitively non-traditional as compared to latency avoidance which is essentially, caching. A large amount of coherence traffic causes high latency. Thus, our coherence protocol highlighted by its almost complete lack of non-local coherence traffic supplemented by interleaved multithreading is indeed very much justified. This along with its inherent elegance and simplicity - it relies solely on visible synchronization and user/compiler directives - and the not inconsiderable plus that it supports the protects consistency memory model[Pro93b] - which eliminates some unnecessary $data- > sync$ precedence arcs that the release consistency memory paradigm enforces and limits the scope of synchronization by a clearer definition of which set of variables each synchronization variable protects (visible synchronization) - make it all the more attractive.

Compiler analysis techniques like alias analaysis, dependence analysis and inter-procedural analysis along with PRC programming notation help direct the cache through the aforementioned state transitions. We shall illustrate PRC programming notation with help of sample code in the next chapter. It is to be noted here that our protocol would work even without having PRC as our programming model; any set of user/compiler directives telling it which variables to uncopy at the end of a synchronization epoch would do and it works with release consistency as well, but with it being linked to PRC, there is scope for reusability of cached copies and more parallelism to be exploited by the hardware since PRC is a less restrictive memory model. Thus, the selective invalidation brought about by the simple but effective technique of visible synchronization answers the main criticism levelled against software-based invalidation(cache coherence), that of being too conservative in its choice of invalidation, rather efficiently. Each time you cross a synchronization point, you are effectively making a state transition and from user directives(and PRC programming notation) and compiler analysis the variables undergoing the transitions are clearly marked(the role that PRC plays shall be dealt with in the next chapter on the protect consistency model and its primitives) and this is in essence what precludes indiscriminate invalidation. What Cheong and Veidenbaum do by means of compile-time memory reference tagging and the change bit, we achieve by the elegant use of visible synchronization. Moreover, our technique is closely linked to a more relaxed memory consistency model.

It should be noted that in transitions involving loads and uncopies , if we use the release consistency memory model, there will be additional load and uncopy overhead. In the case of uncopies, all variables accessed (read/written) within a particular synchronization block would be invalidated - since synchronization variables are not linked to the copies of memory locations that they protect in RC, they're assumed to protect every variable accessed in the epoch - whereas if protect consistency primitives were to be implemented, only the requisite variables which are guarded by the particular synchronization primitive would be invalidated. Similarly, in the case of copies(loads), in RC, *all* cached copies that are accessed within a synchronization interval would have to be copied from memory, because of the previous uncopy, whereas PRC which connects the copies to the synchronization variables in a more direct way(as we shall see in the next chapter), would permit more reuse of cached copies. More details on release consistency and protect consistency can be found in chapter 5.

There is no room for "false sharing" - the line size becoming larger than one leading to loading of words into the cache which dependence analysis can not detect - since we insist on uniword cache blocks. The Cheong-Veidenbaum protocol does find a solution to this problem of false-sharing, but has to resort to additional software or hardware and version numbers in a later version of the protocol we described in chapter 2. The main justification in insisting on uniword blocks, other than the fact they eliminate false-sharing is that "working-sets" are generally small and multiword cache lines cause attenuation in bandwidth. Memory references in a multiprocessor system tend to be spread out among the processors which reduces the available spatial locality compared to a uniprocessor system. Thus, although it might be in fact be possible to lessen the miss ratio by using multiword blocks, simulation studies[D.J93] hint that single word blocks minimize network(miss service traffic in the main, in our case) traffic.

We recommend the use of physical caches over virtual ones, because although there is the additional overhead of virtual-to-physical address translation before the cache entry is located, there is the absence of the need to flush the cache on every context switch. But, our protocol does not need to have additional support in case of context switches; the writeback on miss being part of the basic cache protocol in any multiprocessor system with caches.

As in any new proposition, there always is the question why a protocol, especially one that is laced with very little complexity as ours wasn't thought of before(We regard reasoning of this genre a significant step in developing new trains of thought). It was in the main, we feel, due to the concept that synchronization and invalidation in a coherence protocol were discrete procedures, but the elimination of this redundance leads us to a very time-efficient(because there is no indiscriminate invalidation

```
proc deposit (m: object)
  begin
    P(slot)
      P(sender)
        front := front + 1 mod n
        BUF[front] := m
      V(sender)
    V(message)
  end
```

```
proc remove(var m: object)
  begin
    P(message)
      P(receiver)
          rear := rear +1  mod n
        m := BUF[rear]
      V(receiver)
    V(slot)
  end
```

Figure 17: A producer-consumer problem

and we use switch-on-every-cycle multithreading), simple-minded scheme. Since most parallel programs are composed of cooperating threads with far more synchronization constraint than a set of independent non-cooperating ones, we expect compiler analysis and user directives to successfully guide the caches through the afore-mentioned transition states.

To show the performance advantages of our protocol, we consider the following two examples.

4.4.5.1 A producer-consumer example

In a hardware-implemented protocol, synchronization and invalidation are two separate activities. This is redundant given that visible synchronization can be used to directly coordinate the writeability of shared data. The cached copies of the shared variables front, rear and BUF would be repeatedly invalidated. Superfluous coherence traffic would result. In contrast, our cost/benefit analyzer would see the non-reuse. Therefore, these variables are *economically* uncacheable. The register-coherence protocol would be applied as the variables cycle between "copyable" and "uncopyable".

