Cache System Design for Parallel Processors

Shin-ichiro Mori

December 1994

Abstract

This thesis discusses three subjects concerning to cache system design for parallel processors. The three design policies of *transparency, hierarchy* and *adaptability* are applied to the cache system design in this thesis.

All these subjects have tight connection with practical machine implementations. The first subject discusses a cache system for an experimental parallel machine which opens up new vistas toward large scale shared memory multiprocessing by showing the possibility of real machine implementation through its hardware design. The cache system plays an important role in the design. The second subject discusses another cache system which explicitly exploits the concept of *hierarchy* in both mechanical and protocol designs. The last subject addresses the issue of memory update policy, or write-policy, and proposes a new technique to improve the cache performance.

The first and second subjects are intended to study system-wide aspects of cache systems, which adopt directory based cache coherence scheme in particular, whereas the third subject focuses on the cache itself.

In chapter 2, the cache architecture and cache coherence schemes for the multiprocessor system with a global inter connection network are proposed based on *the kyushu University Reconfigurable Parallel Processor(KRPP)* system design. KRPP is an MIMD-type multiprocessor which consists of 128 processors connected by a full (128×128) crossbar network. The system employs reconfigurable memory and network architectures to overcome the obstacles, topological mapping problems for example, to parallel processing.

When the KRPP is configured as a shared-memory tightly coupled multiprocessor, memory contentions will be obstacles to the performance. To relieve the effects, the system provides each PE with a private unified cache. Each PE may have the cached copy of shared data in its cache whether it accesses to local or remote memory, and therefore the multicache consistency, or *inter-cache coherence*, problem arises.

The KRPP adopts a virtual-address direct-mapped cache to meet the requirements for the hit time and size. The virtual-address cache implementation causes the other consistency problem, the "*synonym*" problem, which we call *intra-cache coherence* problem.

Chapter 2 also presents the cache coherence schemes that we have chosen to resolve these cache coherence problems:(1) Cacheability marking scheme, (2) Fast selective invalidation scheme, (3) Distributed limited-directory scheme, and (4)Dual-directory cache scheme. Cache coherence protocols and their tradeoffs among several aspects are also discussed.

In chapter 3, based on the experience of the KRPP development, a large scale, cluster-based, distributed shared memory multiprocessor ASURA is proposed. Up to 128 clusters are interconnected to form the ASURA system of up to 1024 processors. The architectural features of ASURA are placed in its layered memory system and corresponding hierarchical cache system. The following are the features:

1. Introduction of *hierarchical shared cache* : ASURA provides caches corresponding to the hierarchical levels of the system. In this cache hierarchy, a cache in a certain level is shared among caches in the next level. The hierarchical shared cache provides layer transparency in memory access time and at the same time it works as a multi-function interface which absorbs the architectural differences between two layers of the hierarchy.

2. The hierarchical directory scheme with the *MLI directory* : In order to maintain cache coherence, main memories and the shared caches are configured with full map directories which contain an existence bit per cluster, and the *MLI directories* respectively. The *MLI directory* contains information concerning to the *MultiLevel Inclusion property* and uses this information to enforce cache coherence at the corresponding layer.

3. Software assist for cache coherence : In order to reduce the run time overhead of coherence enforcement, ASURA provides some means for software to assist the hardware to enforce cache coherence. The basis of our software assist is the attribute marking scheme where the attributes corresponding to the line guide the hardware coherence enforcement.

In chapter 4, an improved Write-Back(WB) cache, which we call *Self-Cleanup Cache:SCC*, is proposed. The SCC aggressively enforces write-backs by itself to update memories. In

this way, the SCC keeps itself as clean as possible, while keeping the write merge effect of WB cache.

The SCC is originally designed to efficiently implement the invalidation based Compiler Aided Cache Coherence(CACC) scheme on the multiprocessor system which configured with WB caches. However, even for the systems which adopt directory based cache coherence scheme, the SCC has a good effect in decreasing read latencies.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are confirmed by simulation upon two cache system models: 1)the system with the SCC and the selective invalidation scheme which is one of the CACC schemes, and 2)the other system with the SCC and the full-map directory scheme, respectively. It is also confirmed that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

In chapter 5, the subjects discussed in this thesis are summarized and the thesis is concluded with some future works.

Contents

Abstract i

1 Introduction 1
- 1.1 Introduction to Parallel Processors . . . 1
  - 1.1.1 Classification of Parallel Processors . . . 1
  - 1.1.2 Performance of Parallel Processors . . . 3
- 1.2 Cache System Design . . . 5
  - 1.2.1 The Cache Coherence Problem in Parallel Processors . . . 6
  - 1.2.2 Approach Proposed in this thesis . . . 9
- 1.3 Outline of the thesis . . . 10

2 The Reconfigurable Parallel Processor : KRPP 13
- 2.1 Introduction . . . 14
- 2.2 Overview . . . 17
  - 2.2.1 Processing Element . . . 17
  - 2.2.2 Local/Remote Memory Architecture . . . 18
- 2.3 Cache Design . . . 20
  - 2.3.1 Design Considerations . . . 20
  - 2.3.2 Homonym and Synonym Problems of Virtual Address Cache . . . 22
- 2.4 Cache Coherence Schemes . . . 24
  - 2.4.1 Cacheability Marking Scheme . . . 25
  - 2.4.2 Fast Selective Invalidation Scheme . . . 25
  - 2.4.3 Distributed Limited-Directory Scheme . . . 26
  - 2.4.4 Dual-Directory Cache Scheme . . . 28
- 2.5 Hardware-Based Inter-Cache Coherence Enforcement . . . 29
  - 2.5.1 Protocols . . . 29
  - 2.5.2 Write-Policy Issues . . . 31
  - 2.5.3 Weak Ordering . . . 33
- 2.6 Intra-Cache Coherence Enforcement . . . 35
  - 2.6.1 Protocols . . . 35
  - 2.6.2 Collision Problem . . . 36

2.7 Analysis of the Collision Problem . . . 37
2.7.1 Probabilistic Analysis using a Markov Chain . . . 37
2.7.2 Trace-Driven Simulation . . . 40
2.8 Conclusion . . . 40

3 The Distributed Shared Memory Multiprocessor : ASURA 43
3.1 Introduction . . . 43
3.2 Memory Architecture . . . 46
3.2.1 Memory Organization . . . 46
3.2.2 Cache Architecture . . . 48
3.3 Cache Coherence . . . 52
3.3.1 Coherence Enforcement Strategy . . . 52
3.3.2 Software Assist for Cache Coherence . . . 59
3.4 Performance Simulation . . . 62
3.4.1 Simulation Environment . . . 62
3.4.2 Simulation Results . . . 64
3.5 Conclusion . . . 64

4 Self-Cleanup Cache 67
4.1 Introduction . . . 68
4.2 Related Works . . . 69
4.2.1 Compiler Aided Cache Coherence Control . . . 69
4.2.2 Latency Tolerance . . . 71
4.3 Self-Cleanup Cache . . . 72
4.3.1 Overview of the Self-Cleanup Cache . . . 72
4.3.2 Self-Cleanup Strategies . . . 73
4.3.3 An Implementation Example of the SCC with LRU Replacement Policy 74
4.3.4 Comparison with the Write Buffer Approach . . . 77
4.4 Methodology of the Performance Evaluation . . . 78
4.4.1 Goal of the Performance Evaluation . . . 78
4.4.2 Simulation Models . . . 78
4.4.3 Overview of the Simulator . . . 80
4.5 Simulation Result . . . 81
4.5.1 Summary of Results . . . 81

4.5.2 SOR 82
4.5.3 FFT 83
4.5.4 WATER 86
4.5.5 MP3D 89
4.6 Conclusion 89

5 Conclusion **92**
5.1 Summary of This Thesis 92
5.2 Future Directions 95
5.2.1 Development of the Cache Coherence Scheme for Massively Parallel Processors 95
5.2.2 Development of the Automatic Parallelizing Compiler 95
5.2.3 Quantitative Performance Analysis 96

Acknowledgement **97**

Bibliography **99**

List of Publications by the Author **106**

List of Figures

1.1 Memory Organization 4
1.2 Speedup Attained by parallel Processing 5

2.1 Configuration of the Reconfigurable Parallel Processor 15
2.2 Addressing Scheme 19
2.3 Cache Organization (in case of Block Size = 32 bytes) 21
2.4 Double-Link between V-tag and R-tag 29
2.5 Inter-Cache Coherence Enforcement with Coherence Directory 32
2.6 Collision problem 36
2.7 State Transition 38

3.1 Overview of ASURA 44
3.2 Memory architecture of ASURA 47
3.3 Memory map of ASURA prototype 48
3.4 Memory hierarchy of ASURA 50
3.5 The Hierarchical Cache Coherence Enforcement Mechanism 54
3.6 State Transition Diagram 55
3.7 A ring-crossbar configuration of ASURA 63
3.8 Simulation Results 65

4.1 Read Latency on Write-Back Cache. 72
4.2 Organization of SCC 73
4.3 An implementation of the SCC with LRU replacement policy. 74
4.4 System configuration 80
4.5 SOR : Execution time 84
4.6 SOR : Number of messages 84
4.7 SOR : Execution time (after optimization) 85
4.8 SOR : Number of messages (after optimization) 85
4.9 FFT : Execution time 87
4.10 FFT : Number of messages 87
4.11 WATER : Execution time 88
4.12 WATER : Number of messages 88

4.13 MP3D : Execution time 90
4.14 MP3D : Number of messages 90

List of Tables

1.1 Classification of the Parallel Processor by the Memory Architecture 3
1.2 Hardware Cost for three directory schemes 7

2.1 Specifications of the Recinfigurable Parallel Processor 16
2.2 Specification of the Dual-Directory Cache 20
2.3 Cache Design on Access-Address and Associativity 23
2.4 Results by Probabilistic Analysis 40
2.5 Results by Trace-Driven Simulation 41

3.1 Specification of ASURA prototype 45
3.2 The Summary of Memory Architectures of Typical Cluster Based System . 49
3.3 Memory Access Latencies 63

4.1 System configuration parameters 81

Chapter 1

Introduction

Parallel processing is a very promising approach to the next generation supercomputer. There exist however detrimental factors which impede the speedup of parallel processors. The cache system is one of the techniques to decrease such detrimental factors.

This thesis discusses three subjects concerning to cache system design for parallel processors, beginning with brief overviews of parallel processors and cache system design.

1.1 Introduction to Parallel Processors

1.1.1 Classification of Parallel Processors

1.1.1.1 Flynn's classification

According to the Flynn's classification[25], parallel processors are placed in one of three categories:

1. Single instruction stream, multiple data streams (**SIMD**)
2. Multiple instruction streams, single data stream (**MISD**)
3. Multiple instruction streams, multiple data streams (**MIMD**)

This classification is based on the parallelism in the instruction and data streams.

The MIMD type parallel processors are further classified, by their instruction issuing strategies, into *control flow* (von Neumann style) and *data flow* (non von Neumann style)

parallel processors. Instructions are executed sequentially as controlled by the program counter in the control flow parallel processor, while in the data flow parallel processor the execution of each instruction is enabled whenever its required operands become available.

The MIMD type control flow parallel processor is referred to as *multiprocessor* in this thesis[1].

Discussions in this thesis are focused on the multiprocessor, however, the essence of this study is also applicable to the other types of parallel processors.

1.1.1.2 Classification by The Memory Architecture

From the architectural point of view, the parallel processor is characterized by the processor, memory, and network architectures. In the Flynn's classification, the discussion is focused on the processor architecture. In this section, parallel processors are classified by the memory architecture. Three aspects of the memory architecture: 1) the physical memory organization, 2) the memory access time and 3) the logical view from the programmer side, are considered in the classification.

- The physical memory organization:
  With respect to how memory is physically incorporated into a parallel processor configuration, there are two memory organizations : *concentrated memory organization*(Figure 1.1(a)) and *distributed memory organization*(Figure 1.1(b) and (c)).

- The memory access time:
  The memory access time is the second physical index. The memory organization where all the memory locations in address space are accessible in equal time to all processors is referred to as *Uniform Memory Access*(UMA) architecture (see figure 1.1(a)). Otherwise, it is referred to as *Non Uniform Memory Access*(NUMA) architecture or *local/remote* architecture(see figure 1.1(c)).

  The memory configuration of figure 1.1(b) falls into either UMA or NUMA architecture according to the characteristics of the network architecture. For example, if the system adopts the omega network this configuration will fall into the UMA architecture, while the system with the mesh network will fall into the NUMA architecture.

[1]Some literatures[32, 57] use the word *multiprocessor* in contrast to the *uniprocessor*.

Table 1.1: Classification of the Parallel Processor by the Memory Architecture

Memory Organization		Concentrated Memory	Distributed Memory	
Memory Access Time		Uniform	Uniform	Non-uniform
Logical View	Shared Memory	UMA		NUMA
	Message Passing	NORA		

UMA : Uniform Memory Access, NUMA : Non uniform Memory Access
NORA : No Remote memory Access

- The logical view from the programmer:
  With respect to the programming model which the parallel processor provides to the programmer, parallel processors are divided into two classes:
  - *shared memory* architecture, or *tightly coupled* architecture, and,
  - *message passing* architecture, or *loosely coupled* architecture.

  The shared memory parallel processor communicates with each other via the memory which is shared among processors, whereas the message passing parallel processor communicates by exchanging messages through the message communication facilities.

  The message passing architecture is also referred to as *NO Remote memory Access*(NORA) architecture[66], in contrast to UMA and NUMA .

Figure 1.1 and table 1.1 show the typical memory organizations.

1.1.2 Performance of Parallel Processors

There are two ways to discuss the performance of a computer[57]:

- *Response time* : the time between the start and the completion of an event – also referred to as *execution time* or *latency*, and
- *Throughput* : the total amount of work done in a given time – sometimes called *bandwith*

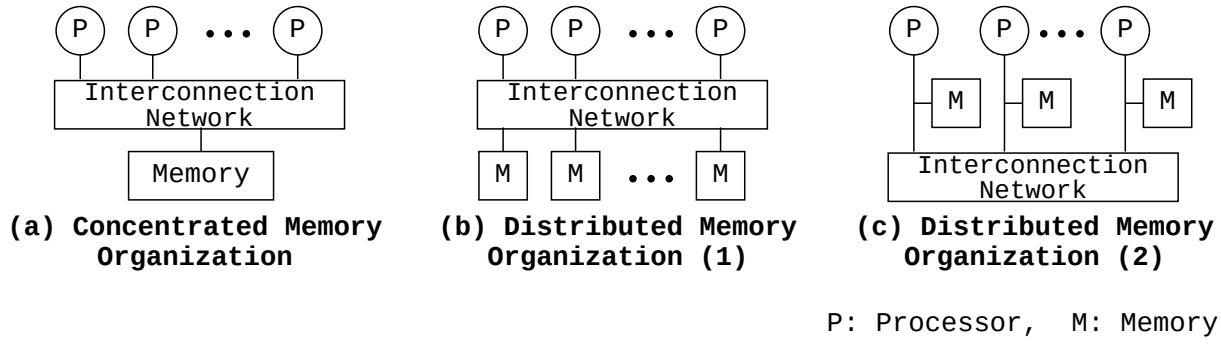


Figure 1.1: Memory Organization

Parallel processing is expected to improve both the response time and the throughput. Here we discuss the performance of the parallel processor in term of the *speedup* in response time over the performance of the single processor.

The speedup that can be achieved by a parallel processor with N identical processors working concurrently on a single problem is at most N times faster than a single processor[32]. In practice, the speedup is much less, since some processors are idle at a given time because of the following factors[55] which impede the speedup:

- the intrinsic serial work in a given program, which can not be executed in parallel, and
- the extrinsic work introduced in parallelizing the program.

The latter is what we call *parallelizing overhead* : which is due to data exchange and *synchronization* between processors, and the other miscellaneous work required in parallel processing, like cache coherence enforcement which is discussed later.

Let's assume that N is the hypothetical speedup of parallel processing over serial processing, S is the ratio of time spent on serial parts of a program, P is the ratio of time spent on parallel parts of the program ($S + P = 1$), and O is the overhead, in relative time to the run time on single processor, incurred in parallel processing, then Amdahl's low[4] states that the practical speedup is given by the following equation.

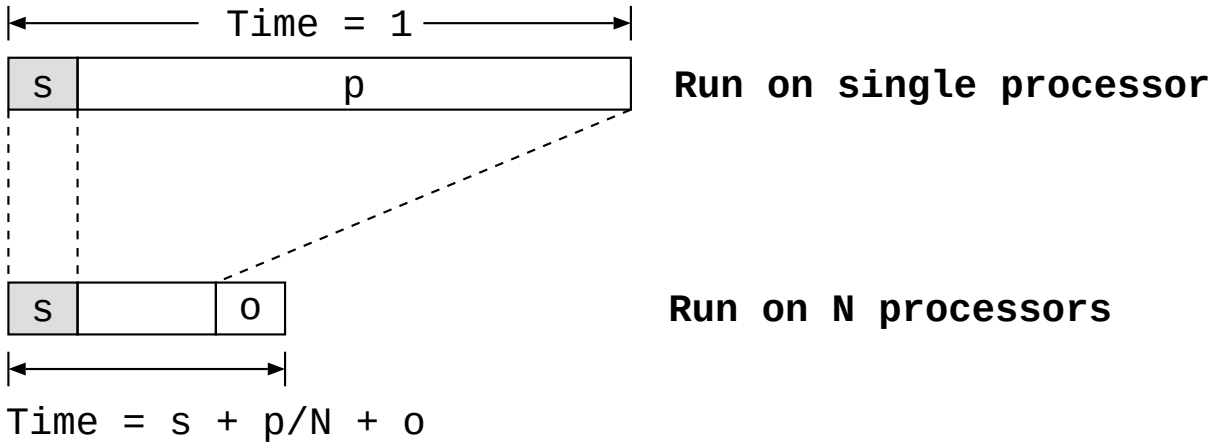


Figure 1.2: Speedup Attained by parallel Processing

$$\begin{aligned} Speedup &= \frac{S+P}{S+\frac{P}{N}+O} \\ &= \frac{1}{S+\frac{(1-S)}{N}+O}. \end{aligned} \tag{1.1}$$

This equation indicates that, even with an infinite number of processors, the maximum speedup obtainable is only $\frac{1}{S+O}$. And hence decreasing both S and O is indispensable to improve the speedup.

Introduction of the cache system is a technique to decrease such parallelizing overhead at hardware level.

1.2 Cache System Design

The *cache memory*, or the *cache* for short, is small, high-speed buffer memory to hold temporarily those protions of the contents of main memory which are (believed to be) currently in use[62].

By exploiting the most important program property[20], *property of locality*, the cache can serve requested data to processors much faster than main memory. The property of locality has two aspects:

- *Temporal locality*(locality in time): means that the data which will be in use in the near future is likely to be in use already.

- *Spatial locality*(locality in space): means that the location of reference of the program in the near future are likely to be near the current location of reference.

In the parallel processor environment, if each processor is configured with its private cache, the cache plays another important role in decreasing the memory access traffic on the interconnection network, and thus decreasing the contention on main memories and on the network. It is because the caches serve the processores, concurrently and independently, to provide requested data instead of main memories. There exists, however, a problem of cache coherence in such parallel processor systems with private caches.

1.2.1 The Cache Coherence Problem in Parallel Processors

A parallel processor system with private caches faces the problem of making sure that all copies of a given piece of information (which potentially could exist in every cache, as well as in the main memory) are the same[62]. The *cache coherence problem* it is.

Although the cache, in the parallel processor system, itself is expected to decrease the *parallelizing overhead*, the cache coherence enforcement has a possibility of increasing the *parallelizing overhead*. Minimizing the overhead of maintaining cache coherence hence has great importance in the parallel processor design. Solutions of the cache coherence problem fall into two categories: hardware-based approach and software-based approach.

1.2.1.1 Hardware-Based Cache Coherence Scheme

There are two classes of schemes to solve the cache coherence problem with some hardware mechanisms[57]:

- *Directory scheme*: The information about one block of physical memory is kept in just one location.

- *Snoop scheme*: Every cache that has a copy of the data from a block of physical memory also has a copy of the information about it. These caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of the shared block.

Table 1.2: Hardware Cost for three directory schemes

Scheme	Hardware Cost
Full-map	$M \times N$
Limited	$i \times M \times \log_2 N$
Chained	$(M + C) \times \log_2 N$

In directory-based schemes there is logically a single directory that keeps the state of every block in the main memory. Information in the directory can include which caches have copies of the block, whether it is dirty, and so on. Of course directory entries can be distributed so that different requests can go to different memories, thereby reducing contention. However, they retain the characteristic that the sharing status of a block is always in a single known location. The implementations of the directory-based scheme is further subdivided into three subclasses[64]:

- *Full-map directory scheme*: A bit vector, called the *presence flag vector*, is associated with each memory block[12]. One bit for each cache indicates which caches have a copy of the block.

- *Limited directory scheme*: Instead of the bit vector, pointers to the caches which contain a copy of the memory block are associated with each memory block. The number of pointers is less than the actual number of caches.

- *Chained directory scheme*: A link list of all caches that store a copy of the memory block is associated with each memory block. An implementation of this scheme is that each cache line and memory block is associated with an entry which contains a pointer to the next cache that stores a copy.

Table 1.2 shows the hardware cost for each directory scheme, given M memory blocks, C cache lines, and N caches(, and i pointers for the limited directory scheme).

Snoop schemes became popular with multiprocessors using microprocessors and caches on a shared memory because they can use a preexisting physical connection: the bus to memory. Snooping has an edge over directory schemes in that the coherency information is proportional to the number of blocks in a cache rather than the number of blocks in main

memory. Directories, on the other hand, do not require a single bus going to all caches and, hence may scale to more processors.

The coherency problem is for a processor to have exclusive access to write an object and to have the most recent copy when reading an object. Thus, both directory-based and snooping schemes must locate all the caches that share the object to be written. The consequence of a write to shared data is either to invalidate all other copies or to broadcast the write to the shared copies. Because of write-back caches, coherency scheme must also help determine who has the most up-to-date value at read misses.

1.2.1.2 Software-Based Cache Coherence Scheme

Software-based cache coherence schemes attempt to avoid the need for complex hardware mechanisms and to reduce the run-time communication overhead in coherence enforcement. However, the price is paid in terms of the increase of cache misses.

There exist two directions to maintain cache coherence by software : 1) Inhibit the caching of data which may cause inconsistency if it is cached, *cacheability marking scheme*[27, 9], and 2) Insert cache invalidation instructions to invalidate the *stale* data, *self-invalidation scheme*[70, 16, 17, 47]. Both of them are based on the static analysis of the program by the compiler.

- Cacheability marking scheme
  To prevent the existence of inconsistent cached data, caching of data is restricted to only when it is safe. For this purpose, the compiler analyzes the program to mark variables as *cacheable* or *noncacheable*. The most trivial and conservative solution is to mark all shared writable variables as noncacheable.

- Self-Invalidation scheme

  A compiler inserts cache invalidation instruction which, when executed, either invalidates the whole data cache of executing processor(*indiscriminate invalidation*) or invalidates a selected cache line(*selective invalidation*). Such a local invalidating action normally occurs at the beginning of a new task which is created at the run time of a parallel program.

Indiscriminate invalidation tends to grossly invalidate data which may actually be valid. On the other hand, selective invalidation requires executing as many invalidating instructions as the number of cache lines to be invalidated, which is quite a high overhead. Recently, a couple of remedies for the indiscriminate invalidation have been proposed to reduce the degree of over-invalidation[16]. Such remedies, termed as *fast selective invalidation*, can be quite effective if the compiler is able to analyze the program's data flow with high precision. Unfortunately, a precise static analysis is known to be difficult, due to such complications in data access patterns as subscripted array and pointers. The inability of the compilers to predict run time execution paths also poses a major obstacle. Finally, dynamic task scheduling brings further difficulties to the static analysis. Some of these problems may be reduced by techniques such as *version control*[17] or *time-stamp*[47] approaches.

1.2.2 Approach Proposed in this thesis

The software-based cache coherence schemes are attractive because they do not generate coherence enforecement traffic, however the effectiveness of software-based approaches relies heavily on the ability of the compiler. So far, the compiler is not so sophisticated that it can deal with the cache coherence problem, thus the software-based approaches risk a high miss-ratio and heavy overall traffic due to over-invalidation.

This dilemma motivates many researchers[15, 36, 43] to find a solution to the cache coherence problem by combining the software-based approach and hardware-based approach of the directory scheme. The study described in this thesis had been pursued also in this direction based on the following policies.

1. *Transparency*
   The self-contained behavior of the cache system, which offers the transparency to the software, thus there is no need to modify the existed program, is one of the reasons that makes the cache become widely used. This feature should be kept even in the parallel processor in order for the effect of cache not to depend on the ability of the software. From this point of view, the directory based cache coherence scheme has been chosen as the common base of this study.

2. *Hierarchy*
   Most of the physical restrictions, like area, access time, and number of pins, in com-

puter design are scaled not continuously but discretely when the computer itself is scaled up. According to the discreteness of such restrictions, hierarchical design of the computer is required. The cache, as a part of the memory hierarchy, also requires itself to be designed hierarchically, so that the most suitable design alternatives are chosen at each level of the hierarchy.

3. *Adaptability*
   The software, or the compiler, has a great possibility to improve the cache performance because it can take advantage of the information about the behavior of the programs to be executed. In order to make use of this advantage, though it conflicts with the policy of transparency, some means to collaborate on cache operations should be provided for the software so that the cache can accommodate itself to the behavior of the programs.

1.3 Outline of the thesis

This thesis discusses three subjects concerning to cache system design for parallel processors. The three design policies of *transparency, hierarchy* and *adaptability* are applied to the cache system design in this thesis.

The first subject of this thesis discusses a cache system for an experimental parallel machine which opens up new vistas toward large scale shared memory multiprocessing by showing the possibility of real machine implementation through its hardware design. The cache system plays an important role in the design. The second subject discusses another cache system which explicitly exploits the concept of *hierarchy* in both mechanical and protocol designs. The last subject addresses the issue of memory update policy, or write-policy, and proposes a technique to improve the cache performance.

The first and second subject are intended to study system-wide aspects of cache systems, which adopt directory based cache coherence scheme in particular, whereas the third subject focuses on the cache itself.

In chapter 2, the cache architecture and cache coherence schemes for the multiprocessor system with a global inter connection network are proposed based on *the kyushu University Reconfigurable Parallel Processor(KRPP)* system design. KRPP is an MIMD-type multiprocessor which consists of 128 processors connected by a full (128×128) cross-

bar network. The system employs reconfigurable memory and network architectures to overcome the obstacles, topological mapping problem for example, to parallel processing.

When the KRPP is configured as a shared-memory TCMP, memory contentions will be obstacles to the performance. To relieve the effects, the system provides each PE with a private unified cache. Each PE may have the cached copy of shared data in its cache whether it accesses to local or remote memory, and therefore the multicache consistency, or *inter-cache coherence*, problem arises.

The KRPP adopts a virtual-address direct-mapped cache to meet the requirements for the hit time and size. The virtual-address cache implementation causes the other consistency problem, the "*synonym*" problem, which we call *intra-cache coherence* problem.

Chapter 2 presents the cache coherence schemes that we have chosen to resolve these cache coherence problems: (1) Cacheability marking scheme, (2) Fast selective invalidation scheme, (3) Distributed limited-directory scheme, and (4)Dual-directory cache scheme. Cache coherence protocols and their tradeoffs among several aspects are also discussed.

In chapter 3, based on the experience of the KRPP development, a large scale, cluster-based, distributed shared memory multiprocessor ASURA is proposed. Up to 128 clusters are interconnected to form the ASURA system of up to 1024 processors. The architectural features of ASURA are placed in its layered memory system and corresponding hierarchical cache system. The following are the features:

1. Introduction of *hierarchical shared cache* : ASURA provides caches corresponding to the hierarchical levels of the system. In this cache hierarchy, a cache in a certain level is shared among caches in the next level. The hierarchical shared cache provides layer transparency in memory access time and at the same time it works as a multi-function interface which absorbs the architectural difference between two layers of the hierarchy.

