ABSTRACT

Title of thesis: Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering

Alokika Dash, Master of Science, 2006

Thesis directed by: Professor Peter Petrov
Department of Electrical Engineering

We present a novel methodology for power reduction in embedded multiprocessor systems. Maintaining local caches coherent in bus-based multiprocessor systems results in significantly elevated power consumption, as the bus snooping protocols result in local cache lookups for each memory reference placed on the common bus. Such a conservative approach is warranted in general-purpose systems, where no prior knowledge regarding the communication structure between threads or processes is available. In such a general-purpose context the assumption is that each memory request is potentially a reference to a shared memory region, which may result in cache inconsistency, if no correcting activities are undertaken. The approach we propose exploits the fact that in embedded systems, important knowledge is available to the system designers regarding communication activities between tasks allocated to the different processor nodes. We demonstrate how the snoop-related cache probing activity can be drastically reduced by identifying in a deterministic way all the shared memory regions and the communication patterns between the processor nodes. Cache snoop activity is enabled only for the fraction of the bus transactions, which refer to locations belonging to known shared memory region for each processor node; for the remaining larger part of memory references known to be of no relation to the given processor node, snoop probings in the local cache are completely disabled, thus saving a large amount of power. The required hardware support is not only cost-efficient, but is also software programmable, which allows the system software to dynamically customize the cache coherence controller to the needs of different tasks or even different parts of the same program. The experiments which we have performed on a number of important applications demonstrate the effectiveness of the proposed approach.

Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering

by

Alokika Dash

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Masters of Science
2006

DEDICATION

I dedicate this thesis to my parents. They have been my role models. Any accomplishments of mine are due in no small part to their support.

Thank you Mommy and Daddy.

ACKNOWLEDGMENTS

I am grateful to my advisor, Dr. Peter Petrov, for his direction and encouragement for the past one year here at the University of Maryland. He has not only been a great academic advisor to me, but has also given his invaluable guidance in many little quirks. I am truly indebted to him for keeping me on point and pushing me to do my best. I am sure the knowledge I have gained as being a student under his supervision will help me throughout my future endeavors in academics and professional career. This work would not have been possible without him.

Further, I would like to express my gratitude towards Dr. Franklin and Dr. Jaja for agreeing to be on my committee.

I would like to thank my Pati family and relatives who have always advised me. I would also like to thank my room-mate Marjan and all my friends for their help and support in my two years of graduate studies in the University. I have a special thanks to offer to my fiance Siddharth, for giving suggestions on my work and proof reading my thesis and critiquing it.

Finally I am indebted to my beloved parents,Puspa and Alok Dash, my sister and brother for believing in me and supporting me especially during the difficult times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Embedded Systems

The use and application of embedded systems in our day to day lives has proliferated in recent years. In fact, embedded processors constitute over 95% of the global processor market share. The decreasing cost of processor based systems coupled with Moore's law has led to the ubiquity of embedded systems and devices. As the name suggests, embedded systems, unlike their desktop counterparts, manifest themselves in the environment. They appear in systems that we use in our everyday lives like microwaves, cell phones, ATMs, PDAs, cash registers, elevators, automated gas stations to name a few. Embedded systems, however, have constraints compared to their desktop counterparts. These constraints are mainly due to the fact that embedded systems are designed to serve a specific purpose in mind. Some of the most important factors for such systems have long been recognized to be the design cost, time-to-market, flexible implementation, performance, energy, real-time guarantees, size (form factor) and portability. The design constraints of embedded systems has led to a large body of research dedicated to design issues in embedded systems.

One such design constraint, *power* and *energy consumption* of embedded systems continues to be a topic of research in system design. This has been widely due to two reasons. First, a large class of embedded systems are battery driven, hence longer battery life is an obvious design metric. Secondly, with shrinking transistor densities, power dissipated is directly proportional to increasing processor core temperatures. Thus, designs with a fixed power budget are of prime importance.

Embedded applications impose high performance demands, while at the same time exhibit stringent power constraints; such applications, to name a few, include multimedia support such as audio/image/video capture and processing, and data-intensive wireless devices, such as sensor nodes for environmental, industrial, or security data acquisition and analysis. Memory in embedded systems is known to be extremely power consuming. Moreover multimedia applications are memory intensive applications. Since design cost and time-to-market are major requirements for product success, implementation platforms based on processor cores are typically utilized instead of custom hardware. In order to meet the high performance requirements, implementations containing multiple processor cores have started to emerge.

1.2 Multiprocessors in Embedded Systems

To be competitive in the market, new computer designs must exhibit rapid increases in functionality, reliability, and bandwidth and rapid declines in cost and power consumption. Although general-purpose processors can handle many tasks, they usually lack the bandwidth needed to perform complex data-processing tasks such as network packet processing, video processing,and encryption. Therefore processors cores such as Multi Processor System on Chips (MP-SoCs) have emerged. These are system on chip platforms with multiple processor cores on a single die [28]. One of the major advantages of MP-SoCs is that it helps reduces the area and communication overhead. These platforms typically contain multiple heterogeneous, flexible processing elements, a memory hierarchy and I/O components. All of these components are linked to each other by a flexible on-chip interconnect structure. These architectures meet the performance needs of contemporary and future multimedia applications, while limiting the power consumption.

MP-SoCs are targeted towards particular application(s) rather than being a general-purpose chip. One of the main advantages exhibited by applications running on such SoCs is task level parallelism. Task-level parallelism is very important in embedded computing. This type of parallelism is relatively easy to leverage since the system specification naturally decomposes the problem into tasks having multiple phases. However, such general-purpose computing architectures come with the price of excessive power consumption, a characteristic of extreme importance for many wearable and battery-powered devices.

In order to map an application to an MP-SoC platform, the application designers need a shared memory model view of the MP-SoC architecture. Memory system programming model in MP-SoC's could use either shared or partitioned memory. In a shared-memory multiprocessor a pool of processors and a pool of memory are connected by an interconnection network. Each component is regularly structured, and the programmer is presented with a conventional programming model. Most multiprocessors today use shared-memory architectures because a shared-memory model makes life simpler for the programmer and is also cheaper in terms of hardware implementation. They also exhibit low communication latency and decentralized topology easily supporting all the processing units.