4.4.5.2 Writer, N Readers problem

Our protocol applies in the following way: The writer writes back the up-to-date

```
Thread W              Thread R1          Thread R2        ...

repeat
for(...i++)           ...                ...
  Write(a[i]);        ...                ...
Writeback(a[i]);      ...                ...
barrier(...);         barrier(...);      barrier(...);
for(...i++)           for(...i++)        for(...i++)
  Read(a[i]);           Read(a[i]);        Read(a[i]);
...                   Uncopy;            Uncopy;
barrier(...);         barrier(...);      barrier(...);
until(...);
```

Figure 18: Writer, N Readers problem

values to memory, and then signals the reader threads so that all threads go into the read epoch together. Before the writer returns to the write epoch, it waits for the reader threads to invalidate their copies of the array. As in the previous example, there is no non-local coherence traffic. Using hardware-implemented protocols would have caused heavy coherence traffic whose volume was linear in the size of the array.

4.5 Comparison with existing protocols

As stated before, already existing cache coherence protocols can be broadly classified into two: Hardware-based ones and the relatively recent software based ones. In the comparisons with our protocol below, we shall highlight the performance advantages that our protocol exhibits with respect to these existing ones.

4.5.1 DASH

Hardware-based protocols like the DASH suffer from too much space being wasted for directory information and have the additional liability that too much of communication bandwidth is wasted on retrieval of directory information from the distributed directory, albeit incurring a lesser overhead compared to a centralized one. Directory lookups are required at the time of writes and invalidations need to be carried out according to state information obtained from the directory. Lookups are required at the time of reads as well, and state updates need to be performed. All this results in high coherence traffic. The reasons for this are (i) non-local invalidation and (ii) the allowing of concurrently writeable copies - the directory controller records which processors have copies of a location, and always invalidates an outstanding writeable copy when another request arrives.

In our protocol, there is no need to keep track of copies of a shared location since concurrently writeable copies of a location are *not* allowed. *Local* invalidations are performed according to the type of state transition the variable undergoes and copies are not made if some other parallel activity might write it - input synchronization acts as a request to permit making copies and output synchronization acts as invalidator, eliminating in the process, the redundancy of having a separate invalidate protocol aside from having a synchronization one. Thus, there is almost total lack of coherence traffic, in our protocol compared to DASH, in which there is both synchronization and coherence traffic.

DASH relies on the release consistency memory model. Our protocol can function with the release consistency model as its memory access map. But, more parallelism can be exploited if it is keyed to the protect consistency model as will be evident in the next chapter.

4.5.2 The one-time identifier scheme

Existing conventional software-based protocols have a proclivity toward conservatism in their approach to invalidation of cached copies of data. More copies than required are discarded because if this "safety-first" approach. The OTI (one-time identifier) approach[A.J85] which was among the first few efforts in this direction tried out the idea of having an identifier for every cache and TLB (translation lookaside buffer) entry as a pointer to the usability of the cached copy of a datum. For every new TLB entry, a new OTI value is generated and when the copy of the shared datum is loaded into a cache, the OTI value of the cache line is equated to the OTI value of the corresponding TLB entry. The cache-hit defining condition requires both these OTI values to be the same.

Upon encountering a write-back shared region, a TLB invalidate instruction is issued to every page in the region. This ensures that subsequent accesses to that page will cause a new OTI value to be loaded along with a new TLB entry. The difference in OTI values with the corresponding cache results in a cache miss.

The obvious disadvantages are: (i) the possibility of wasteful invalidation when fresh TLB entries are made as a result of data replacement rather than data invalidation for which it was originally intended. (ii) sequential invalidation of page entries in a write-shared region and (iii) memory overhead due to the additional space taken up by the OTI entry for every cache line.

As must be clear from the state transition illustrations when we discussed our protocol, there is no indiscriminate invalidation in ours. Only copies of variables that are protected by a particular synchronization epoch are invalidated. And, that too, only wherever necessary. There is no sequential invalidation the avoidance of which is the aim of the uncopy instruction, which does a parallel search for the tag that identifies the particular synchronization interval and invalidates all copies having this tag. Cache lines are short as there is no need for state storage in them, therefore, there is no extra memory overhead.

4.5.3 The Cheong-Veidenbaum protocol

Problems of wasteful invalidations occur in the case of indiscriminate invalidation protocols[A.V86] [R.L87b] [K.K87]. Thus, in essence the key issues associated with software based invalidation are: (1) when to invalidate and (2) what to invalidate. The Cheong-Veidenbaum fast selective invalidation protocol discussed earlier answers both questions to a reasonable extent and resembles our visible synchronization protocol in that both involve selective invalidation of cached copies of variables, thus countering a strong criticism levelled against software-based cache coherence schemes - that the invalidation process was based on extreme "conservatism" and hence resulted in unnecessary invalidations of valid cached copies of data.

The main difference between the Cheong-Veidenbaum protocol and ours is that ours is based completely on compile-time analysis, and doesn't need a "change" bit to indicate that a copy has been loaded into the cache in the current synchronization interval since our protocol follows the principle: If a valid copy is present in the cache, it can be used - we do not have concurrently writeable copies of variables in a cache. Also, the C-V protocol is more complex than ours, with very close attention paid to maintaining a high hit ratio. Plus, the fact that our protocol can be implemented with a more relaxed consistency model such as the PRC one(as we shall see in the next chapter), increases scope for instruction-level parallelism.

False-sharing is eliminated in later versions of the C-V protocol, by relying on extra software/hardware. But, we insist on uniword cache blocks, justification of which we gave in the previous section, thus allowing no room for false-sharing.

Having discussed our protocol and its strengths at length, we turn our attention now in chapter 5, to the protects consistency model and its primitives which enable visible synchronization to be implemented effectively and the architectural base that it requires.