2. The hierarchical directory scheme with the *MLI directory* : In order to maintain cache coherence, main memories and shared caches are configured with full map directories which contain an existence bit per cluster, and the *MLI directories* respectively. The *MLI directory* contains information concerning to the *MultiLevel Inclusion property* and uses this information to enforce cache coherence at the corresponding layer.

3. Software assist for cache coherence : In order to reduce the run time overhead of coherence enforcement, ASURA provides some means for software to assist the hard-

ware to enforce cache coherence. The basis of our software assist is the attribute marking scheme [8, 9] where the attributes corresponding to the line guide the hardware coherence enforcement.

In chapter 4, an improved Write-Back(WB) cache, which we call *Self-Cleanup Cache:SCC*, is proposed. The SCC automatically enforces write-backs by itself to update memories. In this way, the SCC keeps itself as clean as possible, while keeping the write merge effect of WB cache.

The SCC is originally designed to efficiently implement the invalidation based Compiler Aided Cache Coherence(CACC) scheme on the multiprocessor system which configured with WB caches. However, even for the systems which adopt directory based cache coherence scheme, the SCC has a good effect in decreasing read latencies.

The behavior of the Self-Cleanup Cache is examined by simulation. And the simulation results show that 1) Invalidation based software cache coherence scheme can work efficiently with the assist of the Self-cleanup cache, and 2) Read stall time is reduced as a result of eager write-back initiated by Self-Cleanup mechanism.

In chapter 5, the subjects discussed in this thesis are summarized and the thesis is concluded with some future works.

Chapter 2

The Reconfigurable Parallel Processor : KRPP

Abstract

The kyushu University Reconfigurable Parallel Processor system is an MIMD-type multiprocessor which consists of 128 SPARC microprocessors connected by a full (128 × 128) crossbar network. The system employs reconfigurable memory architecture, a kind of local/remote memory architectures, and encompasses a shared-memory TCMP (tightly coupled multiprocessor) and a message-passing LCMP (loosely coupled multiprocessor).

When the system is configured as a shared-memory TCMP, memory contentions will be obstacles to the performance. To relieve the effects, the system provides each PE with a private unified cache. Each PE may have the cached copy of shared data in its cache whether it accesses to local or remote memory, and therefore the multicache consistency, or inter-cache coherence, problem arises.

The cache is a virtual-address direct-mapped cache to meet the requirements for the hit time and size. The virtual-address cache implementation causes the other consistency problem, the "synonym" problem, which we call intra-cache coherence problem.

This chapter presents the cache coherence schemes that we have chosen for resolving these cache coherence problems:(1)Cacheability marking scheme, (2)Fast selective invalidation scheme, (3)Distributed limited-directory scheme, and (4)Dual-directory cache scheme. Cache coherence protocols and their tradeoffs among several aspects are also discussed.

2.1 Introduction

The kyushu University Reconfigurable Parallel Processor system is an MIMD-type multiprocessor which consists of 128 processing elements(PEs), interconnected by a full (128 × 128) crossbar network [54].

The goals of the system are:

1. to construct a high-performance, multi-purpose multiprocessor system which can be tailored to a broad range of applications, and

2. to offer an experimental environment, or testbed, which encourages many researchers in the studies of highly parallel processing [52].

Figure 2.1 and Table 2.1 show the configuration and specification of the system, respectively.

For these goals to be achievable, the system employs a reconfigurable network architecture, which provides dynamic flexible interconnection topologies to fill the gap between intercommunication topologies inherent in parallel algorithms and those of the system. The system also employs reconfigurable memory architecture, a kind of local/remote memory architectures. Although it is built in the form of distributed memory organization without global memory, the system can be reconfigured as either a shared-memory TCMP (tightly coupled multiprocessor) or a message-passing LCMP (loosely coupled multiprocessor) [53].

When the system is configured as a shared-memory TCMP, memory contentions will be obstacles to the performance. To alleviate their detrimental effects, the system provides each PE with a private unified cache. Each PE may have the cached copy of shared data in its cache whether it accesses local or remote memory, and therefore the multicache consistency, or *inter-cache coherence*, problem will occur.

The cache is a virtual-address direct-mapped cache to offer the cache access time of one cycle and the large cache size of 128K bytes. The virtual-address cache implementation causes the other consistency problem, the "*synonym*" problem, which we call *intra-cache coherence*, problem.

This chapter presents the cache coherence schemes that we have chosen for resolving these cache coherence problems. From the standpoint of a testbed, the system provides the various schemes as follows:

1. Cacheability marking scheme to enforce inter-cache coherence [27, 9]

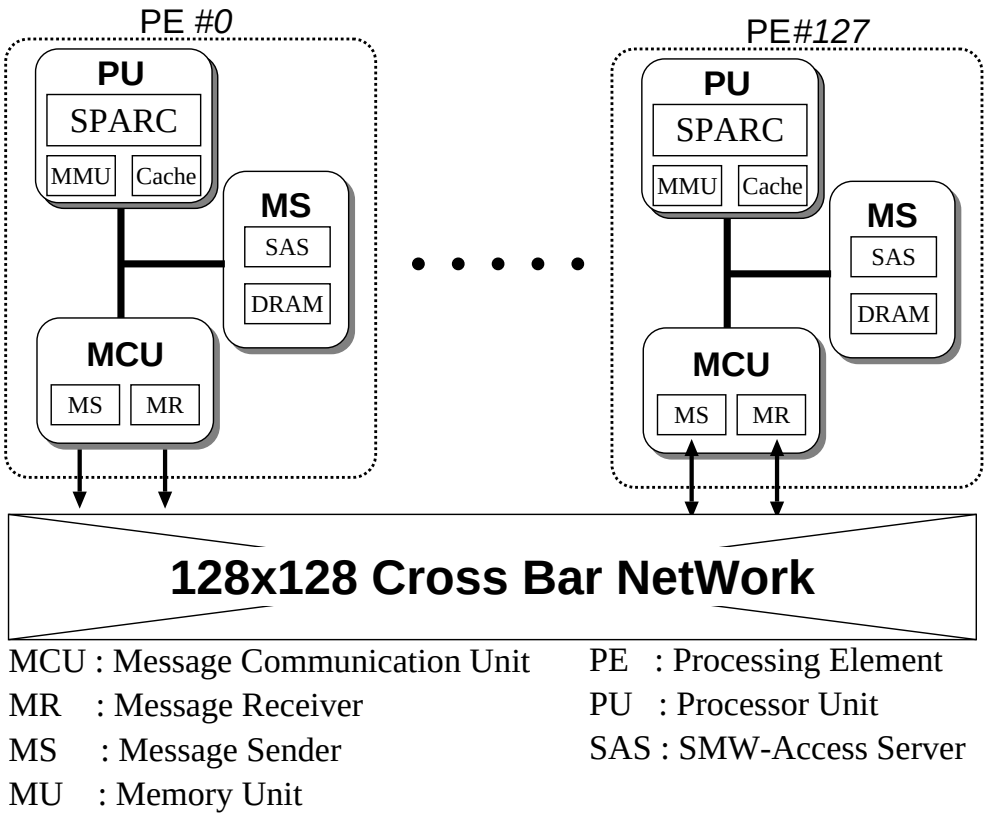


Figure 2.1: Configuration of the Reconfigurable Parallel Processor

Table 2.1: Specifications of the Recinfigurable Parallel Processor

Number of PEs	128
Size of the Crossbar Switch	128 × 128
Inter-PE Communication	8-bit-width circuit switching
Communication Bandwith	17M bytes/sec. (for each PE pair)
	2.1G bytes/sec. (total)
Memory Organization	Distributed Memory Organization
Memory Size	8M bytes (for each PE)
	1G bytes (total)
PE Configuration	MPU : SPARC MB86901 (16.67MHz)
	FPU : Abacus 3170 (16.67MHz)
	Cache : 128K bytes (SRAM)
	Memory : 8M bytes (DRAM)
	Message Communication Unit, etc.
Single PE Performance	10 VAX MIPS
	1.6 single-precision LINPACK MFLOPS
	1.1 double-precision LINPACK MFLOPS
System Performance	1280 VAX MIPS
	205 single-precision LINPACK MFLOPS
	141 double-precision LINPACK MFLOPS

2. Fast selective invalidation scheme to enforce inter-cache coherence [16]

3. Distributed limited-directory scheme to enforce inter-cache coherence [5]

4. Dual-directory cache scheme to enforce both intra-cache and inter-cache coherence [29]

The next section overviews the PE configuration and the local/remote architecture. Section 2.3 discusses several cache aspects that we have taken into consideration on designing the cache. Section 2.4 introduces the cache coherence schemes given above. In section 2.5 we develop the inter-cache coherence protocol and examine two issues: write-policies and weak ordering. In section 2.6 we develop the intra-cache coherence protocols and analyze the collision problem caused by the dual-directory cache scheme in section 2.7. The final section offers a few concluding remarks.

2.2 Overview

2.2.1 Processing Element

Each PE comprises three main units: Processor Unit(PU), Message Communication Unit (MCU), and Memory Unit(MU).

The PU consists of a Fujitsu SPARC[1] MB86901 (S-20)[74] and an Weitek[2] ABACUS 3170. The MB86901 is a RISC processor which performs at roughly 10 VAX MIPS when operating at 16.67 MHz clock frequency. The ABACUS 3170 is a floating-point coprocessor, which achieves 1.6 single-precision LINPACK MFLOPS and 1.1 double-precision LINPACK MFLOPS at 16.67 MHz clock frequency.

The MMU provides zero-wait-state address translation and instruction/data caching for the processor. It contains a TLB (translation-lookaside buffer) and a virtual address cache of 128K bytes. Most of the MMU facilities, except for the RAM array of cache, are implemented in a single-chip semicustom LSI.

The MCU provides the network interface and serves remote memory references which are forwarded to/from other PEs' local memory.

[1]SPARC is a trademark of Sun Microsystems, Inc.

[2]Weitek is a resistered trademark of WEITEK Corporation.

The MU has 8M bytes of local memory. It includes a SAS (Shared-memory-window Access Server) which realizes remote memory references from the other PEs and maintains multicache consistency.

2.2.2 Local/Remote Memory Architecture

As shown in Figure 2.1, the reconfigurable parallel processor system employs a distributed memory organization, and provides no global memory. To exploit the memory reconfigurability, however, the system allows more than one processor to share a local memory. A processor of each PE can directly access the local memories of the other PEs, which memories are called "*remote memories*", via the crossbar network. The memory architecture is referred to as "*local/remote architecture.*" As the result, the system encompasses a shared-memory TCMP and a message-passing LCMP.

The local/remote architectures are employed by some multiprocessor systems such as the CMU Cm* [65], the IBM RP3 [9], and the BBN Butterfly[3] [58]. Those local/memory architectures, including ours, differ in their addressing schemes.

Figure 2.2 schematically depicts the addressing scheme of the reconfigurable parallel processor. The scheme is achieved by two-level address translation. Because of the two-level address translation, three address representations are required: virtual, real, and physical. The two-level address translation is performed as follows.

(a) Mapping VAS to RAS (first-level address translation) : For PE_m, a virtual-address space (VAS_m) of 4G bytes is mapped to the real-address space (RAS_m) of 4G bytes, as shown in Figure 2.2-a. The memory-management unit (MMU_m) translates a 32-bit virtual address (VA_m) generated by a SPARC microprocessor into a 32-bit real address (RA_m) with a conventional two-level paging. The page size is 4K bytes.

(b) Partitioning RAS into CS and PS : Each RAS is partitioned into the following two spaces;

1. Private space (PS): For PE_m, a low-order 2G-byte space (i.e., MSB of real address is 0) is its private space (PS_m), which is identical to its physical-address space (PAS_m). Thus a private-space real-address ($PSRA_m$), excluding its MSB, is directly used as a physical address (PA_m)

[3]Butterfly is a resistered trademark of Bolt Beranek and Vewman, Inc.

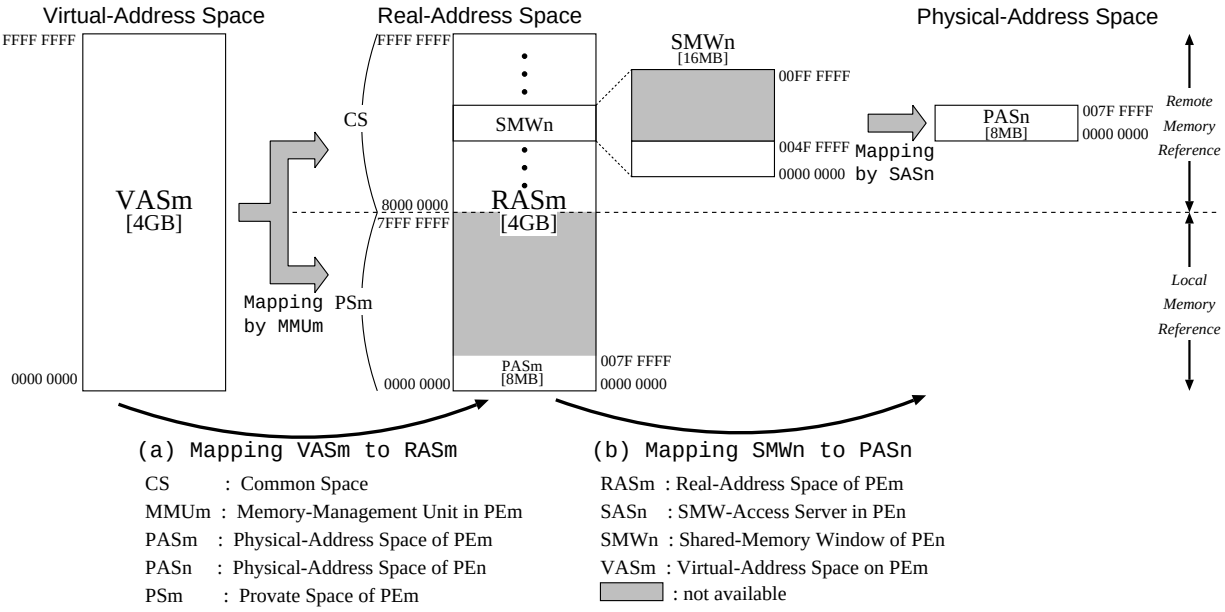


Figure 2.2: Addressing Scheme

2. Common space (CS): A high-order 2G-byte space (i.e., MSB of real address is 1) is common to all PEs. The common space comprises 128 SMWs (Shared-Memory Windows), each of which is 16M bytes in size and segmented into 4K SMW-pages. An SMW is assigned to a PE, and is indexed by the PE number.

(c) Mapping SMW to PAS (second-level address translation) : If an SMW_n is a target of remote memory reference by PE_m, an SMW-access request is forwarded from PE_m to PE_n via the crossbar network in message form. On the PE_n, the target SMW_n is mapped to its physical-address space (PAS_n), as shown in Figure 2.2-b. That is, the SMW-access server (SAS_n) translates the received common-space real-address (CSRA), excluding a portion of SMW index, into a physical address (PA_n) with a conventional one-level paging.

From the viewpoints of a processor and a local memory, the addressing scheme makes no difference between local and remote memory references in the sense that both references can equally perform memory reads/writes for all the data sizes (i.e., byte, halfword, word, and doubleword). The only difference is the local/remote memory access time ratio, and

Table 2.2: Specification of the Dual-Directory Cache

Cache	Access address		Virtual address	
	Associativity		Direct-mapped	
	Replacement policy		——	
	Write policy		Write-Through	
	Write-Miss option		No-Write-Allocate	
	Unified/Split cache		Unified	
	Address block(Line) size[byte]		32	64
	Transfer block size [byte]		32	64
	Coherence block size [byte]		32	64
	Cache size [byte]		64K	128K
	Number of sets		2048	
	Hit time [CLK]		1	
	Miss Penalty	local memory [CLK]	17	25
		remote memory[CLK]	64	96
RTB	Access address		Real address	
	Associativity		4-way set-associative	
	Replacement policy		LRU	
	Number of sets		512	

RTB : Reverse Translation Buffer CLK : Clock cycle

therefore the scheme falls under nonuniform-memory-access (NUMA) architectures. Rough estimates of local/remote memory access time ratio are given in [53]. The effects on cache-miss penalty is shown in Table 2.2.

2.3 Cache Design

2.3.1 Design Considerations

To provide zero-wait-state instruction/data caching for a SPARC microprocessor at 16.67 MHz clock frequency, each PE should have a private cache. On implementing a particular

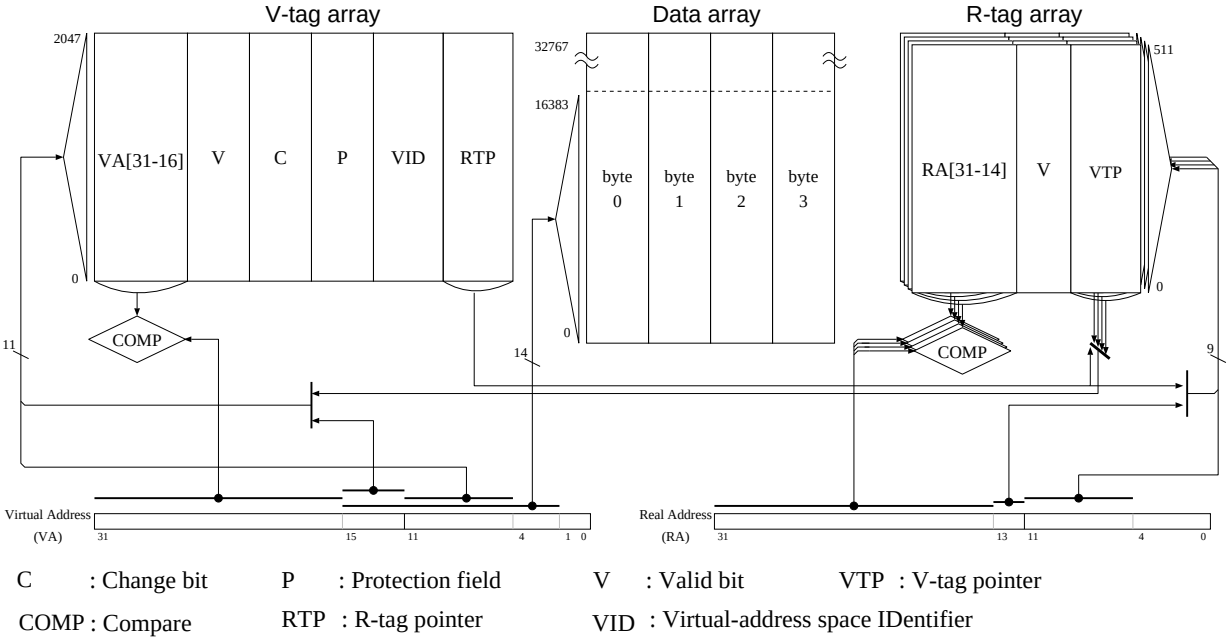


Figure 2.3: Cache Organization (in case of Block Size = 32 bytes)

cache, we must consider the various aspects of cache design [62]. Figure 2.3 and Table 2.2 respectively show the block diagram and the specifications of the cache we have designed.

The cache design parameters we take into consideration here are cache-access address and associativity. Here we do not address the questions of block sizes and write policy. We can select either 32-byte block or 64-byte block appropriately. There are two write policies: write-through and write-back; we have chosen the write-through policy for the reason given in section 2.5.2.

There are two options on the addresses used to access the cache: virtual-address cache and real-address cache [62]. In the two-level address translation scheme there is an additional option, physical-address cache; but we neglect it since it is obviously impractical. For real-address caches, there are two additional options on address translation on TLB lookup: parallel translation or serial translation[33] There are at least two options on associativity or mapping: direct-mapped cache and set-associative cache[62, 33].

The following six alternatives are considered with respect to the cache-access address and the associativity.

1. Virtual-address direct-mapped cache,
2. Virtual-address set-associative cache,
3. Real-address direct-mapped cache with parallel address translation,
4. Real-address set-associative cache with parallel address translation,
5. Real-address direct-mapped cache with serial address translation, and
6. Real-address set-associative cache with serial address translation.

Table 2.3 summarizes their impacts on cache access time (or cache hit time), cache size, and hardware cost. We assume that the critical timing path for the cache access is dominated by the tag-access time and tag-compare time rather than by the data read time.

We have chosen the virtual-address direct-mapped cache organization, because it has the smallest cache access time and the lowest hardware cost, and also because the cache size can get larger only by increasing the number of sets (or lines) without any additional costs.

On the other hand, with respect to the miss ratio, there are arguments against direct-mapped caches. The most important one is that direct-mapped caches have worse miss ratios than set-associative caches of the same size. In fact, however, the miss ratio differences between direct-mapped caches and set-associative caches diminish as the caches get larger[33].

2.3.2 Homonym and Synonym Problems of Virtual Address Cache

There are also arguments against virtual-address cache with regard to consistency inside a single cache. These consistency problems are called "*homonym*" and "*synonym*" problems. We have given the problems the following additional considerations in designing the virtual-address cache.

2.3.2.1 Support to multiple virtual-address spaces

Virtual addressed caches incur the problem of *homonym* between two or more address spaces. All addresses must be tagged with the identifier of the address space with which

Table 2.3: Cache Design on Access-Address and Associativity

Access address	TLB lookup	Assoiativity		
			Time	Cache access time
			Size	Cache size
			Cost	Hardware cost
Virtual address	—	Direct-mapped	**Time**	$T_{tag} + T_{comp}$
			Size	$S \times B$
			Cost	H_c
		Set-associative	**Time**	$T_{tag} + T_{comp} + T_{mpx}$
			Size	$S \times B \times W$
			Cost	$(H_c + H_{mpx}) \times W$
Real address	Parallel	Direct-mapped	**Time**	$max\{T_{tag}, T_{TLB}\} + T_{comp}$
			Size	P
			Cost	$H_c + H_{TLB}$
		Set-associative	**Time**	$max\{T_{tag}, T_{TLB}\} + T_{comp} + T_{mpx}$
			Size	$P \times W$
			Cost	$(H_c + H_{mpx}) \times W + H_{TLB}$
	Serial	Direct-mapped	**Time**	$T_{TLB} + T_{tag} + T_{comp}$
			Size	$S \times B$
			Cost	$H_c + H_{TLB}$
		Set-associative	**Time**	$T_{TLB} + T_{tag} + T_{comp} + T_{mpx}$
			Size	$S \times B \times W$
			Cost	$(H_c + H_{mpx}) \times W + H_{TLB}$

Time Components

- T_{tag} : Tag-array access time
- T_{comp} : Compare time
- T_{mpx} : Time to select way
- T_{TLB} : TLB access time

Size Components

- S : Number of sets
- B : Block size
- W : Number of ways
- P : Page size

Cost Components

- H_c : Hardware cost of cache per way
- H_{mpx} : Hardware cost of way selector per way
- H_{TLB} : Hardware cost of TLB

they are associated, or else the cache must be purged every time an address space is switched[62]. We have chosen the tagging scheme.

As shown in Figure 2.3, the cache, along with the TLB not shown in Figure 2.3, supports up to 256 virtual-address spaces with 8-bit VID (Virtual-address space IDentifier). An operating system need not flush the the cache or the TLB on switching virtual-address spaces unless the number of the active spaces exceeds 256.

2.3.2.2 Combined solution to both the *synonym* problem and the multicache consistency problem

Virtual addressed cache also incur the serious problem of *synonym* when two or more virtual addresses map to the same real address. There are a number of possible ways to deal with the *synonym* problem[29]. Among them, we have chosen the dual-directory cache scheme proposed in [29].

The dual-directory cache scheme exploits the similarity of the *synonym* problem to the multicache consistency problem, and therefore gives a combined solution to both problems. Since both problems can be resolved in an integrated way to enforce cache coherence, we sometimes call them the cache coherence problems in the general. Otherwise, we particularly call the *synonym* problem the *intra-cache coherence* problem, and call the multicache consistency problem the *inter-cache coherence* problem.

As shown in Figure 2.3, one of the two directories is called V-tag array, which is normally accessed by the SPARC microprocessor with virtual addresses; the other is called R-tag array, which is normally accesses by the cache coherence logic in the MMU using real addresses.

2.4 Cache Coherence Schemes

The cache we have designed has two kinds of cache coherence to enforce: *intra-cache coherence* and *inter-cache coherence*. From the standpoint of a testbed, the system provides the various schemes of cache coherence as follows;

1. Cacheability marking scheme to enforce *inter-cache coherence* [9]

2. Fast selective invalidation scheme to enforce *inter-cache coherence*[16]

3. Distributed limited-directory scheme to enforce *inter-cache coherence*[5]

4. Dual-directory cache scheme to enforce both intra-cache and *inter-cache coherence*[29]

2.4.1 Cacheability Marking Scheme

The cache coherence problem can be solved easily if shared data, or writable shared data in particular, are not cached.

An attribute bit, called the *cacheable bit*, is provided in each page-table entry to specify whether data in the associated virtual page are cacheable or not. An operating system, with the help of a compiler, marks only virtual pages, which is mapped to either private space or common nonwritable space, as cacheable pages.

When a cache miss occurs on accessing a line of a noncacheable page, the MMU does not load the line into the cache. While the page is marked noncacheable, accesses to the page are always performed in either local or remote memory.

Although the scheme is easy to implement, the performance will suffer because shared-data accesses account for a large portion of common space accesses.

2.4.2 Fast Selective Invalidation Scheme

In recent years, several compiler-directed cache coherence schemes have been proposed [18]. Among those schemes, we have chosen the fast selective invalidation scheme proposed in [16, 18]. And, we have modified the scheme slightly to tailor it to our system. [39].

First, in addition to a *valid bit*, a status bit called the *change bit* is associated with each cache line as shown in Figure 2.3. Although the original scheme assumes a line size of one word, our cache has a different line size. As pointed out in [18], it is not a problem. The *change bit* is set when a line is loaded into the cache and reset by a cache control instruction.

Second, the original scheme uses two types of cache control instructions: a *Clear-Cache instruction* to reset all the valid bits and an *Invalidate instruction* to reset all the change bits. Although these functions are easy to implement, the SPARC instruction set cannot be augmented. We therefore have implemented them as user I/O commands to the MMU.

Last, the original scheme uses two types of load instructions: one is marked *cache_read* and the other is marked *memory_read*. Since the load instruction marked *cache_read* is

a normal load instruction, it does not cause any problem. On the other hand, the load instruction marked *memory_read* is not included in the SPARC instruction set. The instruction is defined as follows:

1. If the *change bit* of the line accessed is reset, it causes a miss to occur, and therefore the data are loaded from memory.

2. Otherwise, it is the same as a normal load instruction.

We cannot implement the load instruction marked *memory_read* straightforward. Instead we have added the following two user I/O commands to the MMU:

1. memory_read_on: This command causes the cache to treat all load operations as accesses marked *memory_read*. The cache hereafter checks the *change bit* of the line accessed.

2. memory_read_off: This command causes the cache to treat all load operations as accesses marked *cache_read*. The cache hereafter neglects the *change bit*.

Thus a load instruction marked *memory_read* will be replaced by a 3-tuple of memory_read_on, load, and memory_read_off instructions. Since the execution of two extra I/O commands introduces overhead, a compiler will be required to reduce the number of the commands. One approach might be an instruction reordering technique which schedules two or more load instructions marked *memory_read* near each other.

2.4.3 Distributed Limited-Directory Scheme

A number of hardware-based cache coherence schemes have been proposed [64]. Most of them are bus-based schemes which use the bus as a broadcast medium to ensure cache coherence [6]. Each cache "snoops" on the bus for broadcasted coherence commands to maintain coherence information in the cache. Most of these snoopy cache protocols can ensure sequential consistency [21]. In other interconnection networks (such as the crossbar network of our system), however, the network traffic caused by broadcast is unacceptable. Neither are the snoopy cache protocols scalable to a large number of processors (such as 128 PEs in our system).