However, the shared memory architectures have certain disadvantages and therefore networks-on-chips (NoC) have emerged over the past few years as an architectural approach to the design of single-chip multiprocessors[29]. A network-on-chip uses packet networks to interconnect the processors in the SoC. Such direct or point-to-point network is an architecture that overcomes the scalability problems of shared medium networks.In this architecture, each node is directly connected with a subset of other nodes in the network, called neighboring nodes. Nodes are on-chip computational units, but they contain a network interface block, often called a router, which handles communication-related tasks. Each router is directly connected with the routers of the neighboring nodes.

Different from shared-medium architectures, as the number of nodes in the system increases, the total communication bandwidth on direct networks also increases. Direct interconnect networks are therefore very popular for building large-scale systems where as shared memory architecture are used for most small-scale systems.

1.3 Shared Memory Multiprocessors

Power optimization by memory optimization can be done in several ways: the reduction in memory size and improved data-movement strategies over the memory hierarchy [24],[26],[25]. In shared-memory multiprocessors architectures, all proces-

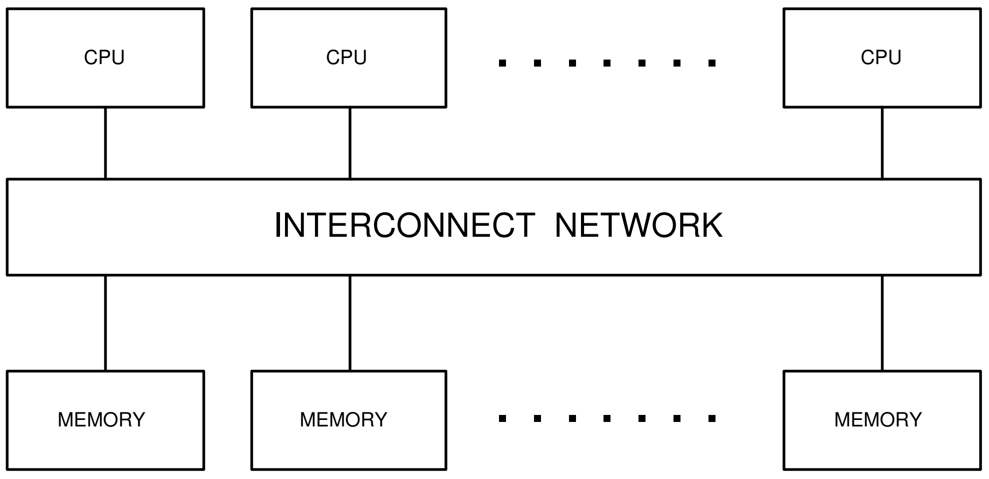


Figure 1.1: Shared-memory multiprocessor system

sors share the bus for their memory accesses as shown in Figure 1.1. The available bandwidth can be easily exhausted and thus can quickly lead to significant performance degradations. To alleviate this problem, caches are used to replicate the data and bring it closer to the requesting processors, thus saving bus bandwidth and minimizing memory contention. However, caches must be maintained coherent, since when a processor modifies a cached data, other caches might be left with an older version of the same data. To resolve this issue, snoop-cache coherence protocols are used to coordinate the many caches distributed throughout the system as part of providing a consistent view of memory.

1.4 Cache-Coherence Protocols

A snoop-cache coherence protocol provides a consistent view of memory by segmenting the shared address space into blocks and controlling the permissions for locally cached copies of these blocks. Since invalidation-based coherence has been used in favor of update-based coherence protocols in most recent systems (e.g.[5],[6],[7]), this thesis only focuses on invalidation-based cache coherence protocols.

Invalidation-based cache coherence protocols manage these permissions to enforce the coherence invariant. Informally, the coherence invariant states that

1. No processor may read a block while another processor is writing to the block
2. All readable copies must contain the same data.

In order to enforce this invariant, current protocols encode the specific permissions and other attributes of blocks in caches using a subset of the MODIFIED, OWNED, EXCLUSIVE, SHARED, and INVALID (MOESI) coherence states.Since these protocols constitute the main essence of the thesis, they are discussed in more detail in the following chapter.

These protocols use broadcast bus transactions and snoopy cache controllers in order to keep caches coherent. The general-purpose nature of this scheme results in significant power consumption, which prevents the utilization of these powerful platforms for energy constrained embedded applications. It has been reported [21] that the power due to snoop related cache lookup can amount to 40% of the total power consumed at the cache subsystem.

1.5 Motivation

Embedded processor and system customization has been shown to be a powerful approach for achieving tremendous performance and power improvements. The processor architecture and micro-architecture are fine-tuned to the needs of the particular application or application fragment. Since the entire application is statically known, and this knowledge lets us more intelligently place data structures in memory to improve the memory performance. The snoop-cache coherence schemes are general-purpose in their nature as no prior knowledge regarding the application structure and communication patterns, in particular, is available; it is assumed that all memory references can potentially access a shared data. This is a very conservative approach, where each bus transaction triggers a cache lookup for all the processors in order to find out whether a locally cached data needs to be invalidated, updated, or written back to the memory; thus leading to significant power consumption. Clearly, only a small fraction of all memory accesses refer to shared memory that need remote cache invalidation or update. This power overhead can be drastically reduced if information regarding the application communication patterns and ranges of shared memory is captured and utilized dynamically by the hardware.

This thesis makes the following contribution. We introduce and evaluate a customizable snoop-cache controller architecture, which can filter out most of the bus transactions and allow cache probing only for the memory accesses relevant to the shared data accessed by each processor node. This is achieved by precisely identifying the shared memory ranges for each task or critical region in the task, and providing this information to the operating system kernel and cache snoop controller for run-time utilization. As the proposed approach identifies application information regarding shared memory regions, the close cooperation of the thread package and operating system memory manager is required. This leads to an energy efficient system design.

1.6 Outline of thesis

This thesis is organized as follows. Chapter 2 describes the background information for the protocols and related work. Chapter 3 motivates the problem and outlines our approach. Chapter 4 describes the functional overview of the approach. Chapter 5 shows the hardware support we have suggested. Chapter 6 presents some preliminary results . Chapter 7 concludes with a summary and future work.

Chapter 2

Background and Related work

This chapter describes the background and basics for understanding the *cache-coherence* protocols which have been modified to implement our approach. It also mentions about related research work that focus on reducing energy in a shared memory system. The readers can also refer to established textbooks on this topic for further background and introductory material such as [3], [4]. Since invalidation-based coherence has been used in favor of update-based coherence protocols in most recent systems (e.g.[5],[6],[7],[8],[2]), this thesis only focuses on invalidation-based cache coherence protocols.