Chapter 5

The protect consistency model

As mentioned before, directory-based protocols generate unnecessary coherence traffic. Existing compiler-controlled ones on the other hand tend to overinvalidate. Our visible synchronization based scheme however, using the protect consistency(PRC) model as its memory access map, implemented by insertion of user/compiler directives - which perform *local* invalidation and/ or writeback as is the case - answers most criticisms levelled against software-based protocols. The PRC model/ programming notation provides increased flexibility in specifying instruction- level parallelism that can be passed directly to the hardware [Pro93b], [Pro93a], thus increasing concurrency within a thread without incurring appreciable synchronization cost; how it does this and what primitives it uses will be the focus of this chapter. Apart from being a supplement to switch- on-every cycle multithreading in itself, PRC more directly supports our cache coherency protocol by promoting the reuse of cached variables in certain situations, as we shall see shortly, which for instance would not be possible with a more restrictive memory model like the release consistency one, thus countering the over-invalidation criticism to an extent.

Before coming to what actually PRC is and the primitives that help implement it, we will take a look back at the more constraining (parallelism-wise) memory models that are already in existence and trace the gradual progression of the idea of lesser restrictions on program execution order, to its culmination in the PRC model.

5.1 Existing consistency models

Censier and Feautrier[L. 78] define a system with coherent caches to be one which guarantees that "the value returned on a Load instruction is always the value given by the latest Store instruction with the same address". The catch involved is that the term "latest" is very vague when the loads and stores are on different processors running asynchronously with respect to each other. Due to transmission and buffering delays both within the processors themselves and inside the interconnection network, there is non-atomicity as far as memory updates go. That is, not all copies are updated at the same time. Hence, each processor "views" a different time-ordering of events, the events in this case being loads and stores. To make the point clearer, on a uniprocessor, one is certain that the latest load *will* return the value of the latest store, because of the presence of hardware mechanisms called interlocks that check data and control dependencies between instructions. But, due to the aforementioned "observational gap" in multiprocessors, inconsistency among processor "views" is brought about. It is herein, that a consistency model becomes relevant. The consistency model of a multiprocessor system defines the programmer's view of the time-ordering of events that occur on different processors; the events including memory accesses and synchronization operations. This model is in effect what the system "promises" the programmer as far as the execution order goes. Thus, the more the assurances the system makes about event ordering, the less the programmer would need to worry about specifying synchronization operations and in turn, there is much less scope to exploit possible avenues of concurrency. Since the driving force is to obtain more concurrency, a model in which the system ensures as little as possible regarding ordering, demanding thus a more explicit synchronization skeleton from the programmer, which leaves more room for time overlap of operations will prove optimal (since he/she would now program in such a way as to ensure correct results in spite of the overlap). The more specific the synchronization skeleton that the programmer outlines, the less the system has to assume when re-ordering the loads and stores, thus allowing for greater freedom in re-ordering, which translates directly to more instruction level concurrency. It is thus this driving force that has motivated the gradual formulation of less restraining consistency models, as we shall see presently.

5.1.1 Sequential consistency

Lamport[L. 79] proposed the multiprocessor correctness condition, sequential consistency(SC) as follows: the result of any execution is the same as if the operations of all the instruction streams had been executed in some sequential order, and the operations of each instruction stream appear in this sequence in program order - that is, in the same order as the instruction stream would have been executed on a uniprocessor(with interlocks, obviously). The program behaves as if the memory accesses of all the threads are interleaved and executed sequentially, respecting program order. The programmer is thus able to *pretend* that operations including memory accesses are atomic, that is, memory updates are "felt" by all processors at the same instant and use Owicki-Gries style assertional reasoning applying the methods of pre-conditions and post-conditions. For instance, we write: IaJ to mean if assertion I is true before a, then assertion J is true after a. Since atomicity is enforced at the granularity of single word accesses of memory, there is no commitment to a high-level programming model. Hence, there is no exploitable concurrency(since you can not have simultaneous memory accesses) and hence no future in sequential consistency, although its role in the development of more relaxed protocols can not be overstressed.

The significant point here is that since synchronization operations are left unmarked, the system presupposes every memory access to be a synchronization one which leaves no room at all for architectural optimization. In interests of clarity, we define the refinement USC of SC by prefixing the words "even if arbitrary program operations might be synchronization operations" to the statement of SC. Thus, the key is to precisely identify synchronization operations, for the more explicit the synchronization skeleton, the less the system has to assume about synchronization operations, opening up more avenues of exploitable parallelism(in other words, more time overlap of program operations).

5.1.2 Weak consistency

In WC[M. 86][V.R86], the programmer identifies synchronization operations. The implementation conditions are the following:

(i) No processor may issue an ordinary shared-variable access until all synchronization operations that precede the access in program order have completed.

(ii) No processor may issue a synchronizing operation until all ordinary shared-variable accesses that precede the operation in program order have completed.

(iii) Synchronizing operations must be kept sequentially consistent.

In this paradigm, synchronizing operations serve as barriers or fences when a synchronizing operation completes, all previous operations have completed, and no further operation has been issued. Synchronizing operations define points at which consistency among processor views is guaranteed. Its one great plus is that processors are allowed the flexibility of reordering operations between these synchronization points, thus obtaining more parallelism whilst respecting uniprocessor control and data dependences.

5.1.3 Release consistency

Release consistency(RC) further weakens WC's hold on parallelism by splitting synchronizing operations into two, acquire and release, asking the programmer to distinguish between the two. What acquire and release are, will be evident from the following implementation conditions for RC.

(i) No processor may issue an ordinary shared variable access until all acquire operations that precede the access in program order have completed.

(ii) No processor may issue a release operation until all ordinary shared variable accesses that precede the operation in program order have completed.

(iii) Special accesses(i.e. synch operations and nsynchs) must be kept processor consistent.