Others are directory-based schemes which do not broadcast but (multi)cast coherence commands exactly to those caches having a copy of the memory block. These schemes

use a directory, either centralized or distributed, to track all cached copies of each memory block [5]. A directory entry for each memory block maintains pointers to identify the caches having a copy of the block. The directory protocols are classified into three categories: full-map directory, limited directory, and chained directory [3, 13]. Advantages and disadvantages of these protocols are reported in [13].

We have chosen the limited directory scheme to alleviate the directory size problem of full-map directory protocols and the complexity of chained directory protocols. The directory is distributed over all the PEs, as is the memory.

Each directory is placed at the SAS (SMW-Access Server) in the MU. We refer to it as the *coherence directory* (CD). The *coherence directory* is indexed using real addresses assuming that all shared-data are allocated in the common space of real-address spaces. Each directory entry for contains one pointer and two attribute and status bits: CC (Coherence Check) bit and CA (Cacheable/CAched) bit. The pointer specifies the PE having the cached copy of the memory block in its cache. Since the number of simultaneously cached copies is restricted to only one, the "*thrashing*" or "*ping-pong effect*" problem will occur if too many PEs share the block. The attribute bit, CC bit, is introduced to alleviate the problem.

For each directory entry, the CC and CA bits are reset and set by user I/O commands to the SAS. The CC and CA bits represent the attribute and status of the memory block as follows.

1. CC=0 and CA=0: The block is noncacheable.

2. CC=0 and CA=1: The block is cacheable but not subject to cache coherence enforcement. Consequently, more than one cached copy of the block are allowed to be present simultaneously.

3. CC=1 and CA=0: The block is cacheable and subject to cache coherence enforcement. A cached copy of the block is absent.

4. CC=1 and CA=1: The block is cacheable and subject to cache coherence enforcement. A cached copy of the block is present in the cache of the PE specified by the pointer.

The *inter-cache coherence* protocol based on the distributed limited-directory scheme is detailed in section 2.5.

2.4.4 Dual-Directory Cache Scheme

Dual-directory cache scheme ensures both *inter-cache* and *intra-cache* coherence. The *inter-cache coherence* protocol using this scheme and the distributed limited-directory scheme is described in section 2.5. The *intra-cache coherence* protocol with this scheme is described in section 2.6. Here we give some background information on the dual-directory organization.

The dual-directory organization is based on that proposed in [29], but it is tailored to our cache design [38]. Both directories contain a tag for each line in the data array. We assume that the page size is 4K bytes and that the block (line) size is 32 bytes.

One of the two directories, the V-tag array, is a direct-mapped tag memory of 2048 sets. It is normally indexed using 11 bits of virtual addresses (VA15-5). The index includes the least significant 4 bits of the virtual-page number (VA15-12). Each V-tag entry contains the following fields which are related to cache coherence (see Figure 2.3).

1. Virtual address tag: This field contains the most significant 16 bits of the virtual address (VA31-16).

2. Status bits: These are a *valid bit* and a *change bit* (see section 2.4.2).

3. R-tag pointer (RTP): This field points to the corresponding R-tag entry. The 4-bit pointer includes the least significant 2 bits of the real-page number (RA13-12) that identify the set in the R-tag array, and two bits that specify the way within the set.

The other directory, the R-tag array, is a 4-way set-associative tag memory of 512 sets. It is normally indexed using 9 bits of real addresses (RA13-5). The index includes the least significant 2 bits of the real-page number (RA13-12). Each R-tag entry contains the following fields, all of which are related to cache coherence (see Figure 2.3).

1. Real address tag: This field contains the most significant 18 bits of the real address (RA31-14).

2. *valid bit.*

3. V-tag pointer (VTP): This field points to the corresponding V-tag entry and the corresponding line in the data array. The 4-bit pointer is the least significant 4 bits of the virtual-page number (VA15-12) that identify the set in the V-tag array.

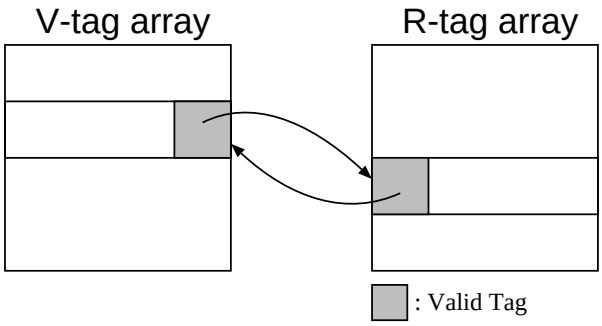


Figure 2.4: Double-Link between V-tag and R-tag

As implied by the description above, a pair of V-tag and R-tag must be doubly-linked using the RTP and VTP, if the corresponding line in the data array is valid (see Figure 2.4).

Although the V-tag array and R-tag array have the same number of entries, they differ in the associativity: direct-mapped for the V-tag array and 4-way set-associative for the R-tag array. The reason is given in section 2.6.2.

2.5 Hardware-Based Inter-Cache Coherence Enforcement

2.5.1 Protocols

The *inter-cache coherence* protocol is based on the distributed limited-directory scheme and the dual-directory cache scheme. Recall that a memory block is subject to cache coherence enforcement only when the CC bit in the corresponding coherence-directory entry is set. Thus we assume that the CC bit is set in the description below. *Inter-cache coherence* is maintained in the following way.

(a) Reads : When a read miss occurs in the cache of processing-element PE_j, PE_j sends a line-fetch message to the memory of PE_k. On receiving the message, PE_k looks up the CA bit and the pointer in the coherence-directory entry corresponding to the requested block. There are three cases according to the status.

1. CA=0: A cached copy of the block is absent. PE_k sets the CA bit and causes the pointer to point to PE_j. PE_k returns the requested data as a line-fetch-reply message to PE_j. When the reply is received at PE_j, the data are copied into the cache of PE_j. In this case, read misses result in two messages.
2. CA=1 and the pointer points to PE_j: The latest owner of the block is PE_j, but a cached copy of the block may not be in the cache of PE_j. PE_k just returns the requested data as a line-fetch-reply message to PE_j. When the reply is received at PE_j, the data are copied into the cache of PE_j. In this case, read misses result in two messages.
3. CA=1 and the pointer points to PE_i (see Figure 2.5-a): A cached copy of the block is present in the cache of PE_i. PE_k sends the real address of the block as a line-invalidate message to the cache of PE_i. PE_k makes the pointer to point to PE_j and returns the requested data as a line-fetch-reply message to PE_j. When the reply is received at PE_j, the data are copied into the cache of PE_j. In this case, read misses result in three messages.

In the case 3, PE_i invalidates its cached copy on receiving the line-invalidate message, as described later.

(b) Writes : We assume the write-through policy here. When a write miss occurs in the cache of PE_j, PE_j sends a write message to the memory of PE_k. Since write buffering is not provided, PE_j stalls until a write-reply message is returned from PE_k. On receiving the write message, PE_k looks up the CA bit and the pointer in the coherence-directory entry corresponding to the requested block. There are three cases according to the status.

1. CA=0: A cached copy of the block is absent. PE_k just returns a write-reply message. When the reply is received at PE_j, PE_j resumes. In this case, write misses result in two messages.
2. CA=1 and the pointer points to PE_j: The latest owner of the block is PE_j, and a cached copy of the block may or may not be in the cache of PE_j. The CA bit remains set. PE_k just returns a write-reply message. When the reply is received at PE_j, PE_j resumes. In this case, write misses result in two messages.

3. CA=1 and the pointer points to PE_i: A cached copy of the block is present in the cache of PE_i. There are two policies for maintaining *inter-cache coherence*: write-invalidate and write-update; we have chosen the write-invalidate policy for its simplicity. PE_k sends the real address of the block as a line-invalidate message to the cache of PE_i. PE_k resets the CA bit and returns a write-reply message. When the reply is received at PE_j, PE_j resumes. In this case, write misses result in three messages.

In the case 3, PE_i invalidates its cached copy on receiving the line-invalidate message, as described below.

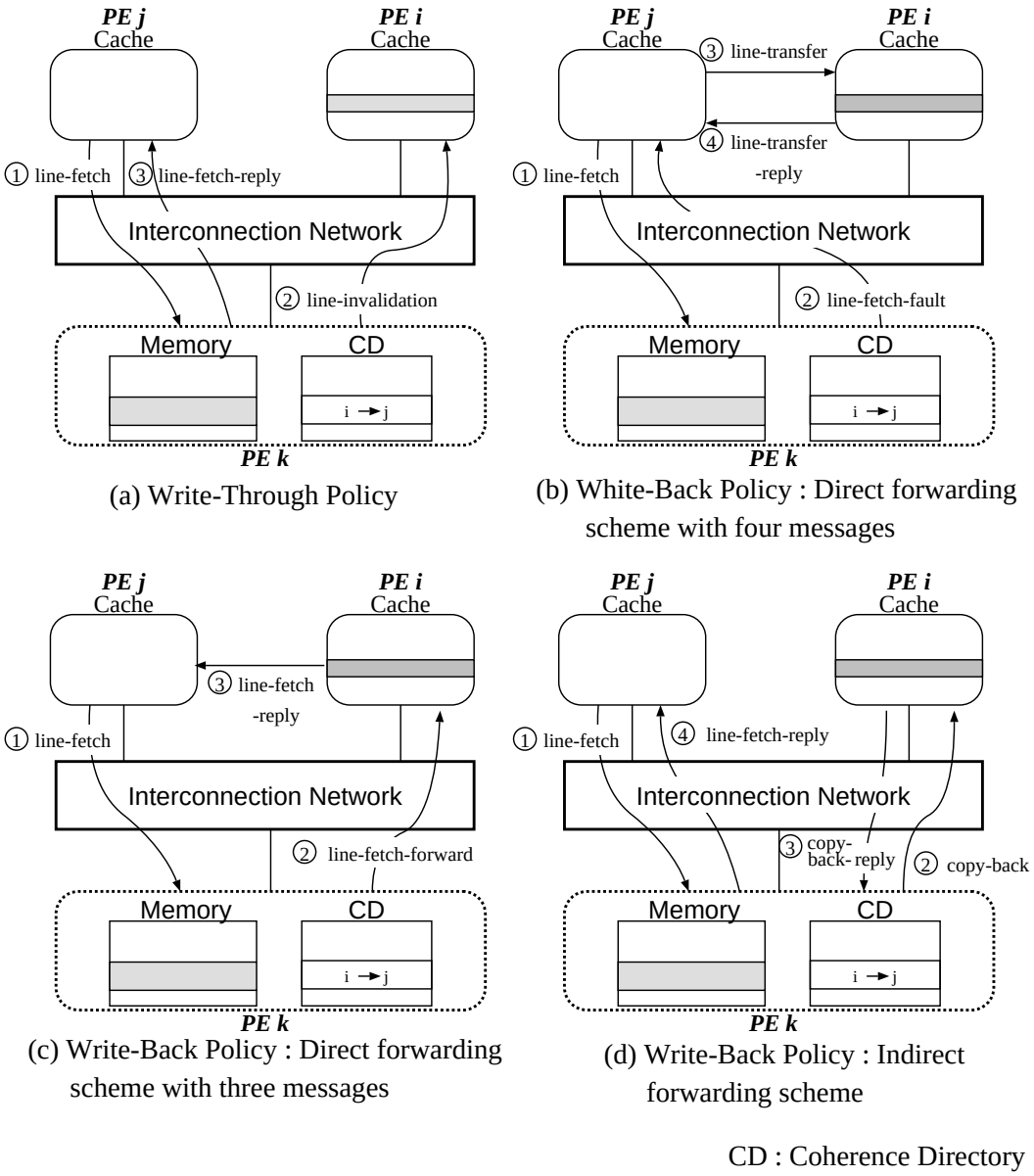
Line Invalidation : When a line-invalidate message is received at PE_i, PE_i looks up the R-tag array using the real address in the message. If a matching is found, then PE_i accesses the V-tag array using the VTP and resets the *valid bit* of the V-tag entry corresponding to the line to invalidate.

2.5.2 Write-Policy Issues

In section 2.5, we assumed the write-through policy. Consider the write-back policy for the case 3 of the reads above. In this case, the cached copy may be updated by PE_i and, if so, the status for the copy is dirty. Since the data of the memory block in PE_k may be stale, PE_k cannot simply return the data to PE_j. Instead, PE_k causes PE_i to forward the dirty copy to PE_j. The following three schemes are possible for the dirty-copy forwarding.

(1) Direct-forwarding scheme with four massages (see Figure 2.5-b) :
This scheme corresponds to the read-miss case of the SCI (Scalable Coherent Interface) coherence protocol. PE_k sends the pointer to PE_i, instead of the requested data, as a line-fetch-fault message to PE_j and makes the pointer to point to PE_j. When the pointer is received at PE_j, PE_j sends a line-transfer message to PE_i. On receiving the massage, PE_i returns the requested data as a line-transfer-reply message to PE_j and resets the *valid bit*. When the reply is received at PE_j, the data are copied into the cache of PE_j. In this scheme, read misses result in four messages.

(2) Direct forwarding scheme with three massages (see Figure 2.5-c) :
This scheme corresponds to the read-miss case of the SDD (Stanford Distributed-



(a) Write-Through Policy

(b) White-Back Policy : Direct forwarding scheme with four messages

(c) Write-Back Policy : Direct forwarding scheme with three messages

(d) Write-Back Policy : Indirect forwarding scheme

CD : Coherence Directory

Figure 2.5: Inter-Cache Coherence Enforcement with Coherence Directory

Directory) coherence protocol. PE_k makes the pointer to point to PE_j and sends the pointer as a line-fetch-forward message to PE_i. On receiving the massage, PE_i returns the requested data as a line-fetch-reply message to PE_j and resets the *valid bit*. When the reply is received at PE_j, the data are copied into the cache of PE_j. In this scheme, read misses result in three messages.

(3) Indirect forwarding scheme (see Figure 2.5-d) :

PE_k continues to intervene between PE_i and PE_j until the dirty-copy forwarding is complete. PE_k sends the real address of the block as a copy-back message to PE_i. On receiving the massage, PE_i returns the requested data as a copy-back-reply message to PE_k and resets the *valid bit*. When the reply is received at PE_k, the data are written into the memory of PE_k. PE_k makes the pointer to point to PE_j and returns the requested data as a line-fetch-reply message to PE_j. On receiving the reply, the data are copied into the cache of PE_j. In this scheme, read misses result in four messages.

The schemes 1 and 3 require four messages instead of three as in the case of the scheme based on the write-through policy. Although the scheme 2 requires only three messages, it would create potential deadlock dependencies, as would the scheme 3.

2.5.3 Weak Ordering

Weak ordering was first defined in [21] as follows.

Definition 1 : In a multiprocessor system, storage accesses are *weakly ordered* if

1. accesses to global synchronizing variables are *strongly ordered*, and if
2. no access to a synchronizing variable is issued by a processor before all previous global data accesses have been *globally performed*, and if
3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been *globally performed*.

Storage accesses are *strongly ordered* if all processors issue their accesses in program order, and if no access is issued by a processor until its previous accesses have been *globally performed*. A write is globally performed when its modification has been

propagated to all processors. A read is *globally performed* when the value it returns is bound and the write that wrote this value is *globally performed.*

However, the conditions imposed by Definition 1 are not necessary for the intuition behind weak ordering[60] and do not clearly specify any system model for software.

In order to facilitate a formal statement of the programmer's view of the system and not to specify unnecessary conditions for hardware, Adve and Hill introduced an idea of the *synchronization model*[1, 2], a certain set of software constraints on synchronization. And they proposed an new definition of weak ordering as follows.

Definition 2 : Hardware is *weakly ordered* with respect to *a synchronization model* if and only if it appears *sequentially consistent* to all software that obey the synchronization model.

Our system is *weakly ordered* by Definition 2 rather than by Definition 1, since the system violates all the conditions of Definition 1.

Although we should develop a synchronization model with respect to which our system can be weakly ordered, we do not address it further. Instead, we below outline the event ordering implemented by our inter-cache coherence protocol.

1. Intra-processor dependencies are preserved.

2. A read-write synchronization operation is executed atomically with respect to other synchronization operations on the *same* location.

3. All synchronization operations are treated as write operations.

4. The value of a write issued by PE_j can be dispatched as a return value for some read when it modifies the memory block. Thus the cached copy of the block in another processing-element PE_i may not be invalidated, although PE_i is requested to invalidate it.

5. No synchronization operation on some memory location is executed until all previous writes to *the memory module including the location* are *globally performed.*

6. The memory block requested by PE_j on a read miss cannot be served until all previous writes to *the memory module having the block* are performed with respect to PE_j.

2.6 Intra-Cache Coherence Enforcement

2.6.1 Protocols

The *intra-cache coherence* protocol is based on the dual-directory cache scheme. *Intra-cache coherence* must be preserved on every cache miss; both read and write misses, in the following way.

(a) Read miss : For a read miss, the protocol is defined as a part of the line replacement procedure as follows.

1. The virtual address causing the miss is translated into a real address through the TLB lookup. Once the real address is obtained, the *intra-cache coherence* enforcement proceeds in parallel with the line-fetch transaction.
2. The R-tag array is examined using the real address. If a hit occurs, the matching R-tag entry remains the entry for the new line, and step 3 is skipped. In this case, the virtual address corresponding to the real address could be a *synonym*.
3. Otherwise, an R-tag entry is selected for the new line. Recall that the R-tag array is 4-way set-associative; thus there are four candidates for a replacement "victim". The choice of a victim is done by comparing all the VTPs (V-tag pointers) of the four R-tag entries with the virtual address (VA15-12) causing the miss. If a matching is found, the matching R-tag entry will be a (false) victim. Otherwise, any R-tag entry will fall a (true) victim to "collision" (see Figure 2.6).
4. The *valid bit* of the V-tag entry doubly-linked to the R-tag entry is reset. Thus the double-link is broken, and the corresponding line is removed.
5. When the new line is fetched, the *valid bit* of the corresponding V-tag entry is set. Its RTP (R-tag pointer) is also made to point to the R-tag entry; similarly, the VTP (V-tag pointer) of the R-tag entry is caused to point to the V-tag entry. Thus a new double-link is created.

(b) Write miss : Although the write-through policy usually regards every write as miss, we below assume for a write miss for an absent line.

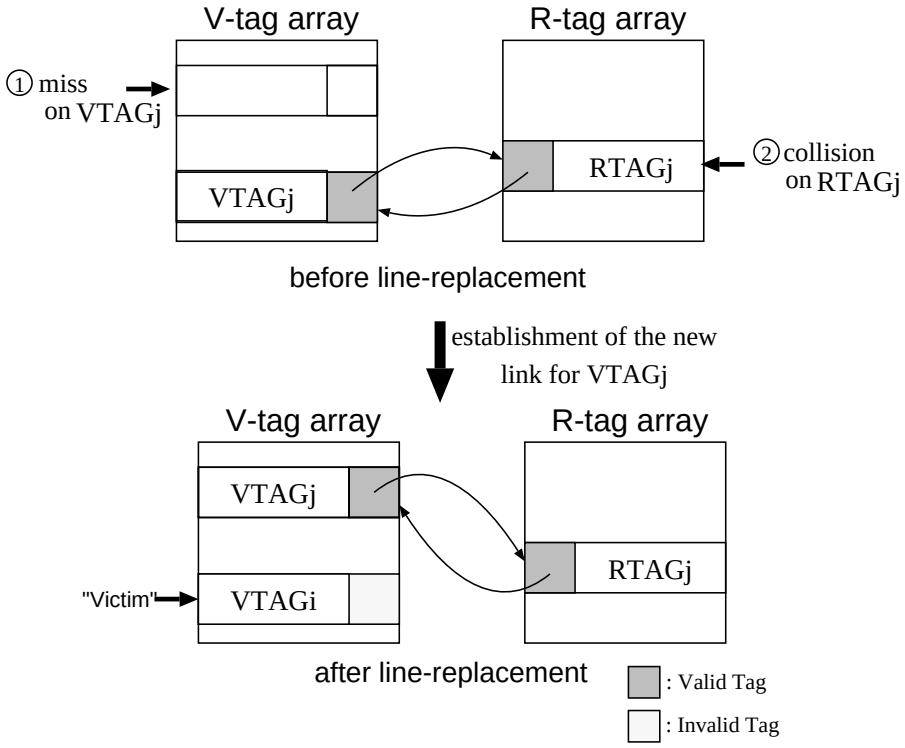


Figure 2.6: Collision problem

1. The virtual address causing the miss is translated into a real address through the TLB lookup. Once the real address is obtained, the *intra-cache coherence* enforcement proceeds in parallel with the write transaction.
2. The R-tag array is examined using the real address. If a miss occurs, the protocol is complete.
3. Otherwise, the *valid bit* of the V-tag entry doubly-linked to the R-tag entry is reset. Thus the double-link is broken, and the corresponding line is removed.

2.6.2 Collision Problem

In the step 3 of the protocol for a read miss, a replacement victim must be selected from a set of the R-tag array. Line replacement normally causes another victim to be selected

from a set of the V-tag array. If there is at least one line belonging to exactly the same pair of sets as the new line, the single line is enough to be purged from the data array. Thus the R-tag entry selected results in a "*false*" victim. Otherwise, two lines must be purged from the data array: one is replaced because its tag is purged from the V-tag array (see step 5), and the other is replaced because its tag is purged from the R-tag array (see step 4). The latter is called a "*true*" victim, because it falls the victim to an avoidable "*collision*" situation.

Intuitively no collision occurs when the R-tag array is fully associative. If the R-tag array is set-associative, the collision is caused by the difference between the *superset* size and the set size of the R-tag array. A superset is a collection of all lines attainable by specifying only those bits that are common to both the virtual and real addresses [29]. The number of lines within a superset (i.e., superset size) is determined by the number of bits of virtual-page number used to index the V-tag array. In our cache, the least-significant 4 bits of virtual-page number are used as an index, and therefore the superset size is 16 lines. On the other hand, the set size of the R-tag array is 4 lines; there is a difference between both sizes. Even for set-associative, if the R-tag is 16-way set-associative, a collision never occurs.

The effect of collision appears to depend on the associativity of the R-tag array assuming that the superset size is fixed. Obviously a direct-mapped implementation of the R-tag array will suffer the most detrimental effect.

We have analyzed this effect as shown in the next section, and finally, taking the hardware cost into consideration, we have chosen four as the associativity.

2.7 Analysis of the Collision Problem

We have examined the relationship between the associativity and the effect of collision using two methods: one is a probabilistic analysis using a Markov Chain based on the proposal in [34], and the other is a trace-driven simulation.

2.7.1 Probabilistic Analysis using a Markov Chain

In this section, we examine the effect of collision by a probabilistic analysis using a Markov chain based on the proposal in [34].

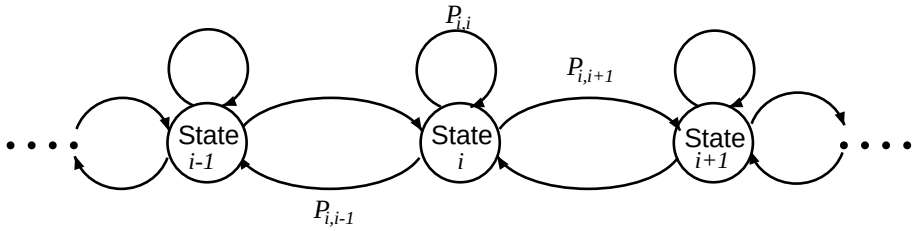


Figure 2.7: State Transition

We use the following notation.

E :number of total entries in the cache,

S :superset size which is the number of all lines attainable by specifying only those bits that are common to both the virtual and real addresses (it is also the ratio of the cache size to the page size),

n :associativity of the R-tag array($n \leq S$),

m_i :miss hit ratio in the virtual address cache of i entries, but having no (collision due to) R-tag array,

π_i :steady state probability of being in state i,

$p_{i,j}$:probability of state transition from state i to state j.

Figure 2.7 shows the state transition probabilities for the dual-directory cache. Cache miss causes a state transition. Assume that the cache is in state i. On a miss, the following three state state transitions are possible.

1. From state i to state $i+1$: If neither V-tag replacement nor R-tag replacement occurs, the state is incremented by 1. This probability is:
   $m_i \times \left(\frac{E-i}{E}\right) \times \left\{1 - \left(\frac{i}{E}\right)^n\right\}$.
2. From state i to state $i-1$: If both V-tag and R-tag replacements occur and these tags ware not doubly-linked to each other, the state is decremented by 1. This probability is:

$m_i \times \left(\frac{i}{E}\right)^{n+1} \times \left(1 - \frac{n}{S}\right)$.

3. From state i to state i: Otherwise, the state is unchanged.

We now get $p_{i,i+1}$ and $p_{i,i-1}$ as follows.

$$p_{i,i+1} = \pi_i \times m_i \times \left(\frac{E-i}{E}\right) \times \left\{1 - \left(\frac{i}{E}\right)^n\right\}. \tag{2.1}$$

$$p_{i,i-1} = \pi_i \times m_i \times \left(\frac{i}{E}\right)^{n+1} \times \left(1 - \frac{n}{S}\right). \tag{2.2}$$

In steady-state, the flow from state i to $i+1$ must be equal to that from state $i+1$ to i, namely $p_{i,i+1} = p_{i+1,i}$:

$$\pi_i \times m_i \times \left(\frac{E-i}{E}\right) \times \left\{1 - \left(\frac{i}{E}\right)^n\right\} = \pi_{i+1} \times m_{i+1} \times \left(\frac{i+1}{E}\right)^{n+1} \times \left(1 - \frac{n}{S}\right). \tag{2.3}$$

(i = 1,2,3,...,E-1)

Since m_i's are given and:

$$\sum_{i=1}^{E} n_i = 1, \tag{2.4}$$

we have a solvable linear system of E equations (Equation 2.4 and Equation 2.3) and E unknowns. Once the π_i's are known, the miss ratio with collisions is:

$$m = \sum_{i=1}^{E} m_i \times \pi_i, \tag{2.5}$$

while the cache utilization is:

$$\mu = \sum_{i=1}^{E} i \times \pi_i. \tag{2.6}$$

In our cache, E is 2048 and S is 16. We get m_i's from the results of the trace-driven simulation for a cache without R-tag array (see next section). Table 2.4 shows the miss ratio and the cache utilization ratio (μ/E) for $n = 1, 2, 4 and 8$.

This analysis concludes that as the associativity increases, better performance can be gained. Equation 2.2 state that no collision occur if $S = n$. The associativity of 16, however, might be costly and complex.

Table 2.4: Results by Probabilistic Analysis

	Single-Directory (w/o R-tag array)	Dual-Directory			
		W=1	W=2	W=4	W=8
Miss Ratio [%]	9.317	9.480	9.416	9.370	9.348
Line-replacement Ratio [%]	0.713	0.876	0.812	0.766	0.745
Cache Utilization Ratio [%]	100	49.15	63.16	78.09	86.42

W: number of ways of R-tag array

2.7.2 Trace-Driven Simulation

We presents the results of trace-driven simulations for the collision problem, using a dynamic trace of the Dhrystone programs executed on the Sun-4/110.

We assume the following conditions:

1. No line is invalidated due to *inter-cache coherence* enforcement.
2. No *synonym* exists.
3. Virtual addresses are translated into real addresses exchanging their most significant bits of set-address and their least significant bits of page numbers, and therefore the collision problem is the case.

Table 2.5 shows the results which are in agreement with that of the probabilistic analysis. The associativity of the R-tag array has more significant effects on the improvement in both the cache miss ratio and the cache utilization ratio.