There are multiprocessor systems with multiple memory modules in which each processor has one or more levels of private cache memory. When processors in such a system share the same physical memory address space, they are called *shared-memory multiprocessors*. These systems manage the shared memory address space by dividing the memory into blocks. Processors cache copies of recently accessed data, to both reduce the average memory access latency and increase the effective bandwidth of the memory system. When a processor needs to cache a copy of a block that it can read, it issues a read request. When a processor needs a copy of the block it can both read and write, it issues a write request. A coherence transaction is the entire process of issuing a request and receiving any responses. Processors issue these requests to satisfy load or store instructions that miss in the cache, as well as for software or hardware non-binding prefetches. Requests and responses travel between processors over an interconnection network.

Allowing multiple processors to cache local copies of the same block results in the cache coherence problem, a problem solved by the introduction of a cache coherence protocol. The goal of a cache coherence protocol is to interact with the system's processors, caches, and memories to provide a consistent view of the contents of a shared address space. The exact definition of consistent view of memory is defined by a memory consistency model [9] specified as part of the instruction set architecture of the system. The simplest and most intuitive memory consistency model is sequential consistency [10]. In this thesis, we assume sequential consistency for both describing and evaluating coherence protocols.

As part of enforcing a consistency model, invalidation-based cache coherence protocols maintain a coherence invariant. The coherence invariant states the following for a block of shared memory:

1. Zero or more processors are allowed to read it.
2. Exactly one processor is allowed to write and read it.

2.1 States of Cache-Coherence Protocol

To enforce the coherence invariant, coherence protocols use protocol states to track read and write permissions of blocks present in processor caches. This section describes the well-established MOESI states [11] that provide a set of common states for reasoning about cache coherence protocols. A processor with a block in the *INVALID* or *I* state signifies that it may neither read nor write the block. When a block is not found in a *S* state signifies that a processor may read the block, but may not write it. A processor in the *MODIFIED* or *M* state may both read and write the block. These three states *(INVALID, SHARED, and MODIFIED)* are used to directly enforce the coherence invariant by

1. Only allowing a single processor to be in the *MODIFIED* state at a given time
2. Disallowing other processors to be in the *SHARED* state while any processor is in the *MODIFIED* state.

When a processor requests a new block, it often must evict a block currently in the cache. The optional *OWNED* or *O* state in a processor's cache allows read-only access to the block (much like *SHARED*), but also signifies that the value in main memory is incoherent or stale. Thus the processor in *OWNED* must update the memory before evicting a block. As with the *MODIFIED* state, only a single processor is allowed to be in the *OWNED* state at one time. Unlike *MODIFIED*, however, other processors are allowed to be in the *SHARED* state when one processor is in the *OWNED* state.The *OWNED* state can reduce system traffic by not requiring a processor to update memory when it transitions from *MODIFIED* to *SHARED* during a read request. In a protocol without the OWNED state, the responder would transition from *MODIFIED* to *SHARED*, both providing data to the requester and updating memory. If another processor issues a write request for the block before it is evicted from the *OWNED* processor's cache, memory traffic is

State	Load	Store	Eviction	Read Req	Write Req
MODIFIED	hit	hit	writeback ↓ INVALID	send data ↓ OWNED	send data ↓ INVALID
EXCLUSIVE	hit	hit	Evict ↓ INVALID	send data ↓ SHARED	send data ↓ INVALID
OWNED	hit	write request ↓ MODIFIED	writeback ↓ INVALID	send data	send data ↓ INVALID
SHARED	hit	write request ↓ MODIFIED	writeback ↓ INVALID	(none)	(none) ↓ INVALID
INVALID	read request (response:shared) → SHARED -or- read request (response:clean) → EXCLUSIVE	write request ↓ INVALID	(none)	(none)	(none)

Table 2.1: MOESI State Transitions

reduced. The final MOESI state is the *EXCLUSIVE* or *E* state. The *EXCLUSIVE* state is much like the *MODIFIED* state, except the *EXCLUSIVE* state implies the contents of memory match the contents of the *EXCLUSIVE* block. By distinguishing this clean *EXCLUSIVE* state from its corresponding dirty *MODIFIED* state, a block in *EXCLUSIVE* can be evicted without updating the block at the home memory. When no other processor is caching the block, the memory responds to a read request with a clean-data response. The requesting processor transitions to *EXCLUSIVE*, granting it read/write permission to the block without the added burden of updating memory. The block can be later be quickly written without an external coherence request by silently transitioning from *EXCLUSIVE* to *MODIFIED* (requiring a writeback upon subsequent eviction). The basic operation of a processor in an abstract MOESI coherence protocol is shown in Table 2.1.

2.2 Types of Protocols

Today, the two most common approaches to cache coherence are *snooping protocols* and *directory protocols* which use these basic states or subset of these states. Snooping protocols broadcast requests to all processors using a bus or bus-like interconnect. This ordered broadcast both

1. Unambiguously resolves potentially conflicting requests

2. Directly locates the block even when it is in another processor cache.

In contrast, directory protocols send requests only to the home memory which responds with data or forwards the request to one or more processors. This approach reduces bandwidth consumption, but increases the latency of some misses.

2.3 Snooping Protocols

The snooping protocols (mostly used in our approach), which is the most commonly used approach to building shared-memory multiprocessors. The key characteristic that distinguishes snooping protocols from other coherence protocols is their reliance on a “bus” or “virtual bus” interconnect. Early multiprocessors used a shared-wire, multi-drop bus to connect all processors and memory modules in the system. Snooping protocols exploit such a bus-based interconnect by relying on two properties of a bus:

1. All requests that appear on the bus are visible to all components connected to the bus (processors and memory modules).

2. All requests are visible to all components in the same total order (the order in which they gained access to the bus) [12].

In essence, a bus provides low-cost atomic broadcast of requests.

Once granted access to the bus, the processor puts its request on the bus, and the other processors listen or snoops the bus. The snooping processors transitions their state and may respond with data. The memory determines if it should respond by either storing state for each block in the memory or by observing the snoop responses generated by the processors. Only after the requesting processor receives its data response (completing its coherence transaction), another processor is allowed to initiate a request.

The advantage of snoop-based multiprocessors is the low average miss latency. Since a request is sent directly to all the other processors and memory modules in the system, the responder immediately knows that it should send a response. Second, bus-based snooping protocols are relatively simple.