RC partial order is weaker than WC partial order. This is so because of the deletion of several redundant $data->synch$ and $synch->data$ precedence arcs as explained below[K. 90]. Firstly, ordinary Load and Store accesses(respecting data and control dependences, obviously) need not be delayed for the release operation to complete since the release operation does not purport to have any say in the ordering of operations/accesses following it; it just allows other processors/threads the use of the write-shared region. This is not the case with WC, as all accesses following the "barrier" sync operation have to wait for this synchronizing operation to complete. Similarly, the acquire operation does not have to wait for previous ordinary Load or Store accesses to complete because it just denotes the acquiring of rights to the write-shared region which it and release operation guard. Whereas, the barrier operation in WC has to wait for all preceding accesses to complete before issuing. Thirdly, non-synch accesses like the competing(simultaneous accesses to the same memory location) accesses to read data in chaotic relaxation algorithms do not have to wait for previous accesses nor delay future accesses. Finally, in RC, synchronzing accesses need only be processor consistent and not sequentially consistent as is the case with WC. Thus, we see that RC strictly allows more flexibility and hence more concurrency of operation, thus making WC a *proper augment* of RC.

5.2 Protect consistency

However, it is also the case that there is still the room for improvement, still the presence of unnecessary $data->synch$ accesses due to the unlimited scope of synchronization variables in RC[Pro93b]; specifically there is the absence of knowledge regarding the mapping of variables or memory accesses to synchronization variables. It is therefore presupposed in RC that a synchronization variable protects any access lying within the particular P-V pair - it is thus *program order* that dictates scope - whereas in reality, it is not *necessary* that all operations/accesses inside the P-V pair wait till the P operation completes. There is room for further parallelism and it is this additional scope for concurrency that PRC looks to exploit by letting programming notation specify the scope of synchronization variables. This is implemented by the use of serial and parallel blocks and single assignment variables - variables which are undefined before assignment, but which after assignment become defined and cannot be redefined. How exactly this is done, we shall see shortly. What it does is to delay an ordinary access only for a *designated subset* of P type operations that precede it in program order, thus in effect telling the system architecture which synchronization variable each memory access is *bound* to, offering architectural optimization as much leeway as possible.

Programmer specification of thread partial order is feasible only for critical program fragments[L. 93]. However, the programmer can demarcate the synchronization skeleton in a thread by defining the *protects* relation between synchronization and ordinary operations. This task is facilitated by an invariant-based view of necessary precedence(true dependence). We define operation b in thread t to be dependent on operation a in thread s if operation a forces invariant I to be true in s's view and operation b's correctness necessitates invariant I to be true in t's view. Using the protects notation, we write: (i) $P- > b$ only if P *awaits* I and {I}b, and (ii) $a- > V$ only if V notifies I and a{I}. The solid arrows are elements of the protects relation indicating in effect the scope of synchronzation variables and the invariants inside curly brackets are guards and post-conditions respectively. And since the scope is now clearly defined, we can use synchronization to directly co-ordinate changes in the writability of shared data by parallel activities, as described in the previous chapter.

In PRC[Pro93b], the programmer identifies: (i) P-type synch operations, which can block the executing thread (ii) V-type synch operations which can unblock some other thread and (iii) PV-type synch operations, which can do both. Within each thread, he or she identifies the requisite temporal precedences among synchronizing and ordinary operations. Using the PRC programming notation, he or she declares the necessary (i) $synch- > data$ precedences (ii) $data- > synch$ precedences and (iii) $synch- > synch$ precedences. It is to be stressed again that these precedences are *not* defined by program order, but by programming notation. The compiler is entrusted with the task of adding the necessary $data- > data$ precedences, completing the partial order of the thread. We state implementation conditions for PRC. (i) No processor may issue an arbitrary shared-variable access until all P-type operations protecting the access - that is, precede the access in protects partial order - have completed. (ii) No processor may issue a V-type operation until all ordinary shared-variable accesses protected by the operation - that is, that precede the operation in protects partial order - have completed. (iii) No processor may issue a PV-type operation until all ordinary antecedent shared-variable accesses have completed. No processor may issue an arbitrary descendant shared-variable access until the PV-type operation has completed. "Antecedent" and "descendant" are defined by protects partial order.

And, obviously, PRC assumes that uniprocessor control and data dependences are respected.

5.2.1 The PRC programming notation

Since in PRC, it is the programming notation that defines partial order which in turn dictates concurrency, its choice is of utmost significance. The notation should allow us to specify $synch- > data$ precedences to the compiler which checks up on data and control precedences. Expressly with this motive in mind, we adopt a slight modification of the CC++ language[K. 93] . More precisely, we add visible synchronization to the language (i.e., P and V), as well as a scheduling directive for parallel blocks. A sequential C++ program can be converted into a parallel program using new constructs provided in CC++. Among these are single assignment variables, parallel blocks, parallel for-loops, spawn and atomic functions. Apart from adding visible synchronization, we modify CC++ by introducing the thread construct(the scheduling directive mentioned earlier) and adopting semantics for single assignment variables that are *private* to a thread.

Parallel blocks have the semantics of a cobegin...end statement, so the order in which components appear within a parallel block is immaterial. The single assignment variables in CC++ can be assigned a value at most once during execution. Before assignment, they are *undefined*; after assignment, they become *defined*. A parallel activity is suspended if it reads an undefined single assignment variable. It is resumed as soon as the variable becomes defined. In CC++, these variables are write-once, read-many synchronization variables with data types.

The thread construct in PRC is a scheduling directive that allows us to express concurrency within a thread. The defining property of a thread is that it must be scheduled on precisely one processor. Single assignment variables when used within the scope of a thread become private to that thread only. A thread may write a private single assignment variable at most once, but read it many times. Each private sing variable v is of type void. It is written by the command v! which is attached to a single instruction, and read by the command v? which appears in a guarded command for a single instruction.

5.2.2 Programming aspects

RC has default scopes of synchronization primitives set by program order that may be arbitrarily large. PRC on the other hand, lets programming notation specify these scopes explicitly using either serial or parallel blocks, augmented by the prudent use of single assignment variables. The scope of a synchronization primitive is the most deeply nested serial block in which it appears. In the following example with parallel blocks, by the scope rules of PRC, P(s) protects a, b and c but none of d, e and f.