2.8 Conclusion

We have presented the cache architecture of the reconfigurable parallel processor system and discuss the inter- and intra-cache coherence problems involved in the system. To ensure cache coherence and to measure the effectiveness of individual coherence schemes, the system provides the following schemes: (1) Cacheability marking scheme, (2) Fast selective

Table 2.5: Results by Trace-Driven Simulation

	Single-Directory (w/o R-tag array)	Dual-Directory			
		W=1	W=2	W=4	W=8
Number of Reference	500,000				
Number of Instruction Fetches	381,068				
Number of Data Read Accesses	75,913				
Number of Data Write Accesses	43,019				
Number of Line-replacement	3,567	12,933	4,209	3,762	3,675
Number of Misses	46,586	55,952	47,228	46,781	46,694
Number of Collisions	——	12,240	3,250	2,291	1,224
Miss Ratio [%]	9.317	11.190	9.446	9.356	9.339
Line-replacement Ratio [%]	0.713	2.587	0.842	0.752	0.735
Cache Utilization Ratio [%]	100	13.23	26.32	51.95	87.70

W: number of ways of R-tag array

invalidation scheme, (3) Distributed limited-directory scheme, and (4)Dual-directory cache scheme.

Both the cacheability marking scheme and the first selective invalidation scheme are compiler-directed inter-cache coherence schemes and their effectiveness relies heavily on the ability of a parallelizing compiler. IPC(Integrated Parallelizing Compiler) system and TINPAR(TINy PARallelizer) system[48], which have been in development at Kyushu University and Kyoto University respectively, are promising compilers for this end.

On the other hand, the distributed limited-directory scheme with the dual-directory cache scheme offers hardware-based inter-cache coherence protocols. The write-through policy we have chosen facilitates the protocols as compared with the write-back policy. Under the protocols the system is weakly ordered by the new definition of [2]. As future work, we will develop a synchronization model with respect to which the system can be weakly ordered.

The dual-directory cache scheme gives a solution to the intra-cache coherence problem, but it might suffer the collision problem if the R-tag array does not have sufficient associativity. We have analyzed the relationship between the associativity and the effect of

collisions.

An interesting problem for inter-cache coherence is how we combine the software-based approach with the hardware-based approach or how we cause both approaches to be share the burden of coherence enforcement. We have provided a method for enabling to make the tradeoffs between software and hardware approaches: the CC(Coherence Control) bit of each limited-directory entry for a memory block. A quantitative performance analysis will give us a promising direction on the software and hardware synergy to inter-cache coherence.

Chapter 3

The Distributed Shared Memory Multiprocessor : ASURA

Abstract

ASURA is a large scale, cluster-based, distributed, shared memory, multiprocessor. Up to 128 clusters are interconnected to form an ASURA system of up to 1024 processors. The basic concept of the ASURA design is to take advantage of the hierarchical structure of the system. Implementing this concept, a large shared cache is placed between each cluster and the inter-cluster network. The shared cache and the shared memories distributed among the clusters form part of ASURA's hierarchical memory architecture, providing various unique features to ASURA.

In this chapter, the hierarchical memory architecture of ASURA and its unique cache coherence scheme, including a proposal of a new hierarchical directory scheme, are described with some simulation results.

3.1 Introduction

Many multiprocessor systems have been developed and have now become widely available as commercial and research machines. However, there is still a strong demand for Massively Parallel Processing since there exist numerous applications (such as QCD, fluid dynamics, and many-body systems) which contain inherent parallelism that is too vast to solve on

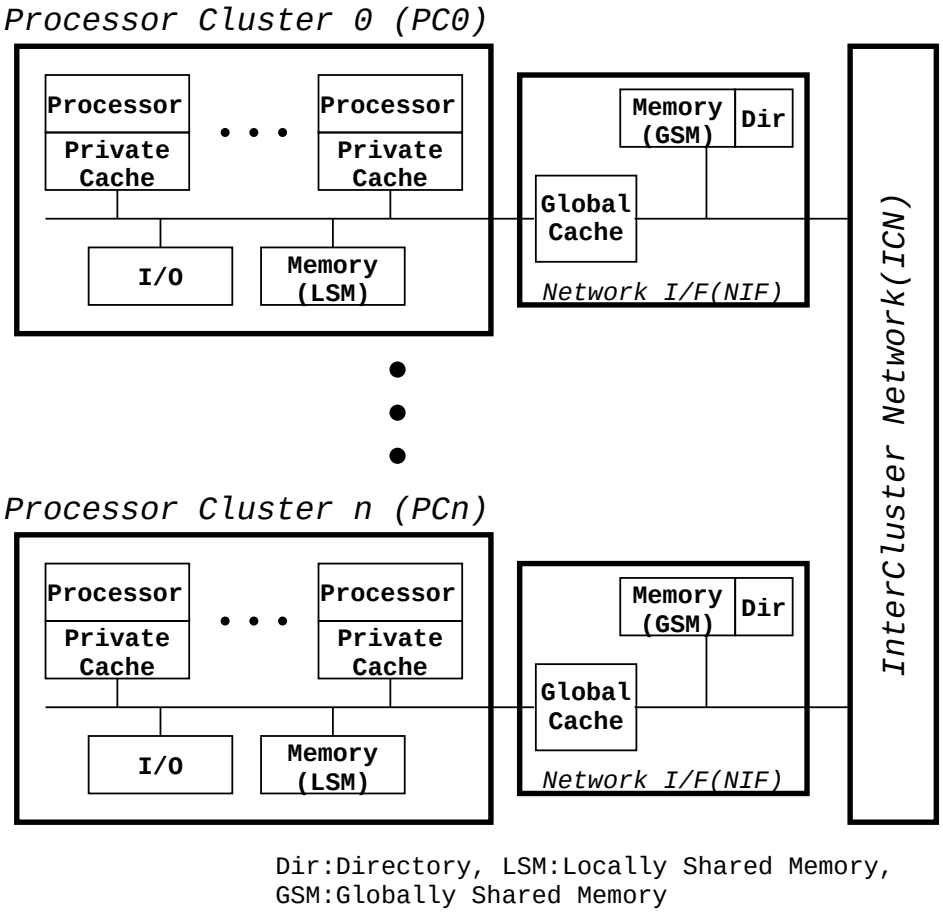


Figure 3.1: Overview of ASURA

available machines.

Working towards the goal of massively parallel processing, the cluster based multiprocessor is one of the promising approaches and many recent large scale systems, such as Cedar [40], DASH [42], Paradigm [19], Willow [8], and the Intel Paragon XP/S [72], have followed this approach. Another machine of this class, ASURA, is proposed in this paper.

The ASURA cluster is configured as a bus-based, shared memory multiprocessor with up to 8 processors. The clusters, interconnected through the Inter-Cluster Network (ICN), form a distributed shared memory multiprocessor. Figure 3.1 and Table 3.1 show, respectively, an overview of ASURA and the specifications of its prototype.

In order for ASURA to scale from 2 to 128 clusters, the ICN must also be scalable over a certain configuration range according to customer's requirements. Several candidate

Table 3.1: Specification of ASURA prototype

CPU	R4000MC (50MHz) 15MFLOPS(DP LINPAC)		
Number of CPUs per cluster			2, 4, or 8
Number of clusters			2∼128
Private Cache	Associativity		1(direct-map)
	Size		1MB∼4MB (SRAM)
	Line size		32B
	Write policy		Write-back
LSM	Size	each cluster	256MB
		full system	32GB
Global Cache	Associativity		4
	Size		8MB∼32MB (DRAM)
	Line size		1024B
	Write policy		Write-back
GSM	Size	each cluster	16MB∼4GB
		full system	2GB∼512GB

topologies for the ICN will be provided from which various configurations for the network can be chosen. In all cases, optical fiber is used as the medium of data transfer and, as a result, we can obtain a transfer rate of 400 megabytes per second for each communication link.

For ease of programming and of porting existing programs to the new architecture, ASURA provides a shared memory environment though the memory modules are physically distributed among the clusters. In order to implement a large scale, distributed, shared memory (DSM) machine efficiently, ASURA adopts a hierarchical cache architecture. This allows it to alleviate memory contention and decrease the latency of memory accesses simultaneously.

The key feature of ASURA is its shared cache and the design of its system-wide cache-coherence mechanism which takes into account the most suitable line sizes at each level of the hierarchy. Although several researchers [8][19][71] have discussed the issue of hierarchical cache design for large scale shared memory machines, little has been discussed about

how to deal with the line size difference between different levels of the cache hierarchy. Furthermore most of the previous researches have been done on centralized shared memory machines instead of DSM machines.

By adopting this hierarchical cache architecture, the intra-cluster architecture can be designed for fine grain parallel processing while the inter-cluster architecture can be targeted towards coarse grain parallel processing. Consequently, the ASURA architecture provides an effective parallel processing environment over a wide range of shared memory applications.

The next section explains ASURA's memory and cache architectures. Section 3.3 discusses how cache coherence is enforced in ASURA and a new directory based coherence mechanism is proposed for the hierarchical cache system. Some simulation results are shown in section 3.4. Section 3.5 contains a conclusion to this chapter.

3.2 Memory Architecture

3.2.1 Memory Organization

As shown in Figure 3.1, an ASURA cluster is basically a tightly coupled multiprocessor with centralized shared memory. Considering the system as a whole, ASURA has another level of shared memory which is distributed among clusters. Figure 3.2 shows the memory architecture of ASURA.

In ASURA, the first type of the shared memory mentioned above is shared only among the processors in the cluster and is called Locally Shared Memory or LSM. The other type of shared memory is accessible from any processor in the system and is called Globally Shared Memory or GSM. Thus ASURA's address space is partitioned into two spaces, private space and common space, corresponding to LSM and GSM respectively as indicated in Figure 3.3.

Although this kind of memory organization has been examined also in Paradigm [19], and Willow [8], the distributed style of GSM in ASURA is different from the centralized style of GSM in those machines. According to the notation of Willow, this kind of memory organization is referred to as *layered memory organization*. As long as the hardware cost is acceptable, the layered memory organization in ASURA has advantages over the flat, or non-layered, memory organization like DASH [42]. These advantages are listed below.

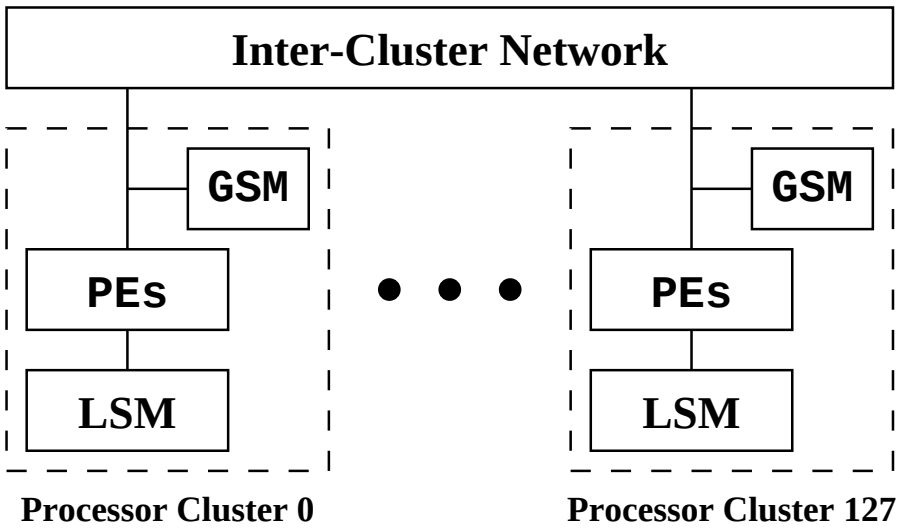


Figure 3.2: Memory architecture of ASURA

1. Hierarchical data sharing: Data which is local to a cluster, such as processor local variables, stack, code, and part of the system data, can be stored in the LSM. Moreover, one may allocate a copy of read only data, shared among many clusters, to the LSM of each cluster. For instance, image data for computer graphics applications, like surface data for ray-tracing and volume data for volume-rendering, can be allocated to the LSM. In addition, such data may be excluded from inclusion in the shared cache to maintain *the MultiLevel Inclusion (MLI) property*[1]. In this way, hierarchical data sharing allows the efficient use of GSM and thus increases the hit ratio of the shared cache.

2. Concurrency of memory accesses: By separating GSM and LSM, GSM in different clusters can be accessed without involving the intra-cluster bus on the GSM side. In other words, GSM accesses and LSM accesses can be performed concurrently at the same cluster.

3. Flexibility of LSM management: In ASURA, LSM management in one cluster is inde-

[1]Following the literature [7], "A MultiLevel cache hierarchy has the inclusion property if the contents of a cache at level $i+1, L^{i+1}$, is a superset of the contents of all its children caches, L^i, at level i."

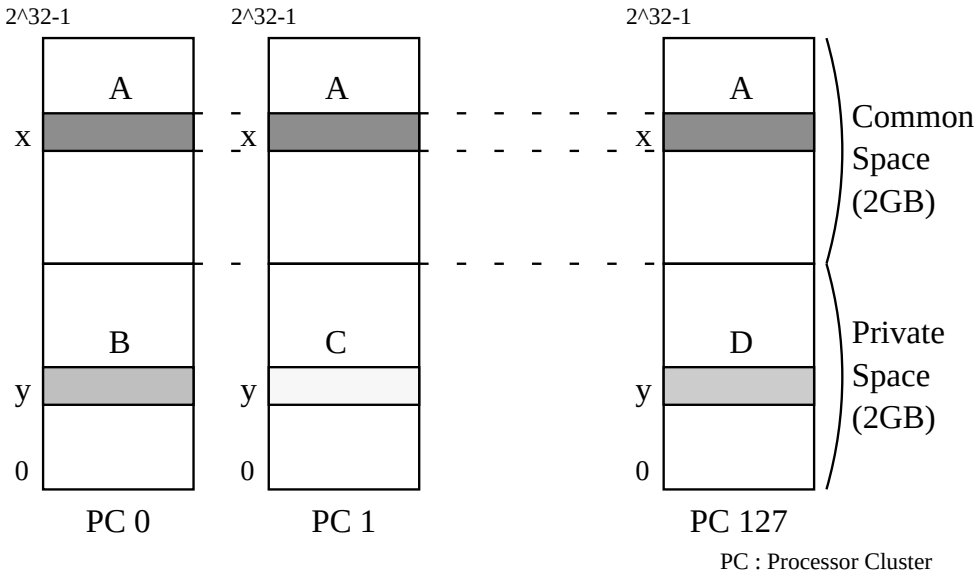


Figure 3.3: Memory map of ASURA prototype

pendent of LSM management in the other clusters. Thus, a cluster is not restricted in its LSM management policy. However, if the shared memory of each cluster is provided only as GSM, as in DASH, then when a cluster would like to enforce a memory management scheme for local data located in its shared memory, it must first negotiate with the other clusters.

4. Scalability in memory size: The private address space for each cluster is independent of the other clusters as shown in Figure 3.3, so the total LSM size over the whole system scales with the number of clusters, though the size is 32GB currently. However the maximum GSM size is fixed at 512 GB independently of the number of clusters.

Table 3.2 summarizes the memory architectures of various cluster based systems for comparison.

3.2.2 Cache Architecture

In order to offer a shared memory environment on a large scale multiprocessor like ASURA, it is necessary to simultaneously alleviate memory contention and decrease memory access

Table 3.2: The Summary of Memory Architectures of Typical Cluster Based System

	GSM	LSM	GSM Access	Cache Coherence Control	
				w.r.t. GSM	w.r.t. LSM
Paragon XP/S	No	Yes	–	–	?
Cedar	Yes	Yes	UMA	Cacheability control	
DASH	Yes	No	NUMA	Snoop-based(Intra-cluster) Directory-base(Inter-cluster)	–
Paradigm(node)	Yes	Yes	UMA	Directory	Directory?
ASURA	Yes	Yes	NUMA	Snoop-based(Intra-cluster) Directory-base(Inter-cluster)	Snoop-based

GSM: Globally Shared Memory NUMA: Non-Uniform Memory Access
LSM: Locally Shared Memory UMA: Uniform Memory Access

latency. To satisfy these requirements, cache memory plays an important role.

For this purpose, ASURA introduces a hierarchical cache architecture reflecting the hierarchy of the system structure, as shown in Figure 3.4. In this figure, and also in the following discussion, the on-chip primary cache together with the secondary cache is treated as a private cache for simplicity. Thus, in ASURA, not only is each processor configured with a private cache but also, each cluster contains a shared cache.

Each processor has a dedicated private cache, or PCache and it allows the caching of data from both the LSM and the GSM. The shared cache placed in the NIF (Network InterFace) of each cluster is shared by at most 8 processors and contains data cached from only the GSM. This shared cache is referred to as the global cache or GCache. In particular, the existence of the GCache distinguishes ASURA from DASH.

The major advantage of the GCache is that it isolates the cluster from the network. This enables the GSM to be isolated from the cluster-bus of the cluster. Then the cluster-bus can be optimized to access the LSM for fine grain parallel processing. On the other hand, for GSM accesses, the GCache provides an interface between the intra-cluster bus and Inter-Cluster Network(ICN). This isolates the differences in the communication protocol, transfer rate, cache line size between the two levels. This fact make it possible to optimize the cluster level architecture independent of the system level.

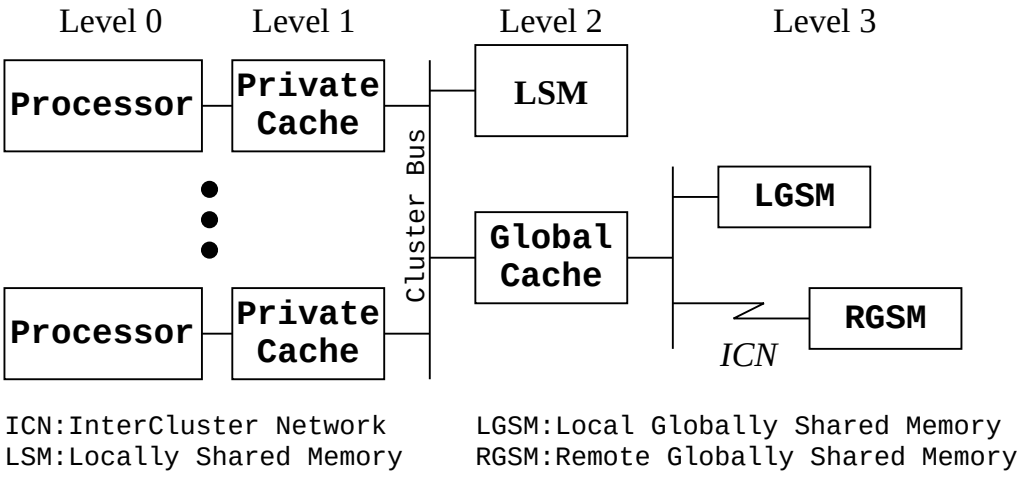


Figure 3.4: Memory hierarchy of ASURA

Several researchers [8][19][71] have discussed the issues of hierarchical cache design, however, little has been reported concerning the line size. In a hierarchical cache architecture, the line sizes at each level of the hierarchy need not be equal. On the contrary, they should be selected according to the architectural issues presented at each level. Hence the GCache line size has been enlarged, taking into account the other cluster level design factors. Note that the effect of enlarging the line size of the GCache is hidden from the PCache and the effect has no influence on the optimization of the intra-cluster architecture. This is because the *MLI (MultiLevel Inclusion) directory* , which is provided as a part of the coherence enforcement mechanism in the GCache, absorbs this effect. This is the most important feature of ASURA's hierarchical cache architecture.

The characteristics of the GCache are summarized below.

1. Up to 8 cache misses are serviced concurrently.
   ASURA adopts a split protocol for the cluster-bus. The memory access request and its reply are treated as separate bus transactions, therefore any processor can issue a memory access request before the previous requests from the other processors are completed. These access requests are served in a pipelined manner, and in order not to block the pipeline even after a cache miss occurs an eight-way bypass mechanism had been prepared in the GCache. Consequently, the GCache continues to serve consecutive

requests while doing line-fetches for the previously missed lines.

2. Large cache size.
   Since the access speed of the global cache need only be as fast as LSM, DRAM can be used to implement the data array. Thus, it becomes cheaper to include a large cache, which may be up to 32 MB in ASURA.

3. Long line size.
   ASURA adopts an unusually long line size of 1024 bytes for the GCache, while the line size of the PCache is 32 bytes. Hereafter, for clarity, the PCache line will be referred to as the *sub-line* and the GCache line as the *line* to distinguish between them.
   The merits of a long line size for the GCache are shown below.

   - Compensation for relatively long ICN latency: ASURA uses optical fiber as an ICN communication medium which provides a maximum transfer rate of 400 MB/s for each link. However, the long latency of the ICN, which is at least 1μs, prevents the effective transfer speed from achieving the maximum transfer speed of the link. The long line size of the GCache compensates for the latency and improves the effective transfer speed.
     Although the GCache line refill time increases in proportion to the line size (, see Table 3.3), this time can be hidden from PCaches by using the bypass buffer which passes the requested data to PCaches without waiting for the completion of GCache line refill. The effect of the bypass buffer is shown later in section 3.4.
   - Reduction of directory overhead: As discussed in the next section, to realize directory based cache coherence control for a large scale shared memory system like ASURA, the directory size should be less than a few percent of the total memory size. Since directory size is inversely proportional to the line size, or coherence block size, the long line size for the GCache helps to decrease the directory overhead in hardware cost.
   - Prefetching effect: Under data-parallel, for example `do-all`, type program executions, if a miss occurs in the private cache of one of the processors in a cluster, the other processors are also likely to miss on the same or nearby data. If the GCache line size is several times longer than the PCache line size then these misses can effectively be merged into one GCache line fetch. In this way, the long line size of the GCache provides a prefetching effect for the PCache, while providing the effects of memory

access combining for the ICN and GSM.

- Explicit cache management in terms of software: Explicit software cache management has been proposed in order to improve cache performance [8, 19]. However, the overhead of such software schemes makes it hard to apply them to caches with ordinary line sizes. The gain in effective transfer rate with the long line size of the GCache overcomes this overhead. Explicit cache management in ASURA is discussed in the next section.

Of course, there are some arguments against the long line size, one of which is the problem of false sharing. Some research has been done on this problem using both compile time static analysis approaches [24, 69, 30] and run time detection approaches [15, 23, 36]. Hopefully, the results of this research can be applied to ASURA to alleviate the false sharing problems.

3.3 Cache Coherence

3.3.1 Coherence Enforcement Strategy

In each of ASURA's clusters because writable, shared data may be cached into both the GCache and each of the processors' PCaches, cache coherence problems will occur and must be solved. There are basically two approaches, the hardware based approach [5, 64] and the software based approach [27, 9, 18].

Software based approaches become promising as the system size goes up because, in order to preserve cache coherence, they rely on self-invalidation of stale data at each processor and don't require any inter-processor communication for stale data invalidation. They are however still in the research stage, and are not yet sophisticated enough for practical use.

Therefore, a hardware based approach has been adopted in ASURA while leaving it possible for the software to assist with the hardware based cache coherence enforcement. Hence if we have a compiler which has the ability to manage the cache satisfactorily, or if the programmer offers information about data sharing, ASURA can use this information for coherence enforcement as will be discussed later in section 3.3.2.

In the development of ASURA's cache coherence enforcement strategy, the matters of intra-cluster and inter-cluster cache coherence enforcement are considered first, independent of each other. This is possible because the cluster level and the system level architectures

have been separated by the global cache. Then, they are united in a hierarchical fashion to enforce the coherence between PCaches and GSMs. In ASURA, this coherence is enforced by maintaining the MultiLevel Inclusion (MLI) property. This is referred to as cache coherence *hierarchical cache coherence*, in contrast with intra- and inter-cluster cache coherence.

Consequently, system wide memory coherence is maintained through these three kinds of coherence control in a hierarchical manner reflecting the system's structure.

The following two sections describe the intra- and inter-cluster cache coherence controls in more detail and section 3.3.1.3 describes how hierarchical cache coherence is preserved.

3.3.1.1 Intra-cluster Cache Coherence

Like ordinary bus based systems, ASURA adopts the snooping scheme. To enforce coherence between PCaches and either LSM or the GCache, the write-back write policy has been chosen.

The GCache appears as normal memory from the view point of intra-cluster cache coherence. Therefore, GCache and LSM can be treated as a single unit for implementing the intra-cluster coherence control, except that the GCache has to snoop the cluster bus in order to reflect modifications on PCache lines to the state of corresponding GCache lines.

In order to enforce intra-cluster cache coherence, one of the following states: INVALID, CLEAN (unmodified, possibly shared), and DIRTY (modified, no other copy), is assigned to each PCache line and the state is represented by the V and M bits of the PCache tag array (Figure 3.5(a)). Intra-cluster cache coherence is maintained in the following way (see Figure 3.6).

1. Read miss : The PCache which causes the miss puts a read request on the cluster bus. If any other PCache in the same cluster has a copy of the PCache line in state DIRTY, this PCache 1) responds the request by writing back the line to LSM or GCache, 2) releases the ownership of the line, and 3) sets its state to CLEAN. Otherwise, LSM or GCache responds to the request by putting corresponding data on the cluster bus. The requesting PCache picks up the corresponding data from the bus, puts it into its data array, and set its state to CLEAN.
2. Write miss : The PCache which caused the miss puts a write request on the cluster bus. If any other PCache in the same cluster has a copy of the PCache line in the DIRTY

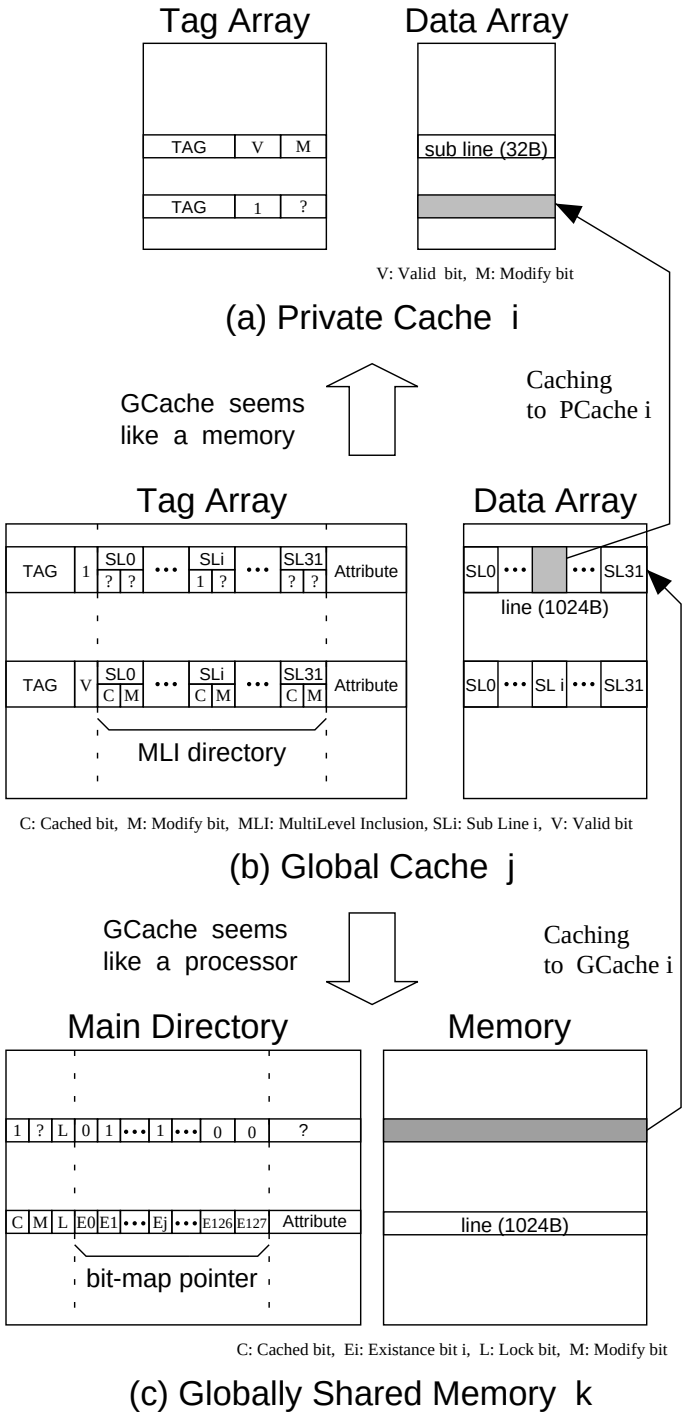


Figure 3.5: The Hierarchical Cache Coherence Enforcement Mechanism

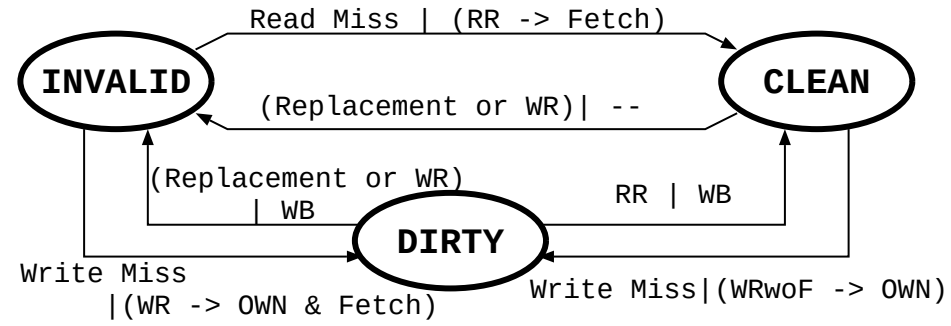


Figure 3.6: State Transition Diagram

state then this PCache 1) responds to the request by writing back the line to the LSM or the GCache, 2) transfers the ownership of the line to the requesting PCache, and 3) sets the state of the line to INVALID. Otherwise, the LSM or the GCache responds to the request by putting the corresponding data on the cluster bus. All PCaches other than the requesting one set the state of their copy of the line to INVALID. The requesting PCache reads the corresponding data from the bus, stores it into its data array in the DIRTY state. In any case, if the write miss is caused by a processor's write on a CLEAN line, no data transfer is made on the bus. By observing the ownership transfer on the cluster bus, the GCache knows the state of any of the sub-lines cached in a PCache.