The main disadvantage of snooping is that such protocols are still by nature broadcast-based protocols; i.e., protocols whose bandwidth requirements increase with the number of processors. Even after removing the bottleneck of a shared-wire bus or virtual bus, this broadcast requirement limits system scalability. To overcome this limitation, recent proposals [13],[14] attempt to reduce the bandwidth requirements of snooping by using destination-set prediction (also known as predictive multicast) instead of broadcasting all requests.

2.4 Related work

Reducing power consumption in system design has been widely studied by researchers at various levels: architecture, compiler, operating systems, applications, middleware. Low power processor and memory design has been an area of interest at architecture level design. With the increased popularity of shared memory multiprocessors, energy and power efficient design of these systems has caught attention of researchers. In particular, improving cache coherence efficiency has been studied at various levels.

Previous work on snoop energy reduction relies on techniques that exploits sharing patterns in snoop-based shared memory multiprocessors with potential applications in reducing bandwidth, latency and energy. In [17], the authors have proposed optimization targeted at snoop protocols. In Jetty scheme proposed by [16], each snoop avoids many snoop-induced lookups that would otherwise result in a miss. The authors have introduced a cache like structure, which dynamically identifies which remote memory references have been known to be not present in the local cache. Nodes maintain two structures that respectively represent a subset of blocks that are not cached (exclusive Jetty) and a superset of blocks that are cached (inclusive Jetty). The introduced table is updated each time the cache is probed by the snoop controller or new data is brought to the cache. One observation is that since the private caches are typically smaller in a chip multiprocessors, the reduction of the energy through the Jetty is to a large extent outweighed by the energy consumed in the Jetty.

There is a similar research proposal called RegionScout[16]. RegionScout is a technique that exploits coarse grain sharing patterns in snoop-based shared memory multiprocessors. RegionScout comprises of a family of filters that dynamically observe coarse grain sharing and allow nodes to detect in advance that a request will miss in all remote nodes. This technique is used to avoid snoop-induced tag lookups thereby reducing energy in the memory hierarchy. When a node sends a request and block-level sharing information, it receives region-level sharing information. If a region is identified as not shared subsequent requests for any block within the region from the same node are identified as non-shared without having to probe any other node. Such information is not available in conventional snoop or directory-based coherence. RegionScout filters utilize imprecise information about the regions that are cached in each node in the form of a hashed bit vector [15]. Every node keeps precise information about the pages it is caching. This information is used to form a page-level sharing vector in response to coherence requests. Subsequent requests are snooped only by those nodes that do have blocks within the same page and thus energy is reduced. The difference between these two related work is that with RegionScout a requesting node can determine in advance that a request would miss in all other nodes. With Jetty every node still snoops all requests.

RegionScout filters are Saldanha and Lipasti proposed serial snooping to reduce energy in shared multiprocessors [18].A number of other approaches that reduce energy dissipation use either cache resizing, circuit techniques for reducing on chip power or energy efficient architectures [19],[30],[31],[32] to filter memory references.

There is some related work that helps reduce power consumption in snoop-based cache coherence. In snoop-based design which employs various forms of speculation to reduce cache miss latency and improve performance, energy reduction can be done by using serial snooping for load misses[19]. In this schemes the authors try to eliminating unnecessary activity in a bus interconnect of a multiprocessor at the architectural level. The serial snooping scheme is based on the assumption that if a miss occurs in one cache, it is possible to find the block in another cache without having to check all the other caches. The authors model an interconnect in the form of a tree with point-to-point connections and a memory controller at the root level. Instead of broadcasting the snoop-transaction to all of the processors in parallel, the caches are checked serially in a system based on serial snooping. The basic idea is to prevent wasting power unnecessarily by transmitting snoop packets to nodes that either do not have a copy of the data or nodes that have a copy but are not responsible for sourcing the data as the result of a snoop. It works by initially transmitting a snoop packet only to the nearest node. This node then does a tag comparison and if it finds the requested block in M(Modified), S(Shared) or E(Exclusive) state it sources the data to the requester and snoop transaction ends without either the memory or any of the other remote nodes seeing the transaction. On the other hand, if the nearest neighbor is unable to satisfy the request, it forwards the request to the next level in the tree hierarchy. The serial snooping is ineffective because in most of the cases all caches have to be checked before it can be concluded that they cannot respond to the request on the common bus.

A comparative evaluation of the above two proposed techniques to reduce snoop-induced power in multiprocessors: serial snooping[19] and Jetty [17] has been studied by authors in [20]. These two techniques were aimed at SMP-servers with two levels of private caches and they have been analyzed in the new context of chip-multiprocessors in [20].

One more related work reports energy savings using Page Sharing Table (PST) scheme [21] which is based on the intuition that there exist a fair number of pages that are not shared. A page is said to be loaded in a processor if at least one block that belongs to the page is loaded in the private cache. For non-shared pages, blocks are not subject to coherence actions and snooping overhead could be eliminated. A unit called Page Sharing Table (PST) is attached to each processor which keeps track of which pages are currently used by the processor. For each such page, the unit keeps a sharing vector that indicates if the other processors also share the page. This sharing vector is broadcast on a separate bus, called sharing vector bus, with as many lines as the number of processors, on a snoop-broadcast action. By reading this sharing vector, the other caches know whether they need to do a tag-lookup to check for the block or not. But PST itself wastes some energy.

2.5 Our work

This thesis work is based on MOSI coherence protocol which includes all the required basic states. It can also be extended to other protocols like MSI, MOESI and MSI. This thesis work focuses on filtering snoop requests based on a deterministic knowledge regarding the shared memory regions of each task resulting in significant energy savings. It uses application knowledge instead of introducing sophisticated hardware structures like Jetty or non-shared region tables. It works in close alliance with compiler and OS support to extract information regarding presence of shared data access patterns for parallel tasks in application software.

In a bus-based interconnect where all system components are connected to the same set of physical wires. A component sends a message by

- Arbitrating for the bus (to avoid having multiple processors driving the bus at the same time),
- Driving themes sage on the bus (allowing all components on the bus to observe the message).

Since all components can observe or “snoop” transactions on the bus, such interconnects support broadcast with little additional cost. Such an interconnect provides point-to-point ordering when all messages sent between a pair of processor arrive in the order in which they were sent. This protocol is modified to include information from the operating system kernel and cache snoop controller to filter out most of the irrelevant bus transactions. This results in cache probing only for the memory accesses relevant to the shared data accessed by each processor node to save power consumed due to unnecessary snoop activity on the bus.