PRC uses single assignment variables whose lexical scope is one thread to express

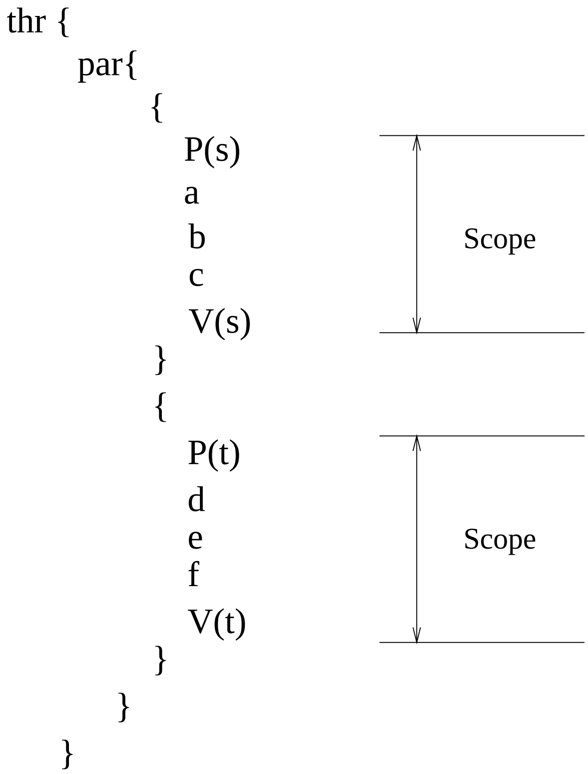

Figure 19: PRC - a parallel blocks example

```
P(s)
x := f(a)
P(t)
y := g(b)
z := h(a,y)
V(t)
```

Figure 20: PRC - another example

point-to-point necessary precedences between pairs of instructions. These variables - they are private to a thread(preventing interference from another thread) and are not genuine synchronization variables, but tagged registers in a single processor - thus, enable the PRC model to exploit instruction-level parallelism within a thread, without incurring appreciable synchronization cost.

The following simple thread fragment(Fig. 17) brings out the performance differences between RC and PRC. In RC, default scope decided by program order makes P(s) protect subsequent ordinary accesses *and* P(t). But, suppose the intended order is: P(s) protects a and nothing else; P(t) protects b and nothing else. V(t) protects z.

We rewrite this fragment in PRC programming notation[Fig. 18], making the instruction-level parallelism more explicit. In PRC, the P operations are executed in parallel. The assignments to x and y wait for *their* P guards only, unlike in RC where the scope of the P operations is unlimited, forcing the second P operation to wait for the first one to finish execution. Also, in PRC, the assignment to z waits for one P guard and one data dependence. In RC, at most P(t) and the assignment to x would be overlapped, all the other instructions would be serialized.

The same fragment of code permits the reuse of cached data in situations where RC would have forced data to be reloaded. To revisit our protocol briefly, we recall that the key to it lay in the fact that it would not cause unnecessary invalidations(as far as possible) and this was because the invalidation(uncopy) instructions inserted by the user/compiler "knew" exactly which variables to uncopy before synchronization and hence copy after synchronization. This is implemented with as less redundant

```
thr{
     sing v                          * private *
       par{
             {
             P(s)
             x := f(a)  > v!
             }
             {
             P(t)
             y := g(b)
             v? --> z := h(a,y)    * If v is defined *
             }
          }
    }
```

Figure 21: PRC - an implementation with single-assignment variables

invalidations as possible by PRC primitives as we shall see presently. In the sample code above, as mentioned before, condition 1 of RC forces a to be reloaded after P(t). In PRC, the sing precedence from P(s) to the second load of a proves that P(t) does not protect a, allowing reuse of the cached value of a. And, hence there is no uncopy of "a" before V(s) and no copy of it after P(t). We could modify RC by checking for synchronization chains that leave a thread between 2 successive loads of the same variable. In this fragment, there are none. However, if V(s) immediately follows the assignment to x, this optimization can not be used. Thus in RC, there is no means to prove that P(t) does not guard a. In PRC, the sing precedence still allows the cached value of a to be reused, showing a latency avoidance difference between RC and PRC.

In the piece of code below[Fig. 20], the use of the single assignment variable m permits the cached value of z to be reused since P(u) does not protect z and the compiler and the run-time system know this. The compiler does not insert an uncopy

```
thr{
      sing v                              * private *
        par{
              {
               P(s) > v!                  * P(s) then v! *
               x := f(a)
                V(s)
               }
              {
               P(t)
               y := g(b)
               v? --> z := h(a,y)     * If v is defined *
                V(t)
               }
          }
    }
```

Figure 22: Final PRC code for the fragment in Fig 16

```
thr{
  sing m
        par{
            {
              P(t)
              z := h(a,y)    > m!
              V(t)
              P(u)
              {
                j := f(r)
               m? --> n :=h(z,r)
               }
               V(u)
             }
            }
    }
```

Figure 23: PRC - another single-assignment variable example

instruction before the V(t) because of the presence of the single assignment variable. This is the private(me)-to-shared(me) transition indicated in the previous chapter.

Thus, we see that PRC promotes reuse of cached variables in certain situations which RC would not have permitted. PRC hence plays a very significant support role to our protocol in terms of latency avoidance with minimum unnecessary invalidation. The coloured uncopy instructions with/without writeback required by our visible synchronization based protocol are thus fully implemented by means of the PRC synchronization primitives(the copy/uncopy instructions are inserted depending on the presence of the single assignment variables).