3. Replacement : When the PCache needs to replace a line on a read or write miss, the PCache set the states of the corresponding sub-line to INVALID, and it is written back if the sub-line had been in the DIRTY state.

4. Read hit / Write hit : No coherence enforcement transaction is made.

3.3.1.2 Inter-cluster Cache Coherence

In this section, the coherence control between the GCaches and the GSM, and between the GCaches is considered. Directory based inter-cluster cache coherence is discussed here.

The directory schemes [12] can be classified as either full-map directory, limited directory, or chained directory.

For simplicity of control, ASURA uses a full-map directory. The drawback of using a full-map directory is directory overhead. In our case, this overhead is proportional to the number of GCaches and inversely proportional to the GCache line size.

However, the GCache line size of 1024 bytes, which is 32 times the PCache line size, makes it possible to reduce the directory overhead of ASURA to a fraction of what it would have been if the line size were the same size as the PCache line. At the same time, the width of the directory is kept narrow because the directory is managed in units of clusters, or GCaches, instead of processors, or PCaches.

So far, several interesting techniques have been proposed to reduce the directory overhead without enlarging the line size [10][31][44]. However, most of these techniques aren't adopted because they increase the complexity of the directory itself and this complexity may prevent high speed coherence control and increase time required to develop the machine.

The organization of the full-map directory is shown in Figure 3.5(c). This directory is referred to as the *main directory* in contrast with the *MLI directory* which is provided for the purposes of ensuring the MLI property and absorbing the effect of enlarging the GCache line size. Each *main directory* entry associated with a line is composed of three parts: 1) 3-bit status, 2) a bit map pointer of 128 bits, and 3) a field to specify the line attributes which are mentioned in section 3.3.2. The 3-bit status indicates 1) whether the line is cached, with the C bit, 2) whether the line is modified, with the M bit, and 3) whether the line is locked, with the L bit. The L bit is used as part of the synchronization mechanism for Locking. Each bit of the bit map pointer indicates if the corresponding GCache has a cached copy of the line or not. The attribute field contains the information not only on the characteristics of the line but also on how the line is manipulated. Such information is stored into this field by software, the operating system perhaps, when the associated page is allocated to a GSM.

3.3.1.3 Assurance of the MultiLevel Inclusion property

To guarantee the MLI property [7] in ASURA, all lines residing in PCaches in a cluster must be contained in either their parent LSM or their parent GCache, where the word "parent" means being closer to the memory in the memory hierarchy. At the same time, all lines residing in the GCaches must be included in the set of all lines in the GSMs.

The inclusion relation 1) between GCaches and GSMs, and 2) between PCaches and their parent LSM with respect to the PCache lines which come from the LSM, are always maintained naturally. Consequently, the inclusion relation that must be considered is that 3) between PCaches and their parent GCache with respect to the PCache lines which come from the GCache. Then only the third inclusion relation will be discussed hereafter.

Assurance of the MLI property requires a slightly stronger restriction than necessary for hierarchical cache coherence enforcement. For instance, from the view point of the hierarchical cache coherence enforcement, a PCache may contain data which does not exist in its parent GCache as long as the hierarchical coherence is maintained, though this situation is not allowed if the MLI property is to be assured. However, the advantages of the MLI property, like the filtering of inter-cluster coherence enforcement traffic, overcomes the disadvantage of this stronger restriction.

As a result, the coherence enforcement to keep the MLI property is initiated when a GCache receives 1) an invalidation or 2) a write-back request from GSM, and when a replacement of 3) clean or 4) dirty line occurs in the GCache.

In the first and third cases, where the data is not modified in this cluster, all of the sub-lines corresponding to the line are invalidated first, and then the GCache line is invalidated.

In the second case, all of the dirty sub-lines corresponding to the line are written back to the GCache first, and then the GCache line is written back to memory. The last case is slightly different from the second case in that it requires invalidations of the GCache line and its corresponding sub-lines after the line is written back.

For simplicity, in the above discussion of coherence enforcement, the difference between the line sizes of the GCaches and the PCaches has not been mentioned. Although the difference is not an essential part of assuring the MLI property, in practice, not only for the efficiency of the coherence enforcement but also for concealing the system level architecture from that of the cluster level, this difference must be taken into account.

As discussed in section 3.2.2, the line sizes of the GCache and PCache are 1024 bytes and 32 bytes respectively. 32 sub-lines with consecutive addresses belong to the same GCache

line. Therefore at most 32 sub-lines with different addresses have to be invalidated or written back when a GCache line is invalidated or written back. The sub-line invalidation requests, which are treated as *Write Request without Fetch*, can be issued in a pipelined manner and they theoretically don't need any acknowledgment. At most 32 invalidation requests need to be issued for an invalidation of a GCache line since all 8 PCaches snoop on the cluster bus. The sub-line write-back requests, which are treated as *Read Request*, can be issued in a pipelined manner as well, but each of them needs an acknowledgment or a write-back corresponding to each sub-line. This causes two more problems to arise.

To explain the first problem, it is assumed that the situation where a sub-line which belongs to a dirty GCache line is cached into PCaches (in state CLEAN) or is not cached into any PCache (in state INVALID). In this situation, there is no PCache which replies to the write-back request for the sub-line because there is no dirty sub-line corresponding to this request, so the GCache which issued this request will wait until a timeout occurs for an acknowledgment.

The second problem is not as serious as the first and is an efficiency problem. If all 32 invalidation requests are issued whenever a GCache line is invalidated or replaced then it may cause unnecessary traffic on the intra-cluster bus because some of the sub-lines are not cached at all.

These problems have originated due to the the lack of information about each sub-line. It would be possible to solve these problems if some history information for each sub-line were to be kept. For this purpose, a directory, which we refer to as the MLI directory, has been created and placed in the tag array of the GCache as shown in Figure 3.5(b). Each directory entry is created dynamically when the tag entry for the associated line is allocated. In order to track the status of each sub-line, the C bit and the M bit are provided for each sub-line.

The meanings of the C bit and the M bit are the same as the meanings of those in the main directory with the exception that they represent the status of a sub-line instead of a line. These two bits are updated whenever a PCache accesses a corresponding sub-line. When a GCache line is invalidated or written back, whether due to replacement or inter cluster coherence enforcement, by examining the corresponding MLI directory entry only the necessary coherence work need be done.

As well as a C bit and a M bit for each sub-line, each MLI directory entry has some bits to store the attributes associated with the line. These attributes are used for coherence

enforcement in the GCache.

Here, we should mention that the GCache is not required to be as fast as the PCache, and hence the overhead of the MLI directory itself and that of the directory management are not significant.

The organization of the MLI directory itself is quite similar to the sectored directory [10] except that the MLI directory does not need either a bit vector to track the PCaches which have a copy of the sub-line nor the pointers to the PCaches which have ownership (write privileges) of the sub-line. The reason why we don't need them is simply that the cluster is a bus-based architecture with a snooping based coherence scheme. However there are great differences between them. 1) In the MLI directory, the unit of data caching to the PCache is a sub-line, while it would be a line in the sectored directory. This characteristic makes it possible to offer transparency of the system level architecture, such as the line size of the GCache, to the cluster level architecture. 2) Each entry of the MLI directory is dynamically allocated in the tag array of the GCache when the corresponding GCache line is allocated, so that the directory overhead is proportional to the total cache size instead of the total memory size which may be a few tens, or rather hundreds, times larger than the total cache size.

3.3.2 Software Assist for Cache Coherence

As has been shown in the previous two sections, ASURA implements a hierarchical cache coherence scheme in hardware. However, with this scheme, inter-cluster communication is required to enforce cache coherence on the fly. In order to reduce this run time overhead, software can assist with the hardware coherence enforcement.

The advantages of software based coherence enforcement are derived primarily from the use of 1) knowledge about the characteristics of the data, and 2) knowledge of how the data is manipulated during the program execution, or the semantics of the data.

Firstly the typical case where software assistance can greatly reduce the run time overhead [59] is shown. Counter based barrier synchronization is examined, where the counter is decreased by one each time the process participating in the barrier reaches the synchronization point in its program. If hardware based cache coherence is enforced without software assistance, then network traffic to invalidate or update the line arises at each counter modification. For the processes which are awaiting the completion of the synchro-

nization, however, the intermediate value of the counter toward the completion is irrelevant. They only need to know when the value of the barrier counter reaches zero. Therefore all this network traffic except the last message, which signifies the completion of the synchronization, is unnecessary. If the hardware were informed of the semantics of the data in some way by software, this unnecessary traffic could be avoided. This is what the software is asked to do.

Hereafter, the manner in which software assist is implemented in ASURA is described. The basis of our software assist is the attribute marking scheme [9, 46, 8] where the attributes corresponding to the line guide the hardware coherence enforcement.

Other than the basic attributes like *cacheable*:allows caching of data, *non-coherent*: disables hardware coherence control, and *exclusive*: allows only a single PCache (or GCache) to hold a cached copy of data, the following attributes are presently being considered and implemented in ASURA by the addition of some additional hardware. A line may be assigned several of these attributes simultaneously.

1. *Update on write-back* : When a line marked with this attribute is written back from a GCache, the main directory updates the other cached copies of the line, and thus the corresponding PCache sub-lines too. For data with one writer and many readers this avoids extra cache misses. A broadcast option is also available for this attribute. Note that this is not like a *reflective memory system* [45] where the update is enforced whenever a processor modifies data; ASURA enforces the update only when a GCache line is written back to memory.
2. *Resident* : is used to force a specific line to reside in the GCache. Once a line marked with this attribute is fetched into the GCache, this line will never be replaced unless it is forcibly invalidated with the GCache I/O command mentioned later.
3. *Store Allocate No Fetch* : When a write access to a line marked with this attribute causes a miss in a GCache, the GCache allocates an entry for this line and requests the GSM to acquire the ownership of the line, but does not require the GSM to supply the data. In order to reduce unnecessary data transfer through the IPN, this attribute can be added to a line which a cluster completely overwrites without any previous reference to the line.
4. *Out-of-MLI* : allows a PCache to hold copies of GSM data regardless of the existence of corresponding GCache lines in its parent GCache. All coherence enforcement requests related to lines with this attribute have to be forwarded to cluster buses even if the

GCache does not have an entry for the corresponding line.

5. *Barrier Synchronization variable* : can be thought of as a variety of the *update on write-back* attribute. The difference is that, instead of updating the GCache lines and the corresponding PCache sub-lines at every occurrence of write-back, it updates them when the GSM storing the line has received the specified number of writes to the line[59].

These attributes are stored into corresponding attribute fields of the main directory (Figure 3.5(c)) first and are copied into attribute fields of the GCache tag array (Figure 3.5(b)) when the line is fetched. Then these attributes are used as a guideline both by the GCaches and by the main directory whenever coherence enforcement actions are required.

In addition to the attribute marking scheme mentioned above, ASURA provides the software (or the programmer) the means to manage both GCaches and GSMs explicitly in the form of I/O commands to the GCache. Currently, the following I/O commands for data transfer [11], cache coherence [18], and synchronization [42] are provided.

- *MOVE* command: forces a pre-fetch or a write-back on the specified line depending on the argument of the command. This command can be used to hide memory access latency.
- *Line-transfer* command: performs data transfer from a specified source address to a specified destination address through the GCache where this command is initiated. The length of data to be transferred is fixed as the GCache line size. The data transfer is performed as follows: 1) Invalidate the GCache line associated with the destination address if it exists in the GCache. Note that the write back is not needed even if the line is in the DIRTY state. 2) Fetch the source line into the GCache if the GCache does not contain the line. 3) Change the tag address of the line associated with the source address with the destination address, and inform the main directory of the change if needed. If the source line was in the DIRTY state it must be written back before changing the tag address. 4) Write back the destination line if needed. It is recommended that the *store allocate no fetch* attribute is assigned to the destination line. Since the line-transfer uses a tricky technique of operating on the GCache tag array, data transfer can be performed without moving the data at all if the GCache contains the source line in CLEAN state. This command is well suited for the copy-on-write operation.
- *Invalidation* command: invalidates the specified line by invalidating or if needed, forcing a write-back before the invalidation, and makes the associated GCache entry empty.

- *Lock/Unlock* command: provides GCache line based synchronization primitives. ASURA supports a *queue based lock* mechanism, as well as DASH, where the existence bits of the main directory are used to trace clusters waiting the acquirement of the lock. *Lock/Unlock* commands combined with *MOVE* command could support a rather complex command which forces a pre-fetch of a line when the Lock associated with the line is acquired or forces a write-back of a line when the Lock associated with the line is released.

3.4 Performance Simulation

3.4.1 Simulation Environment

ASURA's performance has been investigated by the use of trace-driven simulation. As the trace data, transaction sequences on each cluster bus were used, which are composed of the time, address, and the access type for each transaction, instead of the address traces for each processor.

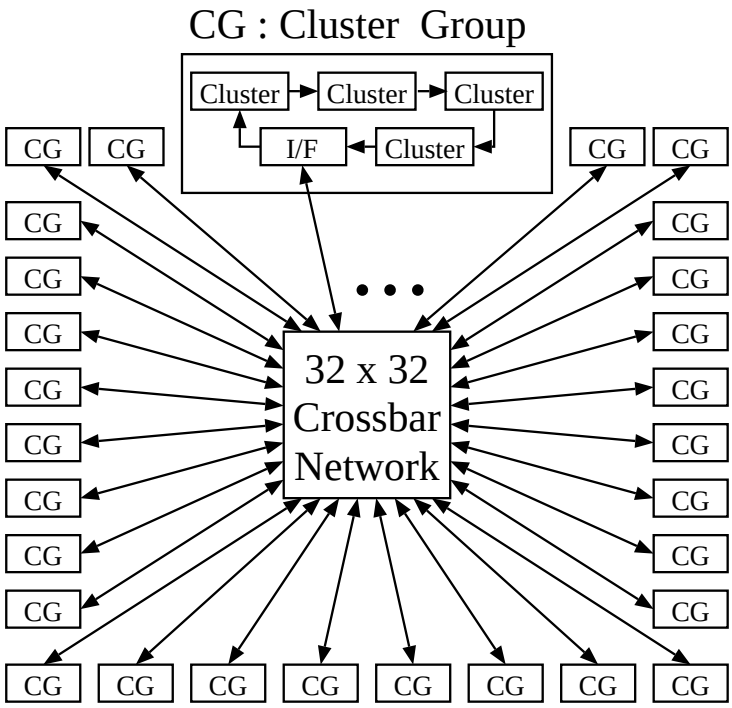
Although various simulations have been done [26][56], here the scalability of ASURA and the effect of the long GCache line size is focused upon.

Matrix multiplication ($C = A \times B$) and LU decomposition ($A = L \times U$), where A,B,C,L and U are matrices of size 1024×1024, have been chosen as sample programs.

All matrices are initially distributed among the GSMs so that their sub-matrices, of form $MATRIX[nI + i, *]$ {n | 0,1,...,1024/I} where I: Number of Clusters, are allocated to the GSM of cluster i.

In order to extract cluster level parallelism, the*data owner compute rule* is applied to the code. To extract fine grain parallelism inside each cluster, the code was further optimized so that it could fit within the sub-line size of a PCache.

Among the various available ASURA configurations, the configuration shown in Figure 3.7 was used throughout these simulations, where the number of processors in a cluster is fixed at 8. A cache size of 8 KB for the primary on chip cache, 2 MB for the secondary cache, and 16 MB for the GCache and the memory access latencies shown in Table 3.3 were assumed. The memory access latencies were derived from a hardware logic simulator.



Note 1 : Number of Clusters in a Cluster Group is in the range from 1 to 4. Clustes inside a CG are interconnected by a ring network.
Note 2 : In case of 16 clusters organization, a 16x16 crossbar network which is a quarter of the 32x32 crossbar network is used.

Figure 3.7: A ring-crossbar configuration of ASURA

Table 3.3: Memory Access Latencies

Memory		Latency [ns]
Private Cache	Primary	10
	Secondary	200
Global Cache		1500
GSM	Local Cluster	2000 (+ 2.5 × n)
	Local Ring	2500 (+ 2.5 × n)
	Remote Ring	3000 (+ 2.5 × n)

n: GCache line size [Byte]

3.4.2 Simulation Results

Figure 3.8 shows the simulation results both for matrix multiplication(**MATMUL**) and for LU decomposition(**LU**), where (a) and (c) show performance versus the number of clusters, and (b) and (d) show performance versus the GCache line size.

In the case of **MATMUL**, the effect of the software prefetching(**PF**) and of the GCache bypass buffer(**BP**) were examined. When the GCache bypass buffer is used the requested data passes through the GCache without waiting the completion of GCache line refill. As shown in Figure 3.8(a), the basic performance scales well up to 64 clusters. Using **BP**, scalability was confirmed up to 128 clusters. The beneficial effect of **PF** was also confirmed. By using **BP** together with **PF**, ASURA could achieve a performance of 9.6 GFLOPS with 128 clusters. This is about 100 times the performance of 93 MFLOPS which is achieved by a single cluster.

For **LU**, the effect of broadcasting(**BC**) was examined. This was simulated by setting the *Update on Write-Back* attribute and explicitly executing the *MOVE* command. The effect of **BP** was also examined. In these simulations (see Figure 3.8(c)), performance increases with the number of cluster until about 32 or 64 clusters. It then starts decreasing with any further increases in the number of clusters. It is assumed that this is because the amount of computation becomes too small compared to the amount of communication required.

For both applications, as shown in Figure 3.8(b) and (c), the performance scales up with the increase of GCache line size. We also confirm that, the larger the line size becomes, the more the effect of **BP** becomes significant because the time spent to transfer a line becomes dominant factor of memory access latency.

From these results, it is concluded that the ASURA architecture and its uniquely long GCache line size are justified as long as the applications have enough parallelism to exploit both system-level coarse grain parallelism and cluster-level fine grain parallelism.

3.5 Conclusion

An overview of ASURA, a cluster based multiprocessor system has been given. ASURA offers a shared memory environment though Globally Shared Memories (GSMs) are distributed among the clusters. For the efficient use of distributed shared memory, ASURA is

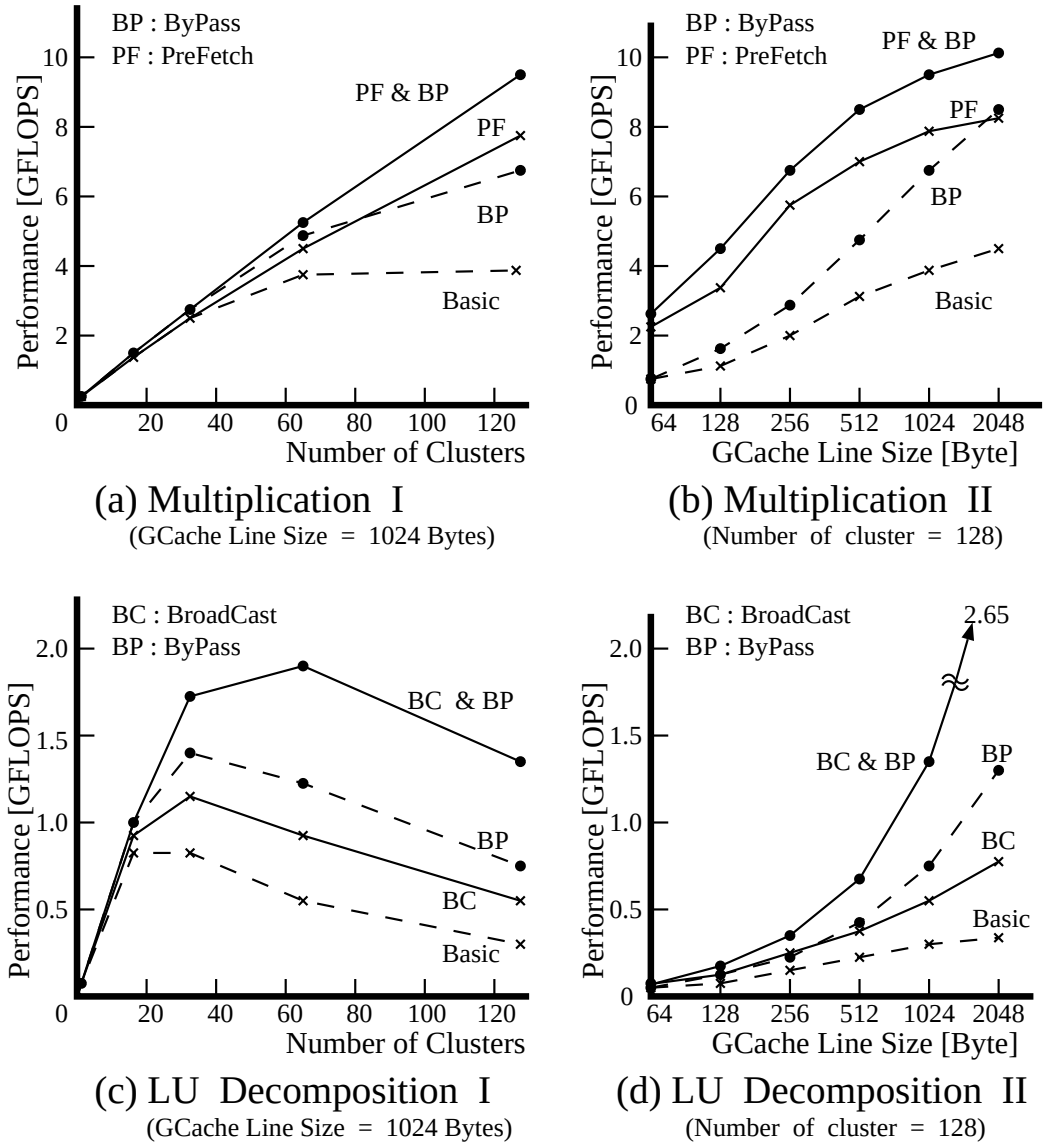


Figure 3.8: Simulation Results

configured with a shared cache, the Global Cache (GCache), on each cluster as an interface between the cluster and the InterCluster Network (ICN). Along with the Locally Shared Memory (LSM), provided independently of the GSM, the GSM and the GCache form a layered memory architecture.

As a result of providing a GCache on each cluster, it becomes possible to consider the inter-cluster and intra-cluster architectures independent of each other. This is also true for cache coherence enforcement. In order to enforce intra-cluster cache coherence, a snooping mechanism is provided for each processor. To enforce inter-cluster cache coherence, assuring the MLI property, a new hierarchical directory scheme has been developed where the directory is composed of two separate directories corresponding to the memory hierarchy: the main directory for the GSM and the MLI directory for the GCache.

In addition, ASURA provides software the means to improve the efficiency of cache coherence enforcement. The property labeling scheme allows the software to set cache line attribute which are used for coherence enforcement, while several I/O commands are available for software to manage the GCache directly.

Finally, simulation results have been presented that confirm that the ASURA architecture could offer a wide range of scalability and it could achieve a performance level of 9.6 GFLOPS in a 128 cluster configuration. It was also confirmed that the long GCache line size of 1024 bytes is justified for some large scale numerical computation.

Chapter 4

Self-Cleanup Cache

Abstract

An improved Write-Back(WB) cache which we call Self-Cleanup Cache:SCC is proposed in this chapter. The SCC aggressively enforces write-backs by itself to update memories. In this way, the SCC keeps itself as clean as possible, while preserving the write merge effect of WB cache.

The SCC is originally designed to efficiently implement the invalidation based Compiler Aided Cache Coherence(CACC) scheme on the multiprocessor system which configured with WB caches. However, even for the systems which adopt directory based cache coherence scheme, the SCC has a good effect on decreasing the read latency.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are examined by simulation upon two cache system models: 1)the system with the SCC and the selective invalidation scheme which is one of the CACC schemes, and 2)the other system with the SCC and the full-map directory scheme, respectively.

The simulation results show that 1) the effect of the write-back distribution appears as the decrease of the stall time at synchronizations, and thus improves the performance of the selective invalidation scheme, and 2) the aggressive write-back decreases the read stall time though it may increase unnecessary write-back transactions. It is also confirmed that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

4.1 Introduction

Rapid increase of the processor speed nowadays has been leading to a wide gap between processor speed and memory speed. In order to exploit the processor's maximum performance, it is required to narrow this gap. The cache memory plays an important role for this purpose. It is not too much to say that the recent high performance RISC processors can not work without the cache memory.

On the other hand, in the large scale shared memory multiprocessors, the more the size of the system scales, the more the memory access latencies increase. The latency hiding with cache memory hence becomes much more important. Furthermore, toward the massively parallel processor, it is indispensable to develop techniques to reduce the latency as well as the latency hiding technique.

Most of the researches on the cache memory so far have been directed toward latency hiding techniques like that to reduce the miss ratio from the standpoint of cache organization[57], and it has been less studied how to reduce the latency itself. The *write-buffer* approach for the Write-Through(WT) cache to reduce the write latency is one of those that have been well studied by now. Together with some appropriate memory ordering model [21, 22, 28], the *write buffer* helps to reduce the write latency even for the Write-Back(WB) cache.

However, the latency which dominates the processor performance is not the write latency but the read latency, because the processor essentially can not proceed its program execution, on the read miss, until the required data is supplied. Although both hardware-based[62] and software-based[11] prefetching techniques have been proposed to decrease this read latency, they don't work well if the latency is too large.

In the Write-Back cache under the directory based cache coherence scheme particularly, if the memory does not contain up-to-date data, the read latency becomes 1.5 to 2 times longer than the case where the memory has up-to-date data. This is because of the write-back overhead inherent in the write-back cache.

In order to decrease such read latency in the write back cache, we propose a new cache system with self-cleanup control, which we refer to as Self-Cleanup Cache(SCC). Although the SCC is originated in Write-Back cache, by means of self-cleanup control, the SCC can keep itself as clean as possible, like the Write-Through cache, while maintaining the write merge effect of the Write-Back cache. In order to realize this, we introduce a sort of working

set to manage the dirty line inside the cache. By means of enforcing the write-back of dirty line when it is forced out from the working set, the SCC reduces the read latency for the other processors. In some sense, it can be said that this is a proposal of a new write policy which has the characteristics intermediate between the write-back and the write-through policies.