Chapter 3

Motivation

With the increased performance demand from many embedded applications, such as in the multimedia and the communication domains, multiple processing cores are utilized in implementing such embedded systems. Multiprocessing hardware implementation is quite natural as the majority of embedded applications, especially in the multimedia and communication domains, exhibit a significant amount of task-level parallelism.

Various forms of multiprocessing configuration and interconnect topologies exist. However, the simplest and the most cost-efficient one is the bus-based, shared memory multiprocessor platform. The advantages of such a system are its simple and well-understood programming model with low communication latency. Additional benefit of this multiprocessor organization is that multi-threading and any uniprocessor system software, in general, can be easily extended for bus-based shared memory multiprocessor. This is due to the fact that the physical memory is shared amongst all the processors, and thus all system code and data structures are placed in that memory. The processor cores simply provide multiple hardware contexts to the shared system software layer. The only modification needed to the uniprocessor system code, is that it be made re-entrant and protected against multiple system threads executing and trying to access the data structures simultaneously.

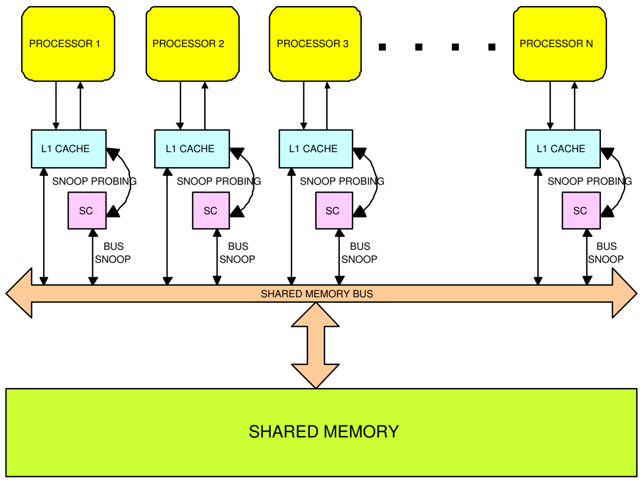


Figure 3.1: Snooping in cache-coherent multiprocessor

The common bus in these systems, however, can quickly become a bottleneck as each access to the shared memory has to be performed through the bus. A common practice to resolve this problem is to employ local caches at each processor node. In this way, the data is replicated and brought closer to the processors. Not only is the amount of traffic on the bus reduced, but also bank conflicts on the shared memory are eliminated. Caching, however, introduces the fundamental problem of incoherent data stored in the local caches. This problem can be easily appreciated if one considers a situation where a shared word is brought for read access in the local caches of two processors, and subsequently, one of the processor writes to this data and modifies its value in the local cache and/or main memory. At this moment the other processor remains with the old copy of that data stored in its cache. The cache coherence problem exists with both write-through and write-back caches. In bus-based shared memory systems, write-back caches are typically used, since the goal is to minimize the bus utilization as much as possible. The trade-offs between write-through and write-back caches for shared memory multiprocessor systems on chip are explored in detail in [34]. Employing write-back caches further exacerbates the cache coherence problems, as write-back cache introduce an additional case of incoherence, where the most recent copy of a data can be present only in one of the caches. In such a case, both the main memory and potentially other caches have an old version of that data.

To resolve the cache coherence problems, coherence protocols have been introduced for general-purpose multiprocessor systems. As the common bus is inherently a broadcast medium, a snoop-based cache coherence protocols are being used in general. The fundamental principle of these protocols is that each memory reference placed on the bus by a processor is detected by all the snoop controllers in the system, and each one of them probes its local cache to check whether the data requested through the shared bus from the memory happens to be present in the local cache. If yes, then depending on the type of request and the state of that data in the local cache, different actions must be undertaken.

The general architecture of the bus-based shared memory multiprocessor with support for snoop cache coherence is shown in Figure 3.1. Each cache is associated with a snoop controller and each cache line has its own state. The snoop controllers monitor the system bus for read and write misses that are generated by the processors in the system. Such memory requests are generated by the processors when the needed data is not present in the local cache.

Fundamentally, the purpose of the snoop controllers is to react to any such memory requests on the bus. For instance, for a read-miss on the bus, the snoop

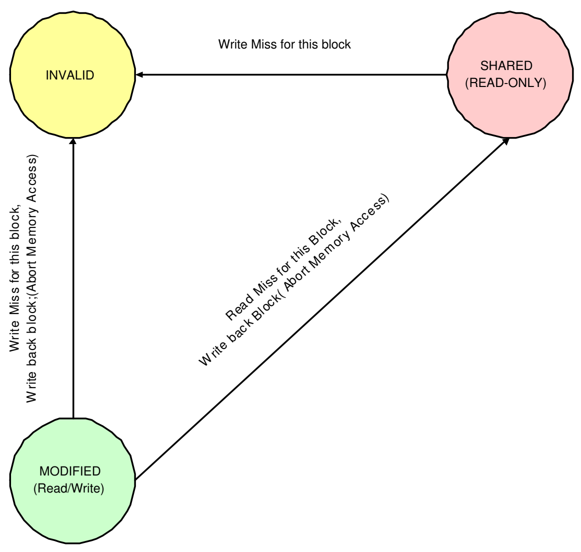

INVALID
Write Miss for this block
SHARED (READ-ONLY)
Write Miss for this block, Write back block;(Abort Memory Access)
Read Miss for this Block, Write back Block(Abort Memory Access)
MODIFIED (Read/Write)

Figure 3.2: State machine for bus requests

controllers for all nodes must probe their local caches to check whether the requested data is present and modified but not yet written-back to main memory. In such a case, the corresponding cache would be the only place in the system where the data can be found. Alternatively, when a processor modifies a data in its local cache, it needs to place a write-miss on the common bus (even when the data is already in the local cache) in order to notify all the other processors to invalidate this data if it happens to be present in some of the other caches as well. When the snoop controllers on all the processors detect a write-miss transaction on the bus, the local caches are probed in order to invalidate this data if it is cached locally.