5.2.3 Compiling aspects

Lack of clarity in expression of $data->synch$ and $synch->data$ precedences are the bane of consistency models like RC where synchronization scopes are dictated by program order. Program order tends to obfuscate these precedence arcs which are in fact known to the programmer. This enforces unnecessary constraints on instruction-level parallelism. PRC seeks to eliminate these redundant constraints on concurrency by *generalizing* the notion of program order slightly, allowing both (i) specification of the scope of synchronization primitives, and (ii) specification of point-to-point precedences within a thread without giving up the benefits of program order vis-a-vis dependence analysis(flat program order facilitates compiler analysis of data and control dependences, obviating this task for the programmer).

Consider the parallel block in Fig. 16. Assume it appears between two serial blocks. It is intuitively clear how an optimizing compiler would analyze this parallel program. The four blocks have a simple lattice structure. The compiler does dependence analysis (i) from the initial serial block to each of the components of the parallel blocks, and (ii) from each of the components of the parallel block to the final serial block. Sing variables preclude interference from writes by competing subthreads. Conceptually, this is scarcely more difficult than dependence analysis of flat program order. Thus, we justify our assumption that compiler analysis in order to implement PRC and our visible synchronization based protocol is indeed a very viable prospect, thus easing fears that this would be a stumbling block to our coherence scheme.

5.2.4 Architectural support for PRC

Convention dictates that compilers analyze parallelism at compile-time and reserializes code, for the hardware to re-analyze parallelism(without violating the consistency model). It would be more direct however, if the parallelism is encoded somehow at compile-time for the hardware to decode it at run-time. A completely unrestricted partial order would entail more encoding; this has practical limits. Each instruction i is part of a partially ordered instruction stream, and has a finite number of immediate *predecessor* instructions upon which it is dependent. Static or dynamic issue order pattern is dictated by the completion of the dependent instructions as follows: Each memory reference upon which an instruction i depends is issued in a non-blocking manner and is identified by a tag, that returns acknowledgement messages to the right address upon completion. When all acknowledgements have been received – the number of predecessors being known, statically or dynamically – a hardware join operation allows \$i\$ to continue.

A serial encoding of partial order would impose restrictions on thread parallelism that is visible to the processor. As an alternative, Tera's compiler prescribes an instruction stream with a 3-bit lookahead field, which affords a dependency lookahead window of size 8[R. 90]. Thus, a new instruction waits for the completion of all instructions that have a lookahead to it. At most, eight instructions from a stream maybe executing concurrently, since the code window size is 8. A hardware instruction waits thus for a *bounded* number of memory references.

Every thread has an eight element lock counter array. One component of this array (say, lock[tag]) is dynamically bound to each instruction instance, which may need to wait for several predecessor instructions to complete execution. When a predecessor issues, the lock counter is incremented and when it completes, is decremented. The instruction instance bound to this tag might issue only when the lock counter has reached zero. In the case of superscalar processors, instructions whose lock counter is zero can be issued concurrently.

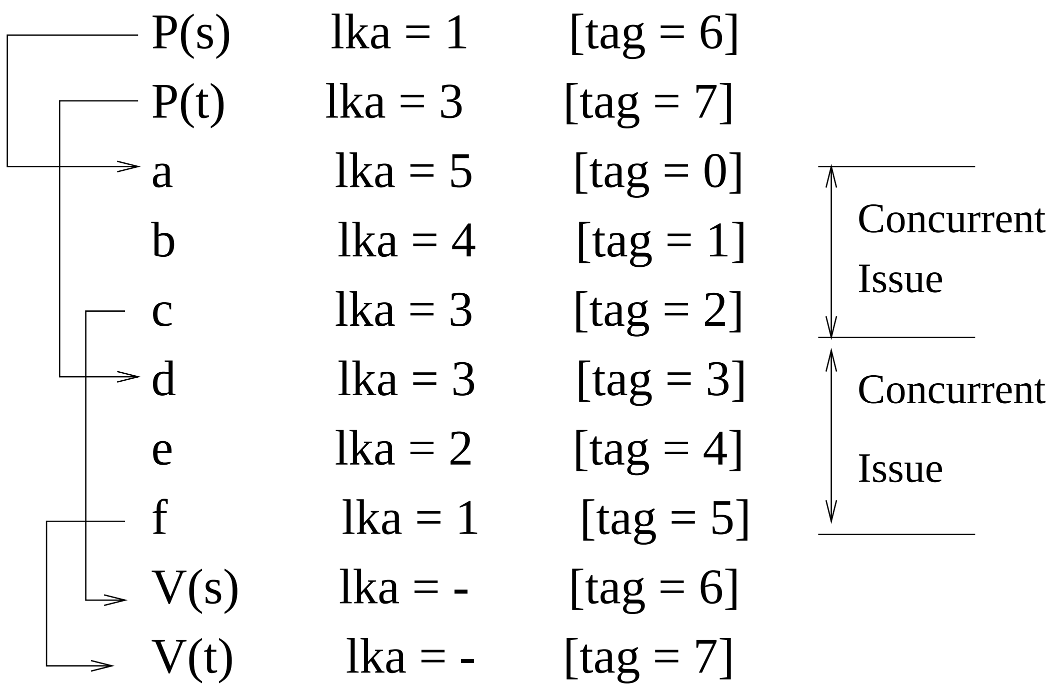

Figure 24: Tagged dependency lookahead

The above is the sample compiled code for the PRC program in the first PRC example in Fig. 16, assuming no data dependences and static issue order. Brackets enclose information available only at run-time. We show the dynamic execution. Every instruction has a tag in 0..7 by the time it issues, following tag[next] = tag[current] + 1(mod 8). When it issues, P(s) increments the lock[tag] counter of "a", viz. lock[0], and P(t) increments that of "d", lock[3]. The whole instruction stream waits for P(s) to complete, in other words for lock[0] to go down to zero. Once this happens, a,b and c are issued concurrently, assuming a superscalar processor. Each increments lock[6] by one, as they issue. Thus, V(s) waits for all three to finish, and assuming static issue order, it has to wait for f to issue. Similarly d has to wait for c to issue and P(t) to complete, once this takes place, all three of d, e and f issue concurrently. Admittedly some parallelism has been lost in this encoding, since P(t) might complete first.