The next section of this chapter describes the background of this research, including the motivation of this proposal. In section 4.3, the concept of the Self-Cleanup Cache is given with an implementation example. In section 4.4 and 4.5, overview of simulation environment and simulation results are presented to demonstrate effectiveness of the SCC.

4.2 Related Works

4.2.1 Compiler Aided Cache Coherence Control

Directory based cache coherence schemes become a conventional approach to the cache coherence problem in recent distributed shared memory machines. With the directory scheme, however, the run-time overhead to enforce the coherence increases rapidly and becomes unacceptable as the system scales up. This fact has motivated to devise software assisted cache coherence scheme such as the cacheability marking scheme[9] and self-invalidation scheme [16, 17, 47, 18, 15, 23, 51, 36] , which is referred to as Compiler Aided Cache Coherence(CACC) schemes in this chapter.

Most of the CACC schemes rely on self-invalidation of the stale data by each processor and require the main memory to have up-to-date data at certain points, which we call *synchronization points*, of program execution on each processor [1]. Therefore they don't suit well with the ordinary write-back cache, because the up-to-date data can be contained not in memory but in cache.

The Self-Cleanup Cache (SCC) has been originally designed to implement the CACC schemes on the system with the Write-Back cache.

[1]Based on the static analysis by a compiler, instructions to invalidate the potentially stale cache lines are inserted at some kind of synchronization points, such as parallel loop boundaries or the entrances and the exits of the critical sections. When the program is executed, the processor will issue instructions to invalidate cache lines and the invalidated copies will be updated during subsequent read misses to these cache line.

In order to apply the CACC scheme to the system with Write-Back cache, at the synchronization points, all up-to-date data have to be written-back to guarantee that memories contain the latest data with respect to the processor, prior to the invalidation. Although several invalidation techniques [16, 17, 47] can be applied to this situation, the efficient means for the write-back are not developed yet. In order to develop the efficient means, we must consider the following two issues associated with the write-back.

- Which lines should be written back?
  Without any cooperation from hardware coherence mechanisms like directory or snooper, there is no way to know immediately which cache lines are in the dirty state. The only way which we can do to find the dirty lines is the exhaustive search over the cache. Of course, there is yet another approach which relies on the compiler to explicitly insert the codes to force the write-back one by one, based on the complete static analysis taking the cache into consideration. Even in the later case, it is still not so efficient without any assistance from the hardware. Thus it is required to develop the hardware to cope with this issue.

- When should the cache write dirty lines back to memory ?
  If processors start writing all the dirty lines back to memory when they reach to any synchronization point, the network traffic increases rapidly. Furthermore, these write-backs start simultaneously at many processors, particularly in DOALL type parallel processing. This may cause network saturation which brings incredible performance degradation. In order to prevent such situations, we should develop some techniques to scatter the write-back traffics over the time.

One solution to these problems is the Self-Cleanup Cache (SCC) proposed in this chapter. By satisfying the following requirements, the SCC makes the invalidation based CACC scheme applicable to the system with write-back cache.

Requirement 1 Keep track of the addresses where a processor made write accesses.

Requirement 2 Write-backs of lines at appropriate points distributed over a period during program execution.

4.2.2 Latency Tolerance

Latency Tolerance is very important issue in the development of memory system. Weak form of memory ordering models[21, 22, 28] are techniques to hide memory access latency. By means of

1. overlapping of processor execution and memory accesses,

2. buffering of memory accesses, and

3. concurrent accesses which are essentially no need to be serialized,

the weak form of memory ordering model provides the following effects.

- Latency hiding by means of background processing of coherence enforcement allowing the temporal inconsistency, and
- Throughput improvement through the pipelining of write accesses.

However, in spite of the advantages of the ordering models, in the system with Write-Back cache, there exist some situations where any ordering model do not either help to hide nor reduce the latency. These situations are due to the issue that memory may not hold the latest data.

Figure 4.1 demonstrates such situation, where a processor Pi causes a read miss hit on a line A which is previously cached in another processor Pj in the dirty state. In this case, the read access takes the latency of roughly 2 times[2] longer than that for the case where the memory has up-to-date data.

However, if the system has ability to satisfy the following requirement, the system will suffer such long read latency less than the system without this ability.

Requirement 3 Dirty lines which are not likely to be accessed any more are written back to memories as soon as possible.

This is what we exactly do in the Self-Cleanup Cache, so that it could hide the latency of write-back and could reduce the latency of cache refill, in cooperation with the buffering and pipelining.

[2]If the cache has a facility of direct cache-to-cache transfer [42], it will be 1.5 times.

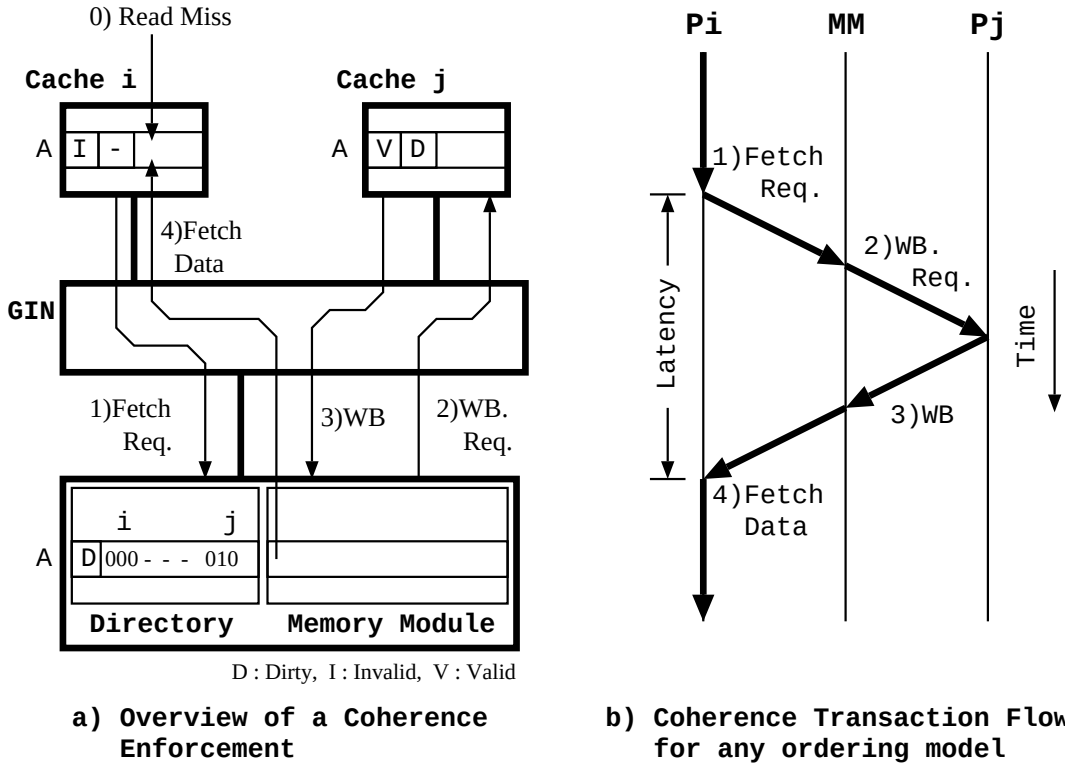


Figure 4.1: Read Latency on Write-Back Cache.

4.3 Self-Cleanup Cache

4.3.1 Overview of the Self-Cleanup Cache

The Self-Cleanup Cache(SCC) has been proposed

- to realize fast and efficient CACC control over the system configured with WB cache, and
- to reduce the write-back overhead inherent to the write-back cache, particularly for the system which adopts a sort of loose memory ordering model.

As shown in Figure 4.2, SCC is composed of two units: an ordinary write-back cache unit and an additional Self-Cleanup Control unit. Self-Cleanup Control unit keeps track of the addresses of the dirty lines in its partner cache and forces the cache unit to write-back

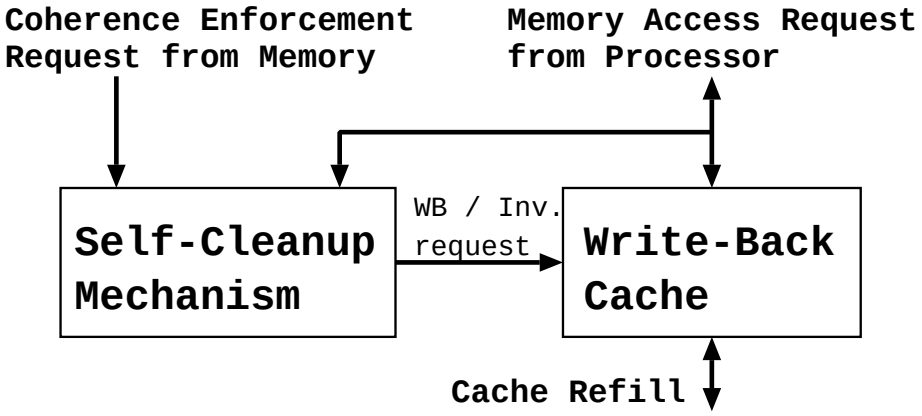


Figure 4.2: Organization of SCC

by informing when and which line should be written back. By requesting the write-back of an appropriate line at reasonable time, those requirements described in section 4.2 are satisfied.

In this way, the Self-Cleanup Cache holds the feature of write aggregation on Write-Back cache, while keeping both cache and memory as clean as possible. And consequently SCC can reduce the memory access latency as we have discussed in the previous section. In some sense, it can be thought that this is a proposal on write policy which is intermediate between the write-back and the write-through.

4.3.2 Self-Cleanup Strategies

The Self-Cleanup cache takes the advantage of the temporal and spatial localities of memory access [57] to make decisions on when and which line should be written-back. Based on these localities, a sort of working set of dirty lines is defined and appropriately managed by self-cleanup control. The dirty line which is forced out from this working set becomes next candidate of write-back.

The definitions of the working set and its management strategy characterize the Self-Cleanup Cache. This means that there may exist various implementations of the SCC according to those self-cleanup strategies.

In a practical implementation, however, the self-cleanup mechanism should not be a time

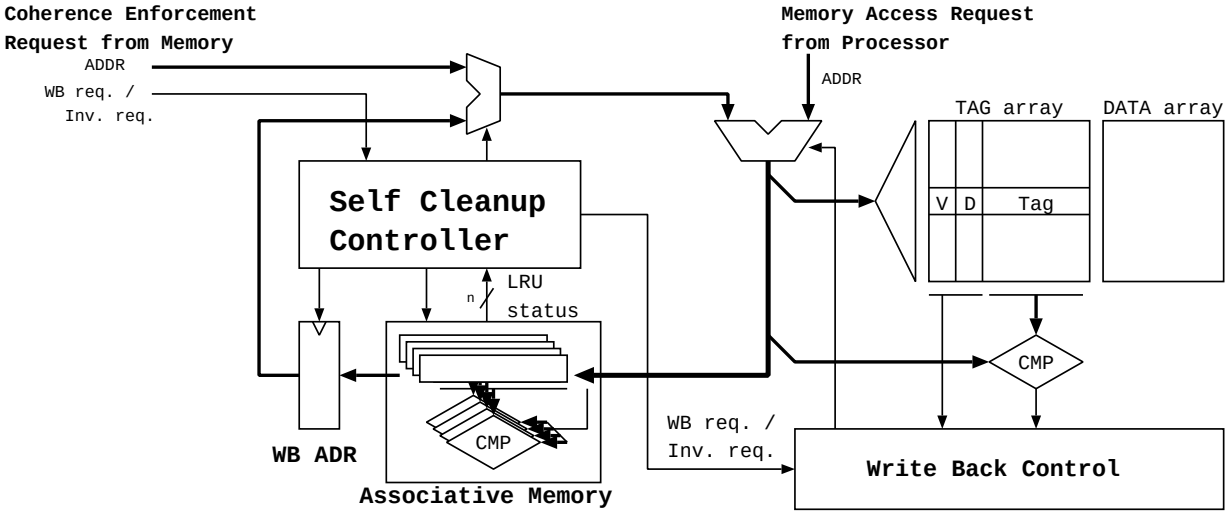


Figure 4.3: An implementation of the SCC with LRU replacement policy.

critical unit in the cache design. Thus practical self-cleanup strategy will be something like as follows.

Working set : N dirty lines which are not private to a particular processor.

Management strategy : First-In-First-Out and Least-Recently-Used replacement policies.

We show such an example in the next section.

4.3.3 An Implementation Example of the SCC with LRU Replacement Policy

We show an SCC implementation which manages the working set in LRU replacement policy. For the simplicity of explanation, we assume all accesses are to the shared data.

4.3.3.1 Hardware Organization

Figure 4.3 shows the organization of the SCC with LRU management. Basic components of SCC are a conventional write-back cache unit and an additional Self-Cleanup Control unit. The Self-Cleanup Control unit consists of the following 4 subunits.

- Associative Memory : keeps track of addresses of dirty lines in a cache.
- Write-Back ADdress Register (WB ADR) : holds the next candidate address to be written back for self-cleanup.
- Address Multiplexor : switches Write-Back Request (WBreq) from self-cleanup control and Cache Coherence enforcement Request (CCreq: Invalidation req. and write-back req.) from memory.
- Self-Cleanup controller : determines the conditions to register/remove the address to/from the associative memory according to the self-cleanup strategies. The address removed from the associative memory is first stored to the WB ADR as the next candidate of write-back.

4.3.3.2 Fundamental Operations

The fundamental SCC operations observed by processor or memory are completely the same as those of conventional write-back cache. The only difference observable from outside is the amount of write-back transactions.

Write-back transactions in the SCC are initiated in the following three cases.

1. When a line replacement is required at cache miss and a dirty line is chosen as the replacement victim, a Write-Back for the line replacement occurs.
2. When a memory requests the cache to write-back a dirty line, a Write-Back to keep coherence occurs.
3. When the self-cleanup controller requests the SCC itself to initiate a write-back, a Write-Back to keep itself clean occurs.

The third situation forces the SCC to initiate more write-back transactions than conventional write-back caches. For the sake of clarity, we refer to the write-back request initiated by self-cleanup controller as *self-cleanup request* in the following context.

In this implementation of the SCC, when a write miss[3] occurs, the self-cleanup controller requests the write-back if all entries of associative memory have been occupied.

In the following, we show the transaction sequence of the SCC when a write miss occurs, provided that the priority of write-back requests from memories or the SCC is higher than the priority of memory access requests from processor, and that no preemption is allowed.

[3]It may be because of either tag mismatch or write hit to a clean line.

- If there is available entry in the associative memory,
  register the address of the line at where the processor causes a miss to the associative memory and update the LRU history of each entries, then proceed with the ordinary miss hit transaction.
- If there is no available entry in the associative memory,
  1. Remove the address marked with Least-Recently-Used and store it to the WB ADR. And then, by asserting the WB req signal, inform the Write-Back cache units that the SCC has an address of line to be written back.
  2. Register the address of the line where the processor causes a miss to the associative memory and update the LRU history.
  3. Proceed the ordinary miss hit transaction for the missed address.
  4. Proceed the write-back transaction for the line indicated by the WB ADR. Note that the line needs not to be invalidated.
  5. If there are some requests from memory, proceed the corresponding transaction until all the requests are served.
  6. Negate the WB req signal and start accepting the request from the processor.

Although it is not observable from outside, all the time when a write hit occurs in its partner cache, LRU history of the associative memory is updated.

As we have mentioned earlier, there are several alternatives in implementing the SCC even with LRU replacement policy. Some of these alternatives are shown below.

- Ignorance of *self-cleanup requests*
  As we mentioned, the SCC works the same as the conventional write-back cache with an exception that it initiates write-back transactions for self-cleaning. And if the write-back cache unit ignores all the self-cleanup requests, the SCC is completely the same as the conventional write-back cache.

  Thus there is an option that the write-back cache unit ignores some of the self-cleanup requests, though this is not allowed if the SCC is used to realize the Compiler Aided Cache Coherence (CACC) scheme in the system with write back cache.

  For instance, this option may be used to prevent the network from being saturated. In this context, the SCC abandons the self-cleanup requests which are always not in a hurry when the network is too busy.

- Start requesting the self cleanup when more than half of the associative memory become occupied.

  In this case, the SCC may postpone the write back for self-cleanup as long as there are some available entries in the associative memory. For example, with this option, the SCC may wait initiating the self-cleanup request while the network is busy.

  This option will hopefully help to distribute the network traffic and function as a sort of load balancing mechanism. It will provide much better effect if the write-buffer is configured with the SCC to allow memory access pipelining[21].

- The dirty line flush function

  This is the indispensable function to efficiently implement the CACC control over the system with write-back cache.

  The invalidation based CACC control [16, 17, 47] requires the memories to contain up-to-date data at certain points during program execution. In order to realize this, each write-back cache has to force the write-backs corresponding to all dirty lines in each cache at these points.

  Using the dirty line flush function of the SCC, these write-backs can be done in constant time in proportion to the associative memory size, since all the dirty line addresses remain in the cache are registered in the memory, so that we don't need any exhaustive search on each cache.

4.3.4 Comparison with the Write Buffer Approach

The Self-Cleanup Cache with LRU replacement policy provides almost equivalent functions to the write-through cache with write-buffer using the write merging strategy[15, 73]. Here we make a brief comparison between the SCC and the Write-Through Cache with the write-buffer.

The advantage of the SCC as compared with the write-buffer approach is that it doesn't need the additional data buffer other than the data-array in the cache. In addition, in order for the write buffer approach to be equivalent to the SCC, the write-buffer has to adopt the write merging strategy which requires further hardware as much as small write back cache. Taking these facts into consideration, we could state that the Self-Cleanup Cache has an advantage in cost-performance.

At this point, one may claim that there must be some possibility of the contention on

data array between processor's access requests and write-back requests from self-cleanup controller. Fortunately, as we showed in the previous sections, the self-cleanup request is motivated in processor's write miss in principle, and at this time processor therefore has been in stall waiting for the completion of the write miss transaction. It may thus proceed the write-back transaction for self-cleanup in behind the write miss transaction including the ownership acquirement, so that SCC is able to be free from the contention.

One may claim also, however, that there exist some ordering models where processor doesn't stall on write miss on clean line. In this case, we may use some tricky techniques which postpone the write-back for self-cleanup until the next read/write miss that causes processor stall[75].

Furthermore, even if the associative memory of *SCM* is full when write miss occurs, the SCC does not bring the processor stall unless the SCC is used with CACC scheme, whereas in the write-buffer approach processor always stalls if the write-buffer is full when write occurs. The reason for this is that, in this context, the SCC may work as write-back cache which may buffer the up-to-date data inside its data array, thereby the SCC does not need to keep the complete address trace and also it may temporarily ignore the write-back for self-cleaning as we mentioned in the previous section.

4.4 Methodology of the Performance Evaluation

4.4.1 Goal of the Performance Evaluation

In order to demonstrate the practical effect of Self-Cleanup Cache, we did some performance evaluations by simulation. The Goal of the evaluations is to confirm the following facts.

1. Self-Invalidation based CACC schemes can work efficiently with the assistance of the Self-Cleanup cache.
2. Read stall time can be reduced as an effect of aggressive write-back initiated by Self-Cleanup mechanism.

4.4.2 Simulation Models

In our simulation works, we have examined two cache system models described below. The Self-Cleanup mechanism with LRU replacement policy is assumed throughout the

simulation works. Private data is excluded from the object of aggressive Write-Back.

SISC model is a system which adopts **S**elf **I**nvalidation based compiler assisted cache coherence scheme in combination with **S**elf-**C**leanup Cache. In the *SISC* model, whenever a processor issues a synchronization instruction, all dirty lines in the processor's cache are written back to the memory and then all cache lines except the lines which contain private data are invalidated, before the instruction is completed. We assume this invalidation is executed in one machine cycle. The Self-Invalidation scheme used in this model is equivalent to the Fast Selective Invalidation scheme[16] under the condition where all accesses to *shared data* are marked with *memory_read* and the rest (access to *private data*) are marked with *cache_read*.

DISC model is a system which adopts **DI**rectory based cache coherence scheme with the assistance of **SC**C. We assumed full-map directory scheme and Illinois protocol (4 states:Modified, Exclusive Clean, Shared, Invalid) as its coherence protocol.

Execution time and its details[4] (time spent in computation, read stall time, write stall time, and time spent waiting the completion of synchronization), and total number of messages exchanged during a program execution and its details (Read Requests and Write Requests issued by caches, and Write-Back Requests and Invalidate Requests issued by memories) are measured under all combination of the following three parameters:

1. Number of LRU entry(4,8,16,32,WOSCM[5]) ,
2. Cache size(1MB,256KB,64KB,32KB[6])
3. Network latency(10clk, 20clk/hop).

In case of the *SISC* model without *SCM* , the exhaustive search of dirty lines is made on each cache at synchronization points. However, the simulator ignores the time spent in this search, assuming that it can be overlapped with the time spent in doing write-backs.

In the simulation, we use 2 benchmarks, SOR and FFT, which have simple memory access pattern and another 2 benchmarks, WATER and MP3D from SPLASH[61], as examples of practical application. Note that only SOR and FFT are used in the simulation of *SISC* model.

[4]Details of execution time is measured on a processor in which the sum of read and write stall times is maximum of all at LRU entry of 4.

[5]Without *SCM*

[6]only for FFT and SOR

4.4.3 Overview of the Simulator

We use an execution driven simulator which consists of processor[7], cache, network and memory modules. The simulator assumes the system configuration shown in Figure 4.4 with configuration parameters shown in Table 4.1. And also we did the following assumption:

- Number of processors is 32,
- Direct-mapped cache with write-back write-policy and write-allocate on write-miss policy,
- line size of 32 bytes,
- weak ordering model,
- at most 4 outgoing write requests (including ownership requests) and 1 read requests are proceeded concurrently, and
- no write buffer exists between cache and network.

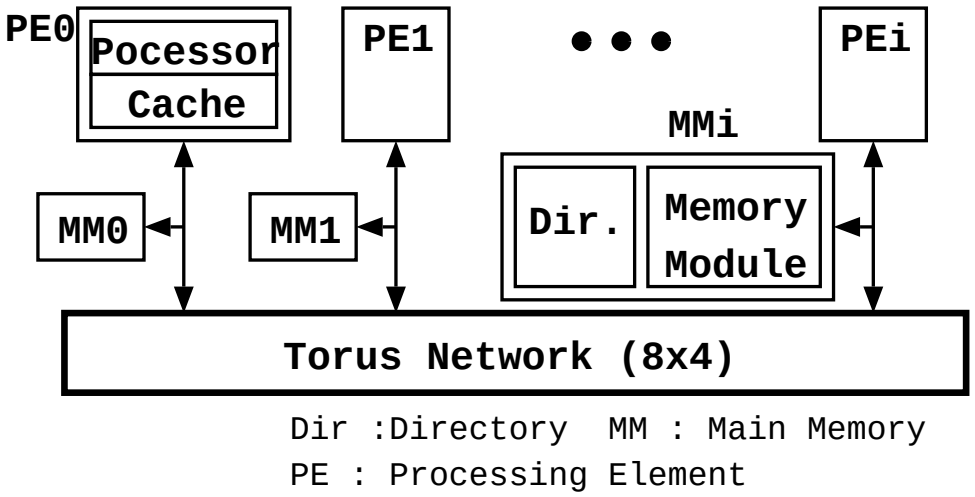


Figure 4.4: System configuration

[7]Processor module simulates a SPARC microprocessor with some extension of primitives, like PARALLEL START/END instructions, for parallel execution. This module had developed with Spa package V1.0 developed in Adelaide university, Australia for reference.

Table 4.1: System configuration parameters

Processor	Execution time of any one instruction	1[cycle]
Cache	line access time	4[cycle]
	word access time	1[cycle]
	cache tag access time	1[cycle]
Network	wo Data(1hop)	10[cycle]
	with 32Byte Data(1hop)	14[cycle]
Memory	line access time	10[cycle]
	word access time	4[cycle]
	directory access time	4[cycle]

4.5 Simulation Result

4.5.1 Summary of Results

Figures 4.5 to 4.14 show the simulation results. The x-axis of the figure indicates the number of LRU entries of Self-Cleanup Mechanism(SCM).

The summary of the results is shown below followed by detailed analysis in the following 4 sections for each benchmark program.

1. With respect to the *SISC* model, we could confirm the effect of the write-back distribution on the decrease of the waiting time at synchronization points. As a result, the speedups of the configurations with *SCM* over the configuration without *SCM* are almost 10% for SOR. For FFT, execution times are similar to each other configurations because the unnecessary write-back transactions increase the write stall time.

   On the other hand, in spite of invalidating all shared data at synchronization point, the execution time of SOR and FFT are 25% and 2% over the corresponding execution times on *DISC* model, except for the configuration without *SCM* . Although 25% increase of execution time for SOR is not acceptable, this performance degradation is due to the over invalidation rather than the unnecessary write-backs. Furthermore, if the optimization to distinguish private data from shared data is enforced on the program, *SISC* model becomes faster than the *DISC* model

2. With respect to the *DISC* model, we could confirm the reduction of read stall time as an effect of aggressive write-back from cache. Note that, for SOR and FFT, increase of

write stall time is greater than the decrease of read stall time. And thus the execution time of the system with the *SCM* is 3% longer, at worst, than the system without the *SCM* .

We also confirmed that the *SCM* never increases the cache miss from the fact that the number of read request messages is constant regardless of the configuration of *SCM* .

3. In both models, the number of LRU entries of *SCM* has some effect on the number of messages but not on the execution time. The only exception is MP3D where the number of LRU entries has some effect on the execution time.
4. To the extent of what our simulation study covers, the difference of network delay has no effect on the behavior of the system other than the proportional increase of execution time. Therefore, we consider the simulation result only for the network delay of 10 clock in the following discussion.

4.5.2 SOR

4.5.2.1 Characteristics of the problem

SOR is a Partial Differential Equation solver using Even-Odd method. A 256×256 grid is divided into 32 subgrids of 32×64 and the subgrids are assigned to processors one by one. In our simulation, we have measured the time and number of messages for the first 2 time steps excluding the sequential execution part for initialization. Since all subgrids are treated as shared data, performance degradation caused by unnecessary write-back transactions may occur. This is an example which exposes the drawback of Self-Cleanup Cache.

4.5.2.2 Analysis of the results

In the *SISC* model, the number of messages and its details are almost the same for all configurations. The differences in execution time are caused primarily by the differences in waiting time at synchronization points. All dirty lines in the cache must be written back to memory at synchronization points but there is no computation to overlap with write-backs at these points. Thus the waiting time is directly proportional to the number of dirty lines to be written back. As a result, the configurations with *SCM* become faster than the configuration without *SCM* .

In the system configured with *SCM* , write-back messages initiated by *SCM* increase significantly in *DISC* model. However, it does not affects the execution time too much because these write-backs overlap with the computation. Although read-stall time decreases as an effect of aggressive write-backs, it is slightly less than increase of the write-stall time, then the total execution time increases.

As we mentioned before, execution time on *SISC* model is roughly 1.25 times longer than on *DISC* model. This performance degradation is due to the over invalidation at synchronization points rather than the unnecessary write-backs. Hopefully, both these causes can be alleviated by some program optimization such that unshared data is appropriately declared as private data.