Several versions of snoop-based cache coherence protocol exists; nonetheless, all of them are based on the same principle. The snoop controller monitors all bus transactions; on write-miss it probes the local cache to invalidate a copy of the data if it is present, and on read-miss, the cache is probed to check if the requested data is cached and modified locally so that it can be written back to the memory and provided to the requesting processor node as well. Additionally, all snoop protocols extend the status of each cache line to indicate the current state of the cache line. Most of the snoop cache coherence protocols, maintain the state of *Invalid*, *Shared Read-Only*, and *Modified* [33]. The snoop controller, considering the type of memory request on the bus and the current state of the locally cached copy of that data, if present, decides on how to change the state. The state transition diagram for one such cache coherence protocol is shown in Figure 3.2. The *Read-Only* state is assigned to all cache lines, which store a read-only copy of the cached data, while the *Modified* state indicates that the cache line has been recently written to and that the data in this cache line must not be present in other caches. The diagram also shows by what type of memory requests on the bus each state transitions is triggered. For instance, on a read miss generated by a processor in the system, the address of data along with the processor *ID* that generated the miss is put on the shared bus. This results in a state transition from *Modified* to *Shared* state in the cache where this data has been most recently modified. Similarly on a write miss on the bus, the snoop controller probes the cache and if the data is present it changes its state to *Invalid*.

Probing the local cache to identify whether an address requested through the common bus is present in the local cache entails an almost full cache lookup. In this probing, the tag arrays of the cache structure need to be accessed. The tags stored in all the associativity ways need to be read and compared with the actual tag of the address present on the bus. Such snoop induced cache lookups for each memory request are the major contributing factor to the excessive power consumption of the

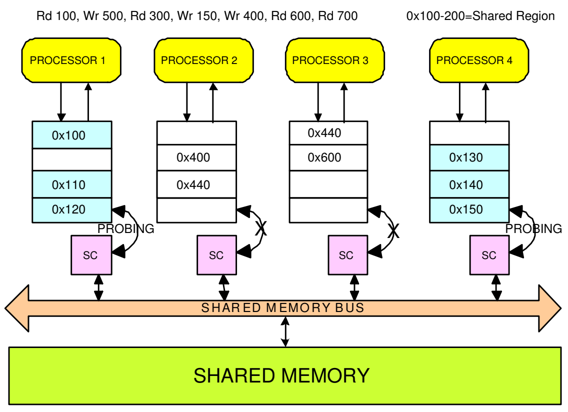


Figure 3.3: Multiprocessor system with a known shared region

cache coherence protocols.

It can be immediately observed from this brief description of snoop-based cache coherence protocols, that these protocols are general-purpose in their nature. It is conservatively assumed that each memory request on the common bus is a request to a possibly shared data; hence, the request needs to be handled appropriately. These protocols are designed to work for arbitrary workload where no prior knowledge regarding the shared memory regions or communication patterns in the applications is assumed. However, if application knowledge regarding the shared memory regions of the tasks running on each processor node is made available to their snoop controllers, a large amount of snoop-related cache probing can be eliminated. Consider for example, that in a shared memory multiprocessor system, a set of tasks is running, which happen to communicate through a single shared memory region from address 100 to address 200. If such a knowledge is made available to the snoop controllers and they are made capable of efficiently identifying whether the address of the current memory request is within that region, the snoop controller can decide to filter completely all memory requests outside of this region. This can be done, since all the snoop related cache line probing, invalidation, and write-back are only needed for the case where the corresponding data is actively shared between processor nodes. Such a situation is illustrated in Figure 3.3. For instance, if the processors generate a sequence of memory requests, such as *Rd 100*, *Wr 500*, *Rd 300*, *Wr 150*, *Wr 400*, *Rd 600*, *Rd 700*, the general-purpose snoop controllers would probe the local caches for each one of them. It is clear, though, that the cache lookups for all but address 100 and 150 would miss. Only the references to addresses 100 and 150 present a potential risk for cache coherence, since they reside withing the shared memory region used for communication in this particular example. If the snoop controllers are made aware of this, then cache probing for addresses 100 and 150 would only be performed, while for the rest of the addresses, no activity is necessary. Such filtering that is based on a deterministic knowledge regarding the shared memory regions of each task would result in significant power savings.

Chapter 4

Functional Overview

In shared memory multiprocessors, application tasks and processor nodes communicate through shared memory. At application level, parallel processes or threads are created by the software developer in order to utilize the underlying multiprocessor platform. Shared memory regions are allocated if required, and tasks are a fixed protocol of accessing it. In the case of multi-threading, this happens implicitly. The threads in case of multi-threading run in the same address space (as they are spawned by the same parent process). In the case of processes executing in different address space, OS facilities are used to map the shared address space into the address space of the processes involved in sharing. For example, shared memory/message queues/FIFOs are interfaces provided by the OS. Thus, in case of different address space sharing memory, the OS takes care of memory mapping.

Information regarding shared data access for each parallel task is readily available when an application software is developed and compiled for the underlying system. For instance, it may be the case that only one of the global arrays is used by a particular task as an input buffer, and one for an output buffer, while all other memory references of that task are to private data. However, this information is lost when the application is transformed into a binary form and loaded into the system. The only events observable from the thread library and the operating system are the creation of threads and utilization of synchronization primitives. What is left at hardware level is simply memory references, which the memory system (hardware) needs to handle assuming that they can refer to any possible memory location.

In the proposed methodology, we make the information regarding shared memory utilization available down to the hardware level, where the snoop controller can judiciously utilize it and filter out all the memory references, which do not refer to a shared memory region of interest to the local processor node. The information is transferred from the application to the system software, which in turn utilizes it to identify the physical page frames, which belong to the shared regions.

4.1 Compiler and Application Support

As part of the task creation the compiler or the software developer makes sure to inform the thread library or the operating system the list of global arrays which should be treated as a shared memory region for a particular task. This can be easily achieved in multiple ways, one of them being to include a pointer to the beginning of the global array and its size when calling the primitive for creating and starting a new thread. In this way, any application will explicitly inform the underlying systems software that only memory references to specified global arrays must be treated as references to shared memory; all other memory references are private for that task; hence, no other processor in the system can generate a valid reference to them.

4.2 System Software Support

The memory manager module, which is usually a part of the system software is the component responsible for allocating the data into the physical memory. The application executes in a virtual address space, which is mapped to the available physical memory. The memory manager maintains a page table, which provides the mapping from virtual memory pages to physical memory frames. The most frequent translations are always cached in the hardware *Translation Look aside Buffer* (TLB), so that the hardware can translate the address generated by the processor quickly with no intervention of the operating system. When a parallel application task is created, the information regarding its shared global array is made available to the operating system. At this stage, the memory manager identifies the set of physical memory frames, which correspond to each shared array of the task. Additionally, a unique identifier is provided for each such shared array/region in the system, which is utilized by the hardware in order to efficiently determine if a given memory reference bus transaction refers to a shared memory region. Note that the bit-width of this unique identifier depends on the maximal number of shared regions. For example, if there are 6 shared regions then a maximum of $\lceil log_2 6 \rceil$ bits $= 3$ bits are necessary to identify to which shared region does this given memory reference belong to. In practice, one of the identifiers (for instance, 0) is used for the memory references, which do not belong to any shared region.