When the processor is not superscalar, issue order is dictated largely by compiler scheduling. For processors that block on read accesses, write latency can still be effectively tolerated as long as write buffer and read caches permit reads to be serviced while writes are still pending. [K. 90]. Tolerating read latency however, for processors that don't block on reads, is considerably more difficult, since computations that follow reads, generally require the read value to proceed. Hence, unless there is very sophisticated compiler scheduling, read latency is tougher to get around.

We now sketch the architectural base for software-binding register prefetch, which is an effective latency tolerance tool. Every processor register has a *reserved* bit and a *full/empty* bit. A load instruction specifies the remote address and response tag. The tag belongs to register name space, and identifies a destination register(say, r) which is both reserved and set to empty when the load issues. Switch-on- every-cycle multithreading minimizes the need for slack instructions to be placed between this load and the first use of this value, as well as the length of time the destination register r is out of service(This is one of the main advantages in employing switch-on-every-cycle multithreading, instructions from the same context are spread apart in time). When a subsequent register operation refers to r, the thread will trap if r is empty.

Using the code fragment below, we illustrate software-managed binding register prefetching. The register fields of the fetch are (i) an address register with the remote address and (ii) a destination register that is allocated until the reference completes. The fetch instruction is non-blocking. The compiler must find unrelated computation to tolerate the expected latency of the load. This task is made easy by a high degree of intra-thread parallelism, so that non-blocking operations that can incur latency can cover for their latency by switching contexts.

```
fetch  r_a, r                \\ r_ now reserved
instruction                  \\ slack instructions
instruction
...
instruction
reg_op  r_, ...              \\ traps if r_ empty
```

Figure 25: Software-managed binding register prefetching

Writes may also be viewed as split-phase transactions, that is, the issue and the completion of the instruction occur in two phases. A store instruction specifies remote address, value to write, and response tag. The tag belongs to *join* name space. which maybe quite small. Suppose multiple writes to a locked object are outstanding. As each write completes, an initialized join counter decrements. When the counter reaches zero, a release operation for the object issues.[G. 93]. A closely related approach is based on explicit language-level split-phase remote memory operations[T. 92].

5.3 Relation to consistency model

The release consistency model states that:

(i) No processor may issue an ordinary shared variable access until all acquire operations that precede the access in program order have completed.

(ii) No processor may issue a release operation until all ordinary shared variable accesses that precede the operation in program order have completed.

(iii) Special accesses like synchronizing operations need to be processor consistent.

In our(base) protocol:

(i) No processor may issue cached loads and stores until all P's that precede them in "program order" have completed.

(ii) No processor may issue a V until all uncopies, including writebacks, that precede the V operation in "program order" have completed.

As presented therefore, our base protocol and sector invalidation, have been made fully consistent with release consistency. It seems clear that the base protocol would need to be extended somewhat to take full advantage of protects consistency, but we have not carried our research this far.

Chapter 6

Conclusion

"Every scientific truth goes through three stages. First, people say it conflicts with the Bible. Next they say it had been discovered before. Lastly, they say they always believed it."- Louis Agassiz.

"It is the customary fate of new truth to begin as heresies and to end as superstitions." - T.H. Huxley.

6.1 Our notion of a good cache coherence protocol

If caches are the front line of defence against both memory latency and memory bandwidth, then protocols will be complex. They will also be brittle and inefficient. They won't scale. If on the other hand, caches are merely handmaidens to multithreading, then compiler-implemented protocols can be simple. They can also be robust, efficient and scalable.

The aim of an efficient cache coherence protocol should be to handle conflicting data accesses, which cause inconsistencies in shared data, which are however lesser in frequency than non-conflicting data accesses, without causing delays to the latter which are generally more in number. Therefore, conservatism in invalidating cached copies should be curtailed, but on the other hand, we needn't be obsessed with high hit ratios. We look to switch- on-every-cycle multithreading to decrease latency and add a low-latency coherence protocol to augment it. We thus would rather have a simple low-latency protocol rather than a complex protocol which might result in slightly higher hit ratios but would be much more difficult to implement.

We propose compiler-implemented cache coherent protocols, where user/compiler knowledge(i.e. alias analysis, dependence analysis and user directives) is used to eliminate most coherence traffic[Pro94]. We follow the convention that visible synchronization directly coordinates changes in writeability of shared data, guiding the cached copy through the three states, viz., (i) private, (ii) shared and (iii) uncacheable. We replace *program-independent* automatic cache-coherency by *program specific* user/compiler control over (i) when to copy from memory to cache after visible synchronization input points and (ii) when to uncopy from caches before visible synchronization output points. PRC directives which use serial and parallel blocks and private-to-thread single assignment variables aid in this respect.

Directory based hardware cache coherency protocols spend too much time on run-time bookkeeping of variable to cache mappings, and entail high coherence traffic. There is also the significant point that in such protocols, synchronization action is separate from invalidation. That is, once the synchronization primitive is executed, the directory is looked up and the necessary *non-local* invalidations are made. Synchronization primitives function merely as guards, they ensure that invalidations(updates) do no harm. They are *not* perpetrators of invalidation action themselves. We seek to eliminate this redundancy by allowing visible synchronization to direct changes in writeability. The *protects* relation allows us to specify selective uncopying before output synchronization. Other optimizations keep invalidation to a minimum, while keeping the protocol's simplicity intact. The payoff is that a transition from n *shared* copies to 1 *private* copy incurs no global communication whatsoever. Since, all coherence actions are *local*, this protocol is fully scalable.