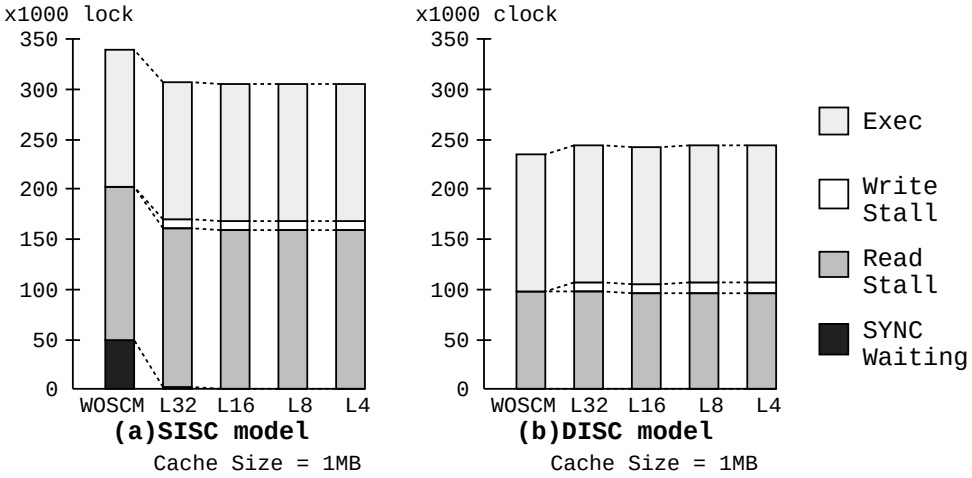
Figure 4.7 and Figure 4.8 show the improved execution time and number of messages after the optimization where just only the outmost part of the subgrid is declared as shared. By applying this optimization to original SOR program, a minimum number of messages which are essentially required are exchanged in *SISC* model, while additional messages for coherence enforcement, like ownership/invalidation request and its corresponding acknowledgement, are required in *DISC* model. As a result, the performance of the *SISC* model overcomes the *DISC* model.

The effect of the number of LRU entries in *SCM* slightly appears on the write-stall time and synchronization waiting time in *SISC* model, while in *DISC* model the effect appears on the number of Write-back request and on its corresponding write-stall time.

4.5.3 FFT

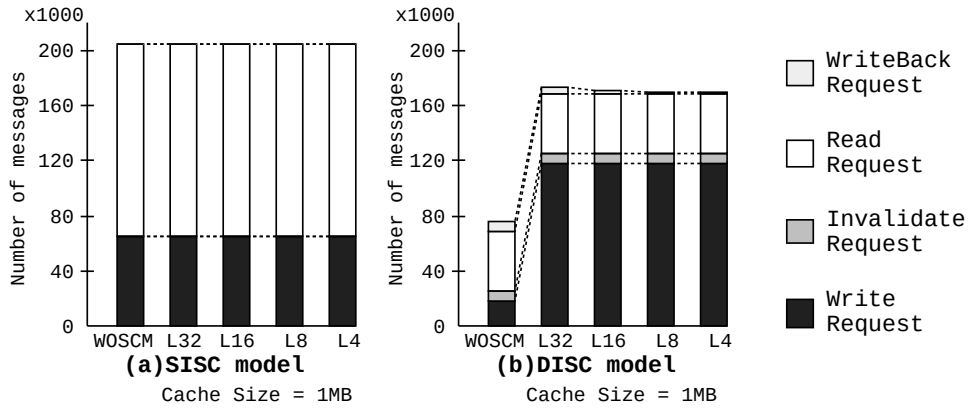
4.5.3.1 Characteristics of the problem

FFT is a parallel 2048-point FFT of complex data without data rearrangement during computation. 2048 input data are distributed among processors in block distribution manner. All the weight constants are previously calculated and delivered to each processor as private data. Parallel Computation is based on the data owner computes rule. During the execution of logN steps, each processor writes to two 16 consecutive cache lines though only half of them are written at each stage of FFT computation.



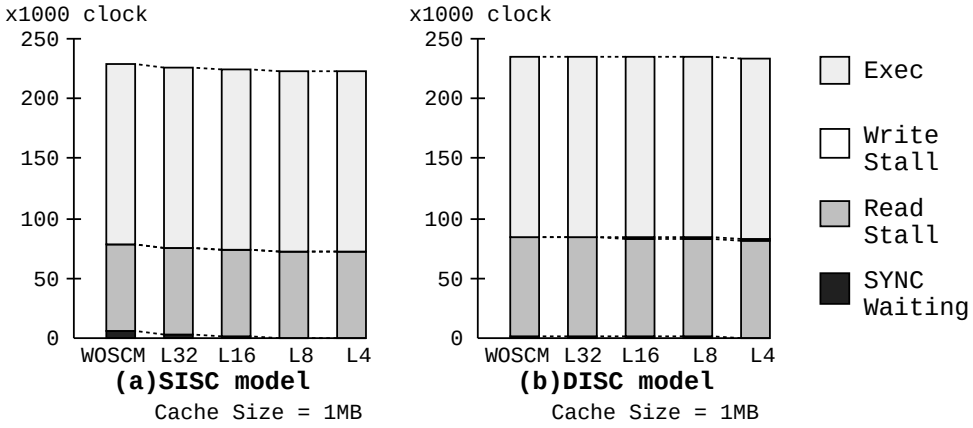
WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.5: SOR : Execution time



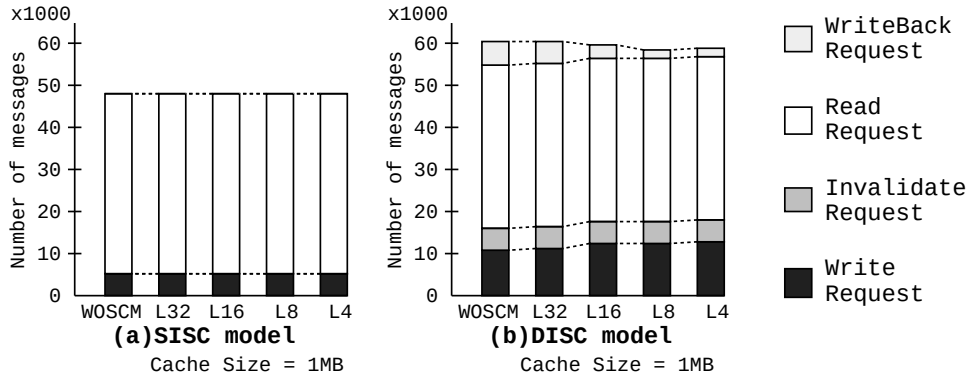
WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.6: SOR : Number of messages



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.7: SOR : Execution time (after optimization)



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.8: SOR : Number of messages (after optimization)

4.5.3.2 Analysis of the results

In case of LRU entries of 32 in the *DISC* model, the system behaves completely the same as the system without *SCM* , since all working set can be kept inside the LRU entries of *SCM* . Although read stall time slightly decreases, increase of unnecessary wait at synchronization point leads at worst 2% increase of execution time as a result.

After the computation reaches at a certain point, no interprocessor communication occurs. And then no more synchronization is required until the end of computation, which makes the working area of each processor to be 32 cache lines. This is the reason why the configurations with LRU entries of 32 and without *SCM* provide faster execution time than the other configurations with fewer LRU entries.

4.5.4 WATER

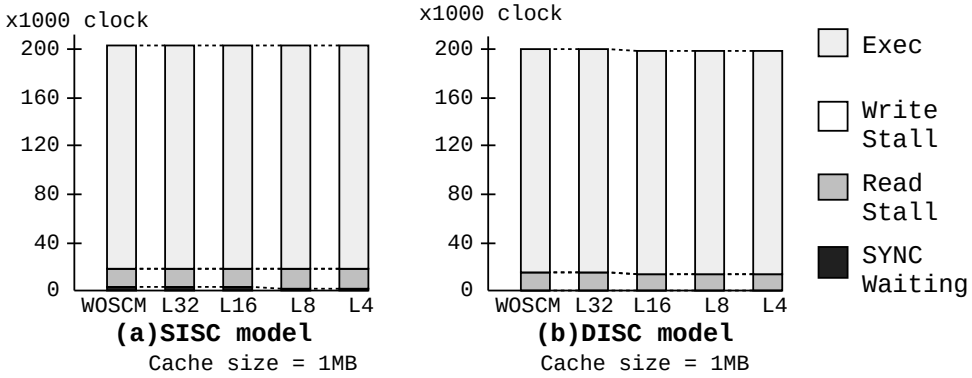
4.5.4.1 Characteristics of the problem

WATER is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. Gear's sixth-order predictor-corrector method is used to solve the Newtonian equations of motion for molecules. In our simulation, problem size of 64 molecules is used and the computation is performed over 2 time-steps. Any measurement is done for only the parallel execution part.

The data set size of 600Byte is required to store the information for each molecule. Parallelism is implemented by assigning every processor a set of molecules.

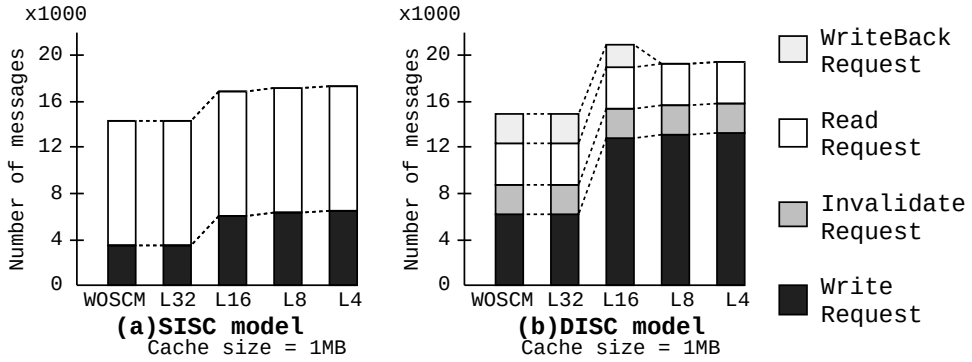
4.5.4.2 Analysis of the results

We could confirm the reduction of read stall time to be 10% to 20% by configuring with the *SCM* . However, the increase of write stall time and synchronization waiting time make the reduction of total memory access time to be 5%. Furthermore, the final speed up obtained is at most 2% since the computation time is the dominant factor of the execution time.



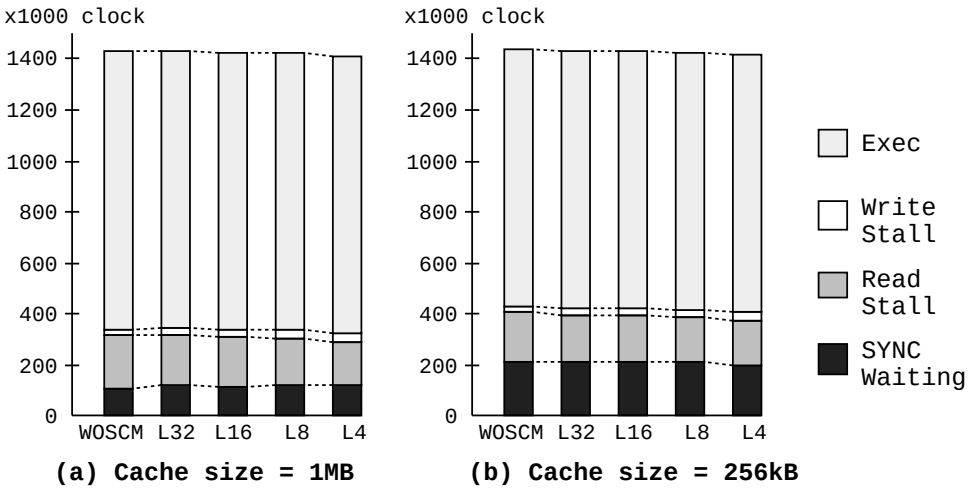
WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.9: FFT : Execution time



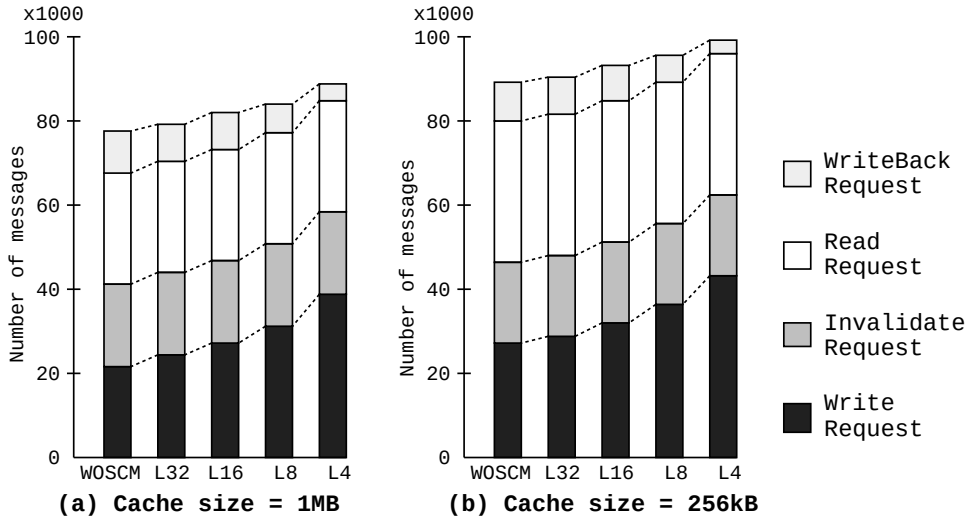
WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.10: FFT : Number of messages



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.11: WATER : Execution time



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.12: WATER : Number of messages

4.5.5 MP3D

4.5.5.1 Characteristics of the problem

MP3D is a program of rarefield hypersonic flow simulation using Monte Carlo method. In our simulation, we assume an active space of rectangular tunnel, composed of $16 \times 16 \times 8$ unit-sized cells, with openings at each end and reflecting walls on their remaining sides, and a single flat sheet placed in the space. Actions of 4096 particles (diatomic molecules) are simulated and the computation is performed over 2 time-steps. Any measurement is done for only the parallel execution part.

Each particle requires 36 bytes to store its state information and each cell requires 40 bytes to store its properties. These data are statically allocated among processors and each processor computes the actions of a certain number of particles statically assigned to itself.

4.5.5.2 Analysis of the results

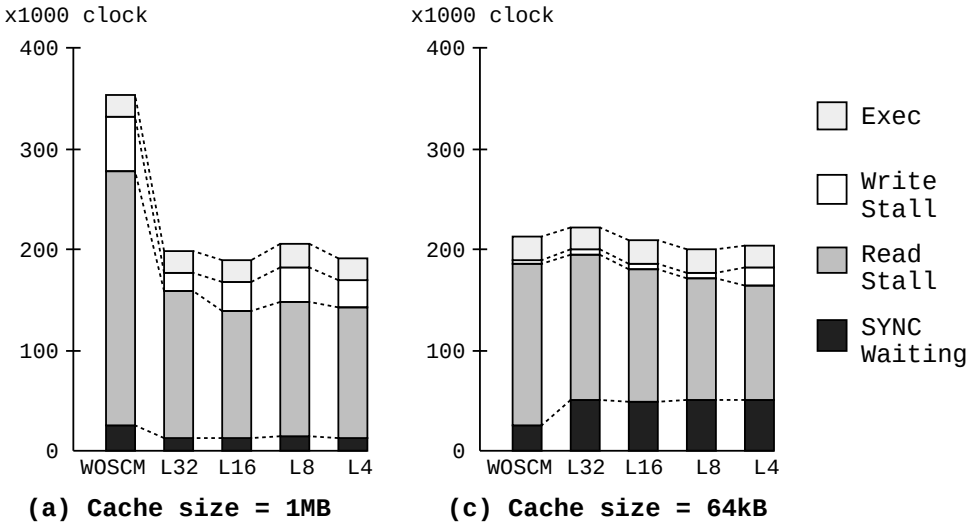
When the cache size is 1MB, in the system configured with *SCM* , it can be obtained at most 45% improvement on execution time as compared with the execution time for the configuration without *SCM* .

Since the memory access time is the dominant factor of the execution time, unlike WATER, the reduction of read stall time as an effect of aggressive write-back directly reduces the execution time.

When the cache size is 64KB, however, we could not observe any advantage of aggressive write-back. This is because the probability that the memory contains up-to-date data becomes high as a result of frequent cache line replacements which occur regardless of the existence of *SCM* . In addition, if the number of LRU entries is 32, execution time is increased by 4% as compared with the configuration without *SCM* because the write-backs initiated by the *SCM* disturb cache line replacements.

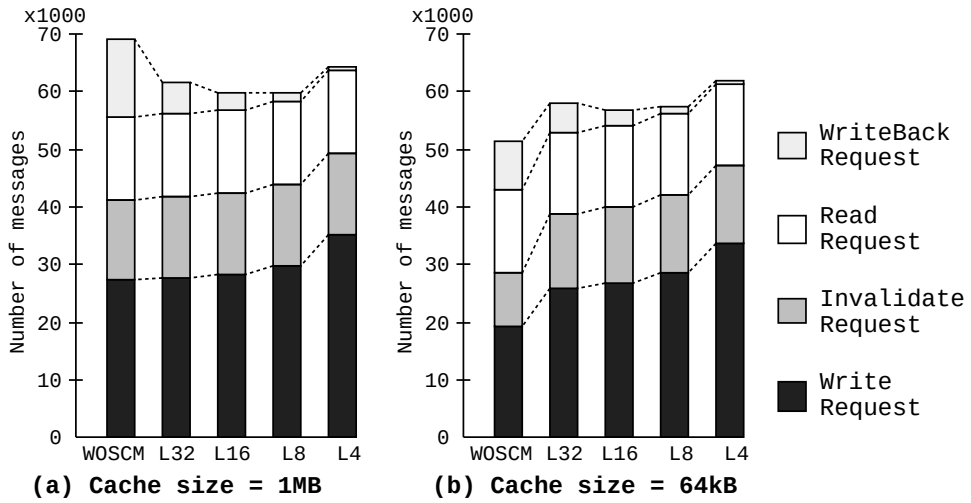
4.6 Conclusion

We have proposed a new cache system which we refer to as Self-Cleanup Cache. The SCC is composed of two units, write-back cache unit and self-cleanup mechanism. The self-cleanup controller keeps track of the address of the dirty lines in the cache and forces the write-back of the dirty lines which are not likely to be accessed by the processor any more.



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.13: MP3D : Execution time



WOSCM : indicates the WB cache without the Self-Cleanup Mechanism.
LX : indicates the SCC with X LRU entries.

Figure 4.14: MP3D : Number of messages

By means of this self-cleanup control, the SCC can keep itself as clean as possible, while maintaining the write merge effect of WB cache.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are examined by simulation upon *SISC* model and *DISC* model, respectively. The simulation results show that 1) the effect of the write-back distribution appears as the decrease of the stall time at synchronizations, and thus improves the performance of the selective invalidation scheme, and 2) the aggressive write-back decreases the read stall time, though it may increase unnecessary write-back transactions which will not affect the execution time too much. It is also confirmed that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

Chapter 5

Conclusion

5.1 Summary of This Thesis

This thesis discusses three subjects concerning cache system design for parallel processors. All these subjects have tight connection with practical machine implementations, KRPP, ASURA, and JUMP-1[35], respectively. The three design policies of *transparency, hierarchy* and *adaptability* are applied to the cache system design in this thesis.

The first subject discusses the cache system of the Kyushu University Reconfigurable Parallel Processor:KRPP. After going through the hardware design of the KRPP, we showed the implementability of distributed shared memory system which has been expected to open up new vistas toward large scale shared memory multiprocessing. The second subject discusses another cache system for cluster-based multiprocessor systems. We proposed a hierarchical cache system and its cache coherence scheme in this subject. The last subject addresses the issue of memory update policy, or write-policy, and proposes the *Self-Cleanup Cache*.

The first and second subjects are intended to study system-wide aspects of the cache systems, whereas the third subject focuses on the cache itself.

In chapter 2, the cache architecture of the KRPP is presented and the inter- and intra-cache coherence problems involved in the system are discussed. To ensure cache coherence and to measure the effectiveness of individual coherence schemes, the system provides the various schemes: (1) Cacheability marking scheme, (2) Fast selective invalidation scheme, (3) Distributed limited-directory scheme, and (4)Dual-directory cache scheme.

Both the cacheability marking scheme and the fast selective invalidation scheme are compiler-directed inter-cache coherence schemes and their effectiveness relies heavily on the ability of a parallelizing compiler. IPC(Integrated Parallelizing Compiler) system and TINPAR(TINy PARallelizer) system[48], which have been in development at Kyushu University and Kyoto University respectively, are promising compilers for this end.

On the other hand, the distributed limited-directory scheme with the dual-directory cache scheme offers hardware-based inter-cache coherence protocols. The write-through policy we have chosen simplifies the protocols as compared with the write-back policy. Under the protocols the system is weakly ordered by the new definition mentioned in [2].

The dual-directory cache scheme gives a solution to the intra-cache coherence problem, but it might suffer the collision problem if the R-tag array does not have sufficient associativity. We have analyzed the relationship between the associativity and the effect of collisions.

In chapter 3, based on the experience of the KRPP development, a large scale, cluster-based, distributed shared memory multiprocessor ASURA is proposed.

The architectural features of ASURA are placed in its layered memory system and corresponding hierarchical cache system as follows.

1. Introduction of *hierarchical shared cache* : ASURA provides, on each cluster, a global cache (GCache) which is shared among private caches(PCaches) in the cluster. The GCache provides the layer transparency by absorbing the architectural differences, such as the line size, between intra- and inter-cluster architectures, so that it becomes possible to consider these architectures independent of each other.
2. The hierarchical directory scheme with the *MLI directory* : To enforce inter-cluster cache coherence, assuring the MLI property, a new hierarchical directory scheme has been developed where the directory is composed of two separate directories corresponding to the memory hierarchy: the main directory for the GSM and the MLI directory for the GCache.
3. Software assist for the cache coherence : In order to reduce the run time overhead of the coherence enforcement, ASURA provides some means for software to assist the hardware to enforce the cache coherence. The basis of our software assist is the property labeling scheme [8, 9] which allows the software to set cache line attributes which are used for the cohererence enforcement. In addition to this scheme, ASURA provides the software with several I/O commands to manage the GCache directly.

As a result of the simulation study, a wide range of scalability is confirmed for two large scale numerical computations, a matrix multiprication and an LU decomposition. In this simulation study, we have also examined the effects of explicit GCache manipulations with the *MOVE* command, which is a combination of the software prefetching and eager write-back together with a property labeling scheme which uses the *Update on Write-Back* attribute. The simulation results show that the appropriate software assistance significantly improves the system performance.

In chapter 4, we proposed the *Self-Cleanup Cache:SCC* which aggressively enforces write-backs to update a memory, thus the cache itself becomes clean, while keeping the write merge effect of the conventional Write-Back cache.

The SCC is originally designed to efficiently implement the invalidation based Compiler Aided Cache Coherence(CACC) scheme on the multiprocessor system which configured with Write-Back caches. However, even for the systems which adopt a directory based cache coherence scheme, the SCC has a good effect in decreasing read latencies.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are examined by simulation upon two cache system models: 1)the system with the SCC and the selective invalidation scheme (*SISC* model), and 2)the other system with the SCC and the full-map directory scheme(*DISC* model), respectively.

The simulation results show that 1) the effect of the write-back distribution appears as the decrease of the stall time at synchronizations, and thus improves the performance of the selective invalidation scheme, and 2) the aggressive write-back decreases the read stall time, though it may increase unnecessary write-back transactions which will not affect the execution time too much.

It is also confirmed by the simulation that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

5.2 Future Directions

5.2.1 Development of the Cache Coherence Scheme for Massively Parallel Processors

In the development of massively parallel processors which adopt the shared memory architecture, the efficiency of the cache coherence scheme is a matter of great importance.

As it is obvious from the simulation result in chapter 4, in the directory based system, the number of messages exchanged to enforce cache coherence is not small. It is 10 to 20% of the total number of massages, though, some of these coherence enforcement messages may not be required from the first if the system employs a weak form of ordering model. Such coherence enforcement messages may incur unacceptable runtime overhead in massively parallel processors, since one of the reasons why we use the cache is the long memory access latency.

In this context, software based cache coherence schemes have an advantage that no message is exchanged at run-time to enforce coherence. However, these schemes are still developing technologies and, as mentioned below, their efficiencies depend upon the compiler.

As an consequence of these observations, we believe that the directory based cache coherence scheme with the assistance of software is the most promising approach. Along this approach, we should study various interesting issues, like how to combine the software based schemes with the directory based scheme or how to apply both schemes to the cache coherence enforcement.

The *coherence control bit* in the KRPP and the *property labeling scheme* in the ASURA encourage us in the studies of this approach[36] to the cache coherence problem.

5.2.2 Development of the Automatic Parallelizing Compiler

In this thesis, we have proposed various mechanisms with which software can collaborate with hardware in improving the system performance, however, unless providing a compiler which takes these machinisms into consideration, these mechanisms only increase the hardware cost.

On the other hand, the effectiveness of software-based cache coherence enforcement approach relies heavily on the ability of a parallelizing compiler and the coherence enforcement

scheme used in this compiler.

These facts intensively induces us to develop an automatic parallelizing compiler. We have actually developed an automatic parallelizing compiler *TINPAR* [41, 48] for the message passing parallel computer *AP1000* [37]. And we have been continuing the optimization work on TINPAR.

Although TINPAR does not take the shared memory machine into consideration in its design, we believe that this compiler can be tailored to meet the shared memory parallel processors, which employs a weak form of memory consistency model in particular.

5.2.3 Quantitative Performance Analysis

Several simulation results are presented corresponding to each subjects in this thesis. However, further simulation works are required to generalize the results of this study.

The following cache design alternatives will be our primal concerns in the future simulation studies.

1. The number of pointers in the limited directory scheme of the KRPP,
2. the cache line sizes at each level of the cache hierarchy in the ASURA, and
3. the trigger on which the self-cleanup mechanism initiates the write back.

The quantitative analysis of how the software assistance in cache coherence enforcement improves the system performance is also important because it will encourage the compiler designers to take cache aspects into consideration in thier compiler designs.

And finally we would like to investigate the various aspects of the Self-Cleanup Cache under the multiprogramming, the vector processing, and real-time processing environments.

Acknowledgment

I would like to express my sincere appreciation to Professor Shinji Tomita of Kyoto University who gave me the opportunity of this research, and has been giving me continuous guidance, interesting suggestions, intensive criticisms and encouragements during this research.

I would also like to thank my committee at Kyushu University, Professor Hiroto Yasuura, Professor Makoto Amamiya, and Professor Akifumi Makinouchi, for their many insightful suggestions that enhanced the quality of my research and the presentation of it.

I would like to thank Associate Professor Toshinori Sueyoshi of Kyushu Institute of Technology, Professor Akira Fukuda of Nara Institute of Science and Technology and Associate Professor Kazuaki Murakami of Kyushu University and all the people who have contributed to the KRPP project: H.Akaboshi, Y.Fukuzawa, K.Hamaguchi, Y.Hirotani, N.Imamura, J.Ishida, E.Iwata, K.Kai, T.Kamachi, K.Kusano, M.Kuwayama, T.Onoue, K.Sugiyama, K.Tanaka, N.Tokunaga, K.Tsunedomi, T.Ueno, K.Yasaka, and N.Yasutomi. I would also like to acknowledge the considerable contributions from N.Sakuta and T.Yokota of Asia Electronics Inc., T.Yoshimoto and S.Koshiba of Bussan System Technology Inc., M.Takamura and K.Uchida of Fujitsu Ltd., Y.Kameda and S.Shigemura of Kyocera Corp., and S.Oyanagi and N.Tanabe of Toshiba Corp. in the development of the KRPP.

I would like to thank M.Yamaguchi and K.Yamada both with KUBOTA Corp. and Professor S.Agusa of Nagoya University who gave me the opportunity to develop the ASURA. And I would like to express my special thanks to the following people who have contributed to the ASURA project: D.Fraser, T.Gleeson, M.Goshima, H.Hoshino, K.Joe, K.Marukawa, K.Nagamori, J.Naito, H.Nitta, I.Shimamoto, T.Tanaka, Y.Tsushima, H.Saito, K.Yabuuchi, M.Yanagihara. and the staffs of the office of computer business in KUBOTA Corporation.