Figure 4.1 illustrates an example, where four regions are defined. A region corresponds to a consecutive set of pages. At application level most often a shared

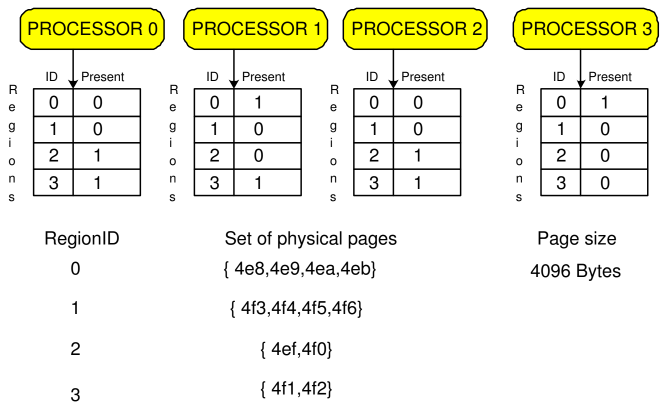


Figure 4.1: Processors and Region ID

region corresponds to a global array; our granularity of forming shared memory regions is at page level.

In order to find all the shared regions we could do one of the following: In the first case we make a list of addresses that are shared by all the processors. In order to accomplish this, we need to find the exact match for the starting address and the end address of the shared array that may span across several pages. Each of these addresses could then be represented as shared regions. However, this process will result in excessive hardware overheads in order to maintain start, end addresses and do an equality check on every access.

We therefore chose an alternate approach in which we capture the intervals of addresses that are shared. The granularity at which these intervals are chosen is at the "page level". The choice of using page level granularity is native to computer systems and makes the design simpler as hardware and operating systems are designed around this level of granularity. This leads to a major savings in hardware by not having to redesign hardware circuitry that keeps track of address ranges. Only a small hardware addition is required as will be explained shortly. A direct consequence of using page level granularity (as opposed to keeping track of ranges) is that it requires fewer number of bits.

Consequently, each region is assigned a unique ID and each physical page is associated with one such ID. The region ID is associated with each translation entry at the page table and is also stored in the hardware translation table.

The shared pages can be identified from the virtual address by doing simple bit shifting and masking operations. The virtual addresses are translated into physical addresses and subsequently passed to the OS for allocation. The OS allocates physical pages. In our approach, the OS can be modified to include additional information regarding the region id. It is to be noted that in our experimental results, we refer to each logical shared array as a region. (In our experiments, we have found out that there can be a maximum of 8 such shared regions).

On each cache miss the address of the required data is put on the shared bus. When the processor generates an address, which is virtual, this address is translated to a physical address by the TLB. Both the physical page number and the associated unique region identifier are extracted from the TLB. At this stage, we can annotate each memory reference with the region identifier of the shared region to which it belongs. If all the snoop controllers in the system can observe this region identifier for each memory request on the system, a very small and efficient hardware would suffice to check whether that region ID matches with the shared regions for the particular processor node. Providing the shared region identifier to all the processor nodes can be performed very easily by placing it on the common bus together with the memory request. It can be easily observed that all memory requests, which are related to cache coherence are mostly cache misses. These memory requests use only the address lines from the common bus, since they need to either fetch a new data or inform every other processor node that they are modifying a data in their local cache. Consequently, the region identifier can be included as a part of the bus transaction by simply using the available data lines in the shared bus; no additional bus lines are needed to transfer the region ID and hence no hardware modifications on the bus structure are needed. Now, the snoop controllers can directly observe the identifier of each snoop related memory request, and easily determine which ones refer to a shared region of their local processor.

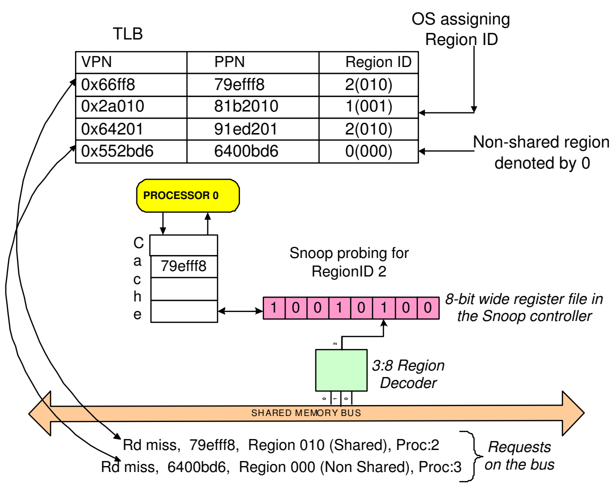
Chapter 5

Hardware Support

The purpose of the hardware support is to capture the set of shared regions of each task currently executing on a given processor in the system. This information becomes a part of the state of the task and is loaded by the operating system or the thread library when the parallel task is scheduled for execution. During program execution the system takes responsibility for translating the programś virtual addresses for instructions and data into the real addresses that are needed to get the instructions and data from the main memory. The system keeps the real addresses of recently accessed virtual-memory pages in a cache called the translation look-aside buffer (TLB). Each memory reference during program execution is put on the shared bus which also has information of this region identifier of the shared region associated with it from the TLB.

If all the snoop controllers in the system can observe this region identifier for each memory request on the system, a very small and efficient hardware would suffice to check whether that region ID matches with the shared regions for the particular processor node.

Such a hardware support could be implemented by using a set comparators in each processor node. When a shared region id is sent on the bus then each of these n bits are compared with the bits of all the shared regions for that task in the



VPN	PPN	Region ID
0x66ff8	79efff8	2(010)
0x2a010	81b2010	1(001)
0x64201	91ed201	2(010)
0x552bd6	6400bd6	0(000)

TLB
OS assigning Region ID
Non-shared region denoted by 0
PROCESSOR 0
Cache
79efff8
Snoop probing for RegionID 2
1 0 0 1 0 1 0 0
8-bit wide register file in the Snoop controller
3:8 Region Decoder
SHARED MEMORY BUS
Rd miss, 79efff8, Region 010 (Shared), Proc:2
Rd miss, 6400bd6, Region 000 (Non Shared), Proc:3
Requests on the bus

Figure 5.1: TLB and hardware logic

local processor cache. This is done by using a n bit comparator inside the snoop controller. Such comparisons are time consuming and may in fact consume more energy as opposed to saving energy on cache probes.