PRC specifies part of thread partial order by using (i) serial and parallel blocks to indicate scope of synchronization primitives and (ii) private to thread single assignment variables to express point-to-point $synch- > data$ precedence arcs. Both these mean that PRC doesn't have the unlimited scope problem with its synchronization primitives that release consistency has. We thus replace conventional program-independent cache coherency protocols by a user/compiler controlled program-specific protocol that is directly linked to a less restraining memory consistency model, whose primitives help in selective uncopying and copying of cached copies.

Cache coherency protocols lose their meaning if they are not intelligent, scalable, and cost-effective. If traditional automatic cache coherency protocols cannot provide us with qualitative use of communication bandwidth, they will be replaced by user/compiler managed coherency techniques that are program-behaviour specific, whilst maintaining simplicity of implementation. Latency tolerance methods like switch-on-every-cycle multithreading backed up by such coherency strategies ensure correctness of execution without compromising on scalability.

Bibliography

[A. 95] A. Agarwal et al. The mit alewife machine: Architecture and performance. In *Proc. of the Int'l Symposium on Computer Arhictecture*, pages 2–13, Ligure, Italy, 1995.

[A.J85] A.J. Smith. Cpu cache consistency with software supp.ort and using one time identifiers. *Proc. of the Pacific Computer Communications*, pages 153–161, 1985.

[A.V86] A.V. Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. In *Proc. ICPP*, pages 1029–1036, August 1986.

[Bur] Burton Smith. private communication with D.K. Probst.

[D. 81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th Annual Symposium on Computer Architecture*, pages 81–87, June 1981.

[D. 90a] D. Callahan and B.J. Smith. A future-based parallel language for a general purpose highly-parallel computer. In D. Gelernter et al., editor, *Languages and Compilers for Parallel Computing*, pages 95–113. MIT Press, 1990.

[D. 90b] D. Lenoski et al. The directory based cache coherence protocol for the dash multiprocessor. In *Proc. of the 17th Int'l Symposium on Computer Architecture*, pages 148–159, 1990.

[D. 91] D. Chaiken, J. Kubiatowicz, A. Agarwal. Limitless directories: A scalable cache coherence scheme. In *Proc. of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[D. 92a] D. Culler et al. Log p: Towards a realistic model of parallel computation. Tech. report ucb/csd 92/713, UC Berkeley, 1992.

[D. 92b] D. Lenoski et al. The stanford dash multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[D.J93] D.J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25:3:303–338, September 1993.

[G. 93] G. Papadopoulus, September 1993. private communication with D.K. Probst.

[H. 88] H. Cheong and A.V. Veidenbaum. A cache coherence with fast selective invalidation. In *Proc. 15th Int'l Symposium on Computer Architecture*, pages 299–307, 1988.

[J. a] J. Laudon. Ph.d. thesis, stanford university.

[J. b] J. Laudon, A. Gupta and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. Technical report, Computer Systems Laboratory, Stanford University.

[J. 92] J. Kubiatowicz, D. Chaiken and A. Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proc. of the 5th Int'l Conference on Architectural Supp.ort for Programming Languages and Operating Systems*, pages 274–284, October 1992.

[K. 90] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symposium on Computer Architecture*, pages 15–26, May 1990.

[K. 93] K. Mani Chandy and C. Kesselman. Cc++: A declarative concurrent object-oriented programming notation. Report caltech-cs-92-01, Department of Computer Science, California Institute of Technology, January 1992(revised March 1993.

[K.K87] K.K. McAuliffe. Ph.d. thesis nyu, May 1987.

[L. 78] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE trans. on Computers*, C27(12):1112–1118, 1978.

[L. 79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE trans. on Computers*, 28:9:690–691, September 1979.

[L. 93] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Src research report 95, DEC, February 1993.

[M. 86] M. Dubois et al. Memory access buffering in multiprocessors. In *Proc. of the 13th Int'l Symposium on Computer Architecture*, pages 434–442, June 1986.

[M.D93] M.D. Hill et al. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, pages 33–54, November 1993.

[Pro93a] Probst, D.K. Efficient implementation of thread partial order for a scalable shared-memory multiprocessor, May 1993. Third Workshop on Scalable Shared-Memory Multiprocessors, M. Dubois and S. Thakkar, orgs.

[Pro93b] Probst, D.K. Programming, compiling and executing partially-ordered instruction streams. In *Twenty-Seventh Hawaii International Conference on System Sciences, T. Mudge and B. Shriver, eds. Vol. I (Architecture), IEEE Computer Society Press, 1994*, pages 584 – 593, Maui, HI, January 1993.

[Pro94] Probst, D.K. User/compiler-managed register caches to eliminate coherence traffic in scalable shared-memory multiprocessors, April 1994. Fourth Workshop on Scalable Shared-Memory Multiprocessors, M. Dubois and S. Thakkar, orgs.

[R. 90] R. Alverson et al. The tera computer system. In *Proc. of the 17th Int'l Conference on Supercomputing*, pages 1–6, June 1990.

[R.L87a] R.L. Lee. The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors, ph.d. thesis, tech. rep. 670, August 1987. Center of Supercomputing Research and Development, U. of Illinois at Urbana-Champaign.

[R.L87b] R.L. Lee, P.C. Yew and D.H. Lawrie. Multiprocessor cache design considerations. In *Proc. of the 14th Int'l Symposium on Computer Architecture*, pages 253–262, June 1987.

[T. 92] T. von Eicken et al. Active messages: A mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture*, pages 256–266, May 1992.

[V.R86] V.R. Pratt. Modelling concurrency with partial orders. *Int'l Journal of Parallel Programming*, 15:1:33–71, February 1986.