I should mention that the parallel system simulator used in chapter 4 is an modification

of a simulator developed by Takeo Hosomi of NEC and Kiyoshi Kamishima of Fuji Xerox while they were students at Kyoto University and I would like to thank them for their assistance in its modification.

I deeply appreciate the interesting comments and useful discussions from Associate Professor Hiroshi Nakashima of Kyoto University. Thanks are also due to all the members of Professor Tomita's Laboratory for their discussions and supports throughout this research. I would like to express particular thanks to my friends, Dr. Morihiro Kuga of Kyushu Institute of Technology, Dr. Tetsuo Hironaka of Hiroshima City University and Dr. Gwun Oubong, for wonderful discussions, criticism and encouragement.

Last, but not the least, thanks are due to my parents, Kijiro Mori and Reiko Mori, for caring and encouraging me these many years.

Bibliography

[1] Adve, S.V. and Hill, M.D., "WEAK ORDERING - A NEW DEFINITION AND SOME IMPLICATIONS," *Computer Sciences Technical Report*, #902, Computer Sciences Department, University of Wisconsin-Madison, December 1989.

[2] Adve, S. V. and Hill, M. D., "Weak Ordering – A New Definition," *Proc. of 17th Int'l. Symp. on Computer Architecture*, pp.2-14, May 1990.

[3] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. of 15th Int'l. Symp. on Computer Architecture*, pp.280-289, June 1988.

[4] Amdahl, G.M., "Validity of the Single-Processor Approach to Acheiving Large Scale Comuting Capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, pp.483–485, Apr. 1967.

[5] Archibald, J. and Bear, J. -L, "An Economical Solution to the Cache Coherency Problem," *Proc. 11th Ann. Int'l. Symp. Comput. Architect.*, pp.355–362, June 1984.

[6] Archibald, J. and Bear, J. -L, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems*, Vol.4, No. 4, pp.273–298, Nov. 1986.

[7] Baer, J.-L. and Wang, W.-H., "Architectural Choices for Multilevel Cache Hierarchies, " *Proc. 1987 Int'l. Conf. on Parallel Processing*, pp.258–261, 1987.

[8] Bennet, J.K., Carter J.B., and Zwaenepoel W., "Toword Large-Scale Shared Memory Multiprocessing," *SCALABLE SHARED MEMORY MULTIPROCESSORS*, ed. by Michel Dubois and Shreekant S. Thakkar, pp.281–300, Kluwer Academic Publishers, 1991, ISBN 0-7923-9219-1.

[9] Brantley, W.C., McAuliffe, K. P., and Weiss, J., "RP3 Processor-Memory Element," *Proc. 1985 Int'l. Conf. on Parallel Processing*, pp.782–789, Aug. 1985.

[10] O'Krafka, B.W. and Newton, A.R., "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Ann. Int'l. Symp. Comput. Architect.*, pp.138–140, May 1990.

[11] Callahan, D, Kennedy, K. and Porterfield, A., "Software Prefetching," *Proc. ACM ASPLOS IV*, pp.40 – 52, 1991.

[12] Censier, M.L. and Feutrier, P., "A New Solution to Cache Coherence Problems in Multicache System," *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp.1112 – 1118, Dec. 1978.

[13] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *IEEE Computer*, Vol.23, No.6, pp.49-58, June 1990.

[14] Chen, Y.-C. and Veidenbaum, A.V., "An Improved Write Buffer Design for A Write-Through Cache," *CSRD Report*, No.1105, pp.1–17, May 1991.

[15] Chen, Y.-C. and Veidenbaum, A.V., "A Software Coherence Scheme with the Assistance of Directories," *CSRD Report*, No.1106, pp.1–22, June 1991.

[16] Cheong, H. and Veidenbaum, A. V., "A Cache Coherence Scheme with Fast-Selective-Invalidation," *Proc. 15th Int'l. Symp. Comput. Architect.*, pp. 299-307, June 1988.

[17] Cheong, H. and Veidenbaum, A. V., "A Version Control Approach to Cache Coherence, " *Proc. Int'l Conf. on Supercomputing '89*, pp. 322-330, June 1989.

[18] Cheong, H. and Veidenbaum, A. V., "Compiler-Directed Cache Management in Multiprocessors," *IEEE Computer*, Vol.23, No.6, pp. 39-47, June 1990.

[19] Cheriton, D.R., Goosen, H.A. and Boyle, P.D., "Pradigm:A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE computer*, Vol.24, No.2, pp.33–46, February 1991.

[20] Denning, P.J., "On modeling program behavior," *Commun. ACM*, vol.11, no.5, pp.323–333, May 1972.

[21] Dubois, M., Scheurich, C., and Briggs, F.A., "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l. Symp. Comput. Architect.*, pp.434–442, June 1986.

[22] Dubois, M., Scheurich, C., and Briggs, F.A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE computer*, Vol.21, No.2, pp.8–21, February 1988.

[23] Dubois, M., "Delayed Consistency," *SCALABLE SHARED MEMORY MULTIPROCESSORS*, ed. by Michel Dubois and Shreekant S. Thakkar, pp.207–218, Kluwer Academic Publishers, 1991, ISBN 0-7923-9219-1.

[24] Eggers, S.J. and Jeremiassen, T.E., "Eliminating False Sharing," *Proc. 1991 Int'l. Conf. on Parallel Processing*, pp.I-377 – I-381, Aug. 1991.

[25] Flynn, M.J., "Very High-Speed Computing Systems," *Proc. IEEE*, Vol.54, No.12, pp.1901–1909, Dec. 1966.

[26] Fraser, D. and Tanaka, T., "An overview of ASURA's Network with Simulation Results," *IPSJ Technical Report*, 93-ARC-99-18, March 1993.

[27] Fuller, S.H. and Harbison, S.P., "The C.mmp multiprocessor," *Technical Report*, Department of Computer Science, Carnegie-Mellon University, March 1978.

[28] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l. Symp. Comput. Architect.*, pp.15–26, May 1990.

[29] Goodman, J. R., "Coherency for Multiprocessor Virtual Address Caches," *Proc. ASPLOS-II*, pp.72-81, Oct. 1987.

[30] Guarna, Jr., V.A., "A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers," *Proc. 1988 Int'l. Conf. on Parallel Processing*, pp.212–220, 1988.

[31] Gupta A., Wever, W.D. and Mowry, T., "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. 1991 Int'l. Conf. on Parallel Processing*, pp.I-312 – I-321, 1990.

[32] Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 2nd printing 1986, ISBN 0-07-Y66354-8.

[33] Hill, M. D., "A Case for Direct-Mapped Caches," *IEEE Computer*, Vol.21, No.12, pp.25-40, Dec. 1988.

[34] Hill, M. D., Private Communication, 13 Sept. 1989.

[35] Hiraki, K., Amano, H., Kuga, M., Sueyoshi, T., Kudoh, T., Nakashima, H., Nakajo, H., Matsuda, H., Matsumoto, T. and Mori, S., "Overview of a Massively Parallel Computer Prototype: JUMP-1," *IPSJ Technical Report*, 93-ARC-102-10, Oct 1993.

[36] Hosomi, T., Mori, S., Nakashima, H. and Tomita, S., "A Hardware Cache Coherence Scheme with the assistance of software (in Japanease)," *IPSJ Technical Report*, 93-ARC-99-16, Mar. 1993.

[37] Ishihata, H., Horie, T., Inano, S., Shimizu, T., Kato, S. and Ikesaka, M., "Third Generation Message Passing Computer AP1000," *Proc. of the Int'l Symp. on Supercomputing*, pp.46–55, Nov. 1991.

[38] Iwata, E., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Cache System of the Kyushu University Reconfigurable Parallel Processor (in Japanese)," *IPSJ Technical Report*, 89-ARC-79-3, Nov. 1989.

[39] Iwata, E., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Integrated Parallelizing Compiler System – Compiler-Assisted Cache Coherence Scheme – (in Japanese)," *IPSJ Technical Report*, 90-ARC-83-21, July 1990.

[40] Konicek, J., Tilton, T., Veidenbaum, A., Zhu, C.Q., Davidson, E.S., Downing, R., Haney, M., Sharma, M., Yew, P.C., Farmwald, P.M., Kuch, D., Lavery D., Lindsey, R., Pointer, D., Andrews, J., Beck, T., Murphy, T., Turner, S. and Warter, N., "The Organization of the Cedar System," *Proc. 1991 Int'l. Conf. on Parallel Processing*, pp.149–156, Aug. 1991.

[41] Kubota, A., Miyoshi, I., Ohno, K., Mori, S., Nakashima, H., and Tomita, S., "A Parallelizing Compiler Technique for Loops with Irregular Accesss — New Algorithms to Improve the Performance of Inspector/Executer — (in Japanese)," *Trans. of IPS Japan*, Vol.35, No. 4, pp. 532-541, Apr. 1994.

[42] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J., "The Directory-based Cache Coherece Protocol for the DASH Multiprocessor," *Proc. 17th Ann. Int'l. Symp. Comput. Architect.*, pp.148–159, May 1990.

[43] Zhiyuan Li, "Software Assistance for Directory-Based Caches," *Proc. 8th Int'l. Parallel Processing Symp.*, pp.151–157, April 1994.

[44] Maa, Y.-C., Pradhan, D.K. and Thiebaut, D., "Two Economical Directory Schemes for Large-Scale Cache Coherent Multiprocessors," *ACM SIGARCH Comput. Architect. News*, Vol.19, No.5, pp.10–18, Sept. 1991.

[45] Maples, C., "A High-Performance Memory-Based Interconnection System for Multicomputer Environments," *Proc. of Supercomputing'90*, pp.295-304, 1990.

[46] Matsumoto, T. , "Finge-Grain Support Mechanisms(in Japanese)," *IPSJ Technical Report*, 89-ARC-77-12, pp.91–98, July 1989.

[47] Min.,S.L. and Baer.,J.-L., "A Timestamp-Based Cache Coherence Scheme," *Proc. 1989 Int'l. Conf. on Parallel Processing*, pp.I-23 – I-32, June 1989.

[48] Miyoshi, I., Mori, S., Nakashima, H., and Tomita, S., "TINPAR: A Parallelizing Compiler for Message-passing Multiprocessors," *Proc. of the Third Parallel Computing Workshop for Fujitsu Parallel Computing Research Facilities*, pp.P1-H-1 – P1-H-9, Nov. 1994.

[49] Mori, S., Kamachi, T., Hamaguchi, K., Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "Inter-PE Communication Subsystem of the Kyushu University Reconfigurable Parallel Processor (in Japanese)," *Trans. IPS Japan*, Vol.30, No.12, Dec. 1989.

[50] Mori, S., Murakami, K., Iwata, E., Fukuda, A., and Tomita, S., "The Kyushu University Reconfigurable Parallel Processor – Cache Architecture and Cache Coherence Schemes –," *Proc. Int'l Symp. on Shared Memory Multiprocessing*, pp.218–229, April 1991.

[51] Mori, S., Saito, H., Goshima, M., Tomita, S., Tanaka, T., Fraser, D., Joe, K. and Nitta, H., "Overview of the ASURA: A Distributed Shared Memory Multiprocessor (in Japanese)," *IPSJ Technical Report*, 92-ARC-94-6, June 1992.

[52] Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "An Overview of the Kyushu University Reconfigurable Parallel Processor," *ACM SIGARCH Computer Architecture News*, Vol.16, No.4, pp.130-137, Sept. 1988.

[53] Murakami, K., Mori, S., Fukuda, A., Sueyoshi, T., and Tomita, S., "The Kyushu University Reconfigurable Parallel Processor – Design of Memory and Intercommunication Architectures – ," *Proc. 1989 ACM SIGARCH Int'l. Conf. Supercomput.*, pp.351-360, June 1989.

[54] Murakami, K., Mori, S., Fukuda, A., Sueyoshi, T., and Tomita, S., "The Kyushu University Reconfigurable Parallel Processor – Design Philosophy and Architecture – ," *Proc. IFIP 11th World Computer Congress*, pp.995-1000, Aug. 1989.

[55] Murakami, K., *Design and Evaluation of Advanced Processor Architectures*, Ph.D. Thesis, Department of Computer Science, Kyoto Univesity, Nov. 1993.

[56] Naito, J., Joe, K., Matsuno, H. and Nitta, H., "Performance Evaluation of the ASURA Cluster (in Japanese)," *IPSJ Technical Report*, 92-ARC-97-10, Dec. 1992.

[57] David A. Patterson and John L. Hennessy, *Computer architecture : A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990, ISBN 1-55860-069-8.

[58] Rettberg, R. and Thomas, R., "Contention is no Obstacle to Shared-Memory Multiprocessinf," *Commun. ACM*, vol.29, no.12, pp.1202-1212, Dec. 1986.

[59] Saito, H., Mori, S., Tomita, S., Tanaka, T., Fraser, D. and Joe, K., "The Event Correspondent Cache Coherency Scheme and Its Application to Barrier Synchronization (in Japanese)," *IPSJ Technical Report*, 92-ARC-95-2, Aug. 1992.

[60] Scheurich, C.E., "ACCESS ORDERING AND COHERENCE IN SHARED MEMORY MULTIPROCESSORS," Ph.D. Thesis, Department of Comuter Engineering, *Technical Report*, No. CENG 89-19, University of Suthern California, May 1989.

[61] Singh, J.P., Weber, W.D., and Gupta, A.,: "SPLASH:Stanford Parallel Applications for Shared-Memory", *Tech. Rep. CSL-TR-91-469, Stanford Univ.*,Apr.,1991

[62] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, Vol.14, No.3, pp.473-530, Sept. 1982.

[63] Stenström, P., "Reducing Contention in Shared-Memory Multiprocessors," *IEEE Computer*, Vol.21, No.11, pp.26-37, Nov. 1988.

[64] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessor," *IEEE Computer*, Vol.23, No.6, pp. 12-24, June 1990.

[65] Swan, R. J., Fuller, S. H., and Siewiorek, D. P., "Cm* – A Modular, multi-microprocessor," Proc. National Computer Conf., Vol.46, pp.637-644, 1977.

[66] Terasawa, T., Amano, H. and Kudoh, T., "Memory Systems for Multiprocessors (in Japanese)," *Journal of IPS Japan*, Vol.34, No.1, pp.96-105, Jan. 1993.

[67] Tokunaga, N., "*Memory Consistency Models: – Frameworking and Performance Evaluation* – (in Japanease)," Master Thesis, Dep. of Information System, Kyushu University Feb. 1993.

[68] Tomita, S., *Parallel Computer Architectures*(in Japanese), Shokodo, 1990, ISBN 4-7856-3066-3.

[69] Torrellas, J., Lam, M.S. and Hennessy, J., "Shared Date Placement Optimization to Reduce Muliprocessor Cache Miss Rates," *Proc. 1991 Int'l. Conf. on Parallel Processing*, pp.II-266 – II-270, 1990.

[70] Veidenbaum, A. V., "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. 1986 Int'l. Conf. on Parallel Processing*, pp.1029 – 1039, 1986.

[71] Wilson Jr., A.W., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. 14th Ann. Int'l. Symp. Comput. Architect.*, pp.244–252, 1987.

[72] "Paragon XP/S Product Overview," *Intel Corporation*, 1991.

[73] "Alpha Architecture Handbook," *DIGITAL EQUIPMENT Co.*, 1992.

[74] "SPARC MB86901 (S-25) High Performance 32-bit RISC Processor Product Description, " *Fujitsu Microelectronics, Inc.*, Sept. 1988.

[75] "SuperSPARC User's Guide," *TEXAS INSTRUMENTS*, Oct. 1992.

List of Publications by the Author

Major publications

[1] Murakami, K., Mori, S., Fukuda, A., Sueyoshi, T., and Tomita, S., "System Philosophy of a Reconfigurable Parallel Processor (in Japanese)," *Kyushu University Engineering Sciences Reports*, vol.10, no.3, pp.337-343, Dec. 1988.

[2] Mori, S., Hamaguchi, K., Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "Inter-PE Communication Mechanism in the Reconfigurable Parallel Processor (in Japanese)," *Proc. of 1989 IPSJ Joint Symposium on Parallel Processing* pp.123-130, Feb. 1989.

[3] Murakami, K., Mori, S., Hamaguchi, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "Architecture of the the Reconfigurable Parallel Processor (in Japanese)," *Tohoku University Applied Information Science Reports*, vol.14, no.2, pp.141-151, Mar. 1989

[4] Murakami, K., Mori, S., Fukuda, A., Sueyoshi, T., and Tomita, S., "The Kyushu University Reconfigurable Parallel Processor - Design of Memory and Intercommunication Architectures - ," *Proc. 1989 ACM SIGARCH Int'l. Conf. Supercomput.*, pp.351-360, June 1989.

[5] Murakami, K., Mori, S., Fukuda, A., Sueyoshi, T., and Tomita, S., "The Kyushu University Reconfigurable Parallel Processor - Design Philosophy and Architecture - ," *Proc. IFIP 11th World Computer Congress*, pp.995-1000, Aug. 1989.

[6] Mori, S., Kamachi, T., Hamaguchi, K., Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "Inter-PE Communication Subsystem of the Kyushu University Reconfigurable Parallel Processor(in Japanese)," *Trans. of IPS Japan*, vol.30, no.12, pp.1593-1602, Dec. 1989.

[7] S.Mori, K. Murakami, E. Iwata, A. Fukuda, and S. Tomita, "The Kyushu University Reconfigurable Parallel Processor — Cache Architecture and Cache Coherence Schemes —(in Japanese)," *Proc. of Int'l Symp. on Shared Memory Multiprocessing*, pp.218–229, Apr. 1991.

[8] Tokunaga, N., Ueno, T., Iwata, E., Kai, K., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "KRPP: Kyushu University Reconfigurable Parallel Processor — Design of Processing Element —(in Japanese)," *Proc. of 1991 IPSJ Joint Symposium on Parallel Processing*, pp.229–236, May 1991.

[9] Iwata, E., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Cache Architecture of the Kyushu University Reconfigurable Parallel Processor — Cache Design and Intra-cache Coherence Protocol —(in Japanese)," *Trans. of IPS Japan*, Vol.32 No.6, pp.777-789, June 1991.

[10] Mori, S., Saito, H., Goshima, M., Yanagihara, M., Tanaka, T., Fraser, D., Joe, K., Nitta, H., and Tomita, S., "A Distributed Shared Memory Multiprocessor : ASURA — Memory and Cache Architecures —," *Proc. of Supercomputing '93*, pp. 740-749, Nov. 1993.

[11] Keshk, H., Mori, S., Hakashima, H., and Tomita, S., "A new technique to improve parallel automated single layer wire routing," *Proc. of the Performance Evaluation of Parallel Systems*, pp. 134-141, Nov. 1993.

[12] Araki, S., Goshima, M., Mori, S., Nakashima, H., and Tomita, S., "Application of Parallelized DP an A* Algorithm to Multiple Sequence Alignment," *Proc. of Genome Informatics Workshop IV*, pp. 94-102, Dec. 1993.

[13] Kubota, A., Miyoshi, I., Ohno, K., Mori, S., Nakashima, H., and Tomita, S., "A Parallelizing Compiler Technique for Loops with Irregular Accesss — New Algorithms to Improve the Performance of Inspector/Executer — (in Japanese)," *Trans. of IPS Japan*, Vol.35, No. 4, pp. 532-541, Apr. 1994.

[14] Tsushima, Y., Akashi, H., Jin, X., Susukita, M., Kuroda, T., Mori, S., Nakashima, H., and Tomita, S., "A Parallel Comuter Architecture for Volume Rendering (in Japanese), " *Proc. of 1994 IPSJ Joint Symposium on Parallel Processing*, pp. 89-96, May 1994.

[15] Hosomi, T., Mori, S., Nakashima, H., and Tomita, S., "Evaluation of cache coherence schemes based on update policy (in Japanese)," *Proc. of 1994 IPSJ Joint Symposium on Parallel Processing*, pp. 287-294, May 1994.

[16] Toerresen, J., Mori, S., Nakashima, H., and Tomita, S., "Parallel back propagation training algorithm for MIMD computer with 2D-torus network," *Proc. of Int'l Conf. on Neural Information Processing*, pp.140-145, Oct 1994.

Oral Presentations

[1] Kamachi, T,. Kai, K., Mori, S., Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "Network Control in the Kyushu University Reconfigurable Parallel Processor(in Japanese)," *IEICE Technical Report*, CPSY89-16, Aug. 1989.

[2] Iwata, E., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Cache System of the Kyushu University Reconfigurable Parallel Processor (in Japanese)," *IPSJ Technical Report*, 89-ARC-79-3, Nov. 1989.

[3] Kai, K., Mori, S., Murakami, K., Fukuda, A., Sueyoshi, T., and Tomita, S., "The Local/Remote Architecture of the Kyushu University Reconfigurable Parallel Processor(in Japanese)," *IPSJ Technical Report*, 90-ARC-80-11, Jan. 1990.

[4] Iwata, E., Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Integrated Parallelizing Compiler System – Compiler-Assisted Cache Coherence Scheme – (in Japanese)," *IPSJ Technical Report*, 90-ARC-83-21, July 1990.

[5] Mori, S., Murakami, K., Fukuda, A., and Tomita, S., "Multiprexed Crossbar Network of the Kyushu University Reconfigurable Parallel Processor(in Japanese)," *IEICE Technical Report*, CPSY90-14, July 1990.

[6] Mori, S., Saito, H., Goshima, M., Tomita, S., Tanaka, T., Fraser, D., Joe, K., and Nitta, H., "Overview of the ASURA: A Distributed Shared Memory Multiprocessor (in Japanese)," *IPSJ Technical Report*, 92-ARC-94-6, June 1992.

[7] Saito, H., Mori, S., Nakashima, H., Tomita, S., Tanaka, T., Fraser, D., Joe, K., "The Event Correspondent Cache Coherency Scheme and Its Application to Barrier Synchronization (in Japanese)," *IPSJ Technical Report*, 92-ARC-95-2, Aug. 1992.

[8] Goshima, M., Mori, S., Tomita, S., "The On-Memory FIFO Mechanism for Inter-Processor Communication (in Japanese)," *IPSJ Technical Report*, 92-OS-56-15, Aug. 1992.

[9] Horikawa, K., Mori, S., Tomita, S., "Optimial Load Balancing for Parallel Load-flow Analysis (in Japanese)," *IPSJ Technical Report*, 92-NA-42-4, Aug. 1992.

[10] Hosomi, T., Mori, S., Tomita, S., "Loop Optimization for Superscalra Processor(in Japanese)," *IEICE Technical Report*, CPSY92-32, Aug. 1992.

[11] Akashi, H., Kitasuka, T., Susukita, M., Mori, S., Nakashima, H., and Tomita, S., "A Parallel Computer Architecture for Volume Rendering (in Japanese)," *IPSJ Technical Report*, 92-ARC-97-15, Dec. 1992.

[12] Saito, H., Mori, S., Tomita, S., Tanaka, T., Fraser, D., Joe, K., "An Implementation of the Event Correspondent Cache Coherency Scheme and its Performance Analysis(in Japanese)," *IEICE Technical Report*, VLD92-86, Jan. 1993.

[13] Hosomi, T., Mori, S., Nakashima, H., and Tomita, S., "A Hardware Cache Coherence Scheme with the assistance of software (in Japanese)," *IEICE Technical Report*, CPSY92-89, Mar. 1993.

[14] Kubota, A., Ohno, K., Misyoshi, I., Mori, S., Nakashima, H., and Tomita, S., "An Automatic Parallelizing Compiler Technique for Loops with Irregular Accesss (in Japanese)," *IEICE Technical Report*, COMP92-95, Mar. 1993.

[15] Kubota, A., Miyoshi, I., Ohno, K., Mori, S., Nakashima, H., and Tomita, S., "Automatic Parallelizing Compiler for Distributed Memory Parallel Computers — New Algorithms to Improve the Performance of Inspector/Executer — (in Japanese)," *Proc. of 1993 IPSJ Joint Symposium on Parallel Processing*, pp. 47-54, May 1993.

[16] Mori, S., Omori, Y., Nakashima, H., and Tomita, S., "Proposal on Self-Cleanup Write-Back Cache (in Japanese)," *IPSJ Technical Report*, 93-ARC-100-2, June 1993.

[17] Ohno, K., Mori, S., Nakashima, H., and Tomita, S., "Extension of Distributed Tree Search for Dynamic Load Balancing and its Implementation as a Library on AP1000 (in Japanese)," *IPSJ Technical Report*, 93-PL-13-8, Aug 1993.

[18] Keshk, H., Mori, S., Nakashima, H., and Tomita, S., "A new technique to improve parallel automated single layer wire routing," *IPSJ Technical Report*, 93-PL-13-10, Aug 1993.

[19] Goshima, M., Okada, C., Hosomi, T., Mori, S., Nakashima, H., and Tomita, S., "The Intelligent Cache System for Fine-Grain Inter-Processor Communication (in Japanese), " *IPSJ Technical Report*, 93-ARC-101-16, Aug 1993.

[20] Horikawa, K,, Mori, S., Nakashima, H., and Tomita, S., "Simulated Annealing Heuristics for Static Tack Assignment (in Japanese)," *IPSJ Technical Report*, 93-AL-34-6, Aug 1993.

[21] Hiraki, K., Amano, H., Kuga, M., Sueyoshi, T., Kudoh, T., Nakashima, H., Nakajo, H., Matsuda, H. Matsumoto, T., and Mori, S., "Overview of a Massively Parallel Computer Prototype: JUMP-1," *IPSJ Technical Report*, 93-ARC-102-10, Oct 1993.

[22] Horikawa, K., Mori, S., Nakashima, H., and Tomita, S., "Simulated Anneling Heuristics for Static Task Assignment," *Proc. of the Second Parallel Computing Workshop for Fujitsu Parallel Computing Research Facilities*, pp.P1-B-1 – P1-B-11, Nov. 1993.

[23] Keshk, H., Mori, S., Nakashima, H., and Tomita, S., "A new technique to improve parallel automated single layer wire routing," *Proc. of the Second Parallel Computing Workshop for Fujitsu Parallel Computing Research Facilities*, pp.P1-R-1 – P1-R-11, Nov. 1993.

[24] Toerresen, J., Mori, S., Nakashima, H., and Tomita, S., "Parallel back propagation training algorithm for MIMD computer with 2D-torus network," *IEICE Technical Report*, CPSY94-30, July 1994.

[25] Keshk, H., Mori, S., Nakashima, H., and Tomita, S., "A Parallel Automated Wire Router : Amon," *IEICE Technical Report*, CPSY94-43, July 1994.

[26] Tsushima, Y., Jin, X., Mori, S., Nakashima, H., and Tomita, S., "A Parallel Comuter Architecture for Volume Rendering: ReVolver/C40 (in Japanese)," *IPSJ Technical Report*, 94-ARC-107-18, July 1994.

[27] Goshima, M., Ikeda, Y., Mori, S., Nakashima, H., and Tomita, S., "Virtual Queue: A Message Communication Mechanism for Massively Parallel Comuters (in Japanese)," *IPSJ Technical Report*, 94-ARC-107-19, July 1994.

[28] Miyoshi, I., Mori, S., Nakashima, H., and Tomita, S., "A Parallelizing Compiler for Message-Passing Multiprocessor (in Japanese)," *IPSJ Technical Report*, 96-PL-18-5, July 1994.

[29] Miyoshi, I., Mori, S., Nakashima, H., and Tomita, S., "TINPAR: A Parallelizing Compiler for Message-passing Multiprocessors," *Proc. of the Third Parallel Computing Workshop for Fujitsu Parallel Computing Research Facilities*, pp.P1-H-1 – P1-H-9, Nov. 1994.

[30] Mori, S., Nakashima, H., and Tomita, S., "Evaluation of Self-Cleanup Cache (in Japanese)," *IPSJ Technical Report*, 94-ARC-109-7, Dec. 1994.