Therefore we recommend an alternative approach. Since the only information that the snoop controllers require is a status bit, indicating whether a region identifier is part of the shared regions of the local processor, this can be easily implemented by a bit-mask register. The hardware register is basically an n bit register where n is the number of shared regions and $log_2 n$ bits are used to denote the region ID. Our experimental results indicate that 8 shared regions are enough for all the benchmarks we have used. Each bit of the n bit register indicates whether the region with ID equal to the bit index is a shared region for that task or not. For example if region ID is 3 then the fourth LSB is flagged as 1. Note that region identifiers are assigned the values from 0 to n. The proposed hardware architecture is depicted in Figure 5.1.

When a bus request is seen by the snoop controller, the region ID is used as an index to check the values of the corresponding bit in the bit-mask register. In order to accomplish this, a simple decoder circuit is required, which in the case of 8 or 16 regions is trivial in size, power, and delay. If the bit in the register is set, the snoop controller probes the cache. Otherwise, no cache probing is needed as the address on bus is either a private address of a remote node or a shared address that is not operated by the particular local processor node. It is noteworthy, that the introduced hardware is extremely cost efficient, as it constitutes one bit-mask registers, whose bits are indexed with the region ID of the snoop related bus transactions.

One other question that can be raised due to this hardware support scheme is the need of additional wires for the shared bus. Since the region ID information is also a part of the memory request and is placed on the system bus, there can arise concerns about if the bus bandwidth limits the performance or increases overhead when additional information(such as region ID) is transmitted on the bus. If this is the case then we would require additional wires to widen the bus.

Every bus is composed of two distinct parts: the data bus and the address bus. The data bus are the lines that actually carry the data being transferred. The address bus is the set of lines that carry information about where in memory the data is to be transferred to or from. In addition, there are a number of control lines that control how the bus functions, and allow users of the bus to signal when data is available. In an event of a read miss, the address lines of the bus are used to fetch the instruction from the memory keeping the data lines free. In an event of a write miss, the address that needs to be written is placed on the address lines of the bus and is transferred to all other local caches keeping the data lines of the bus free. If the local caches do not have this memory address then the processor writes into the address. Only in case of a write-invalidation; when a write miss is generated by a processor and other caches have copies of the address location, the address lines of the bus are used to notify other caches to invalidate this address. And, after writing into the address location the processor uses the data bus lines to transfer the data and update this data in the main memory. As we can see these memory requests use only the address lines from the common bus, since they need to either fetch a new data or inform every other processor node that they are modifying a data in their local cache. Taking advantage of the fact that the data lines in the shared bus are free, we can use those lines to transmit the region ID information without having to add additional wires to the existing shared bus.

Chapter 6

Experimental Results

We have evaluated the proposed approach on benchmarks chosen from the SPLASH-2 benchmark suite [22]. The benchmarks were chosen from the suite because of the realistic workload provided by the kernels and the benchmark's inherent property to expose application parallelism for shared-memory multiprocessor systems. Specifically, we chose FFT, LU and $RADIX$ kernels out of the SPLASH-2 benchmark suite.

The LU kernel factors a dense matrix into the product of lower triangular and upper triangular matrices. In our case, we have used LU to decompose a data set consisting of 128×128 matrix. The FFT data set consists of 2048 complex data points to be transformed, and another set of 2048 complex data points containing the roots of unity. The $RADIX$ kernel implements the traditional radix sort. In our experiments, we have used $RADIX$ to sort 3072 keys. It is to be noted that increasing the size of data sets in each of these kernels results in an increased number of shared pages per region.

We simulated our target platform using $Simics - 2.0.25$ [35] functional simulator. Simics is a full system simulation platform capable of running an unmodified commercial operating systems on top of a simulated multiprocessor machine. For our experimental results, we simulated a 4-processor system running sparc proces-

		Node 0	Node 1	Node 2	Node 3
fft sa	RM	28143	4280	4225	5838
	WM	9084	3202	2442	4556
	Total	37227	7482	6667	10394
fft dm	RM	14913	12070	5061	4755
	WM	9711	3583	3266	4380
	Total	24624	15653	8327	9135
lu sa	RM	12073	5855	6842	8880
	WM	8103	4199	4571	7670
	Total	20176	10054	11413	16550
lu dm	RM	48483	25789	12902	10201
	WM	22648	7169	9356	7456
	Total	71131	32958	22258	17657
radix sa	RM	11540	5407	5173	5105
	WM	9383	5487	4774	6180
	Total	20923	10894	9947	11285
radix dm	RM	17030	16665	4394	5846
	WM	12369	6291	3929	6428
	Total	29399	22956	8323	12274

Figure 6.1: Read/Write misses per processor

	Sh Region ID	Pages
fft	1	3
	2	2
	3	21
	4	21
lu	1	32
	2	1
radix	1	7
	2	9
	3	5

Figure 6.2: Shared regions in benchmarks

sors and solaris operating system. As our approach focuses on the memory system, the choice of particular RISC instruction set architecture makes no difference for the methodology, which we propose. We have used *Ruby* [36] as a memory simulator which plays the role of a detailed memory system simulator, including shared memory, communication bus, local caches, and snoop controllers. Since we target the memory system, the ruby simulator is our main driver.

In order to obtain the experimental results, we added instrumentation code

	Total Misses	Sh RM	Sh WM
fft sa	14243	5146	2775
fft dm	15461	5332	2757
lu sa	18073	9421	4743
lu dm	28080	15211	7142
radix sa	10341	1273	1546
radix dm	11358	1441	1874

Figure 6.3: Shared Read/Write misses

to the cache coherence protocol module to generate our desired statistics. We have also inserted additional code in the benchmark suite to obtain the virtual addresses of the shared memory regions. These virtual addresses were translated into physical addresses inside the simulator. The region based statistics is thus based on actual physical addresses. The physical addresses (pages) that belong to shared regions were assigned a unique region ID. Further, the region ID is used to match the physical address requests being generated by the bus requests in our modification to the snoop controller simulation module.

For our baseline architecture, we have performed experiments on a 32K direct mapped (DM) and a 2-way set associative (SA) L1 cache. The results in terms of total misses, read misses(RM) and write misses(WM) per processor node are given in Figure 6.1.

For each of the benchmarks we identified the shared arrays by inspecting the benchmark code. We calculated the size of these arrays and divided them into physical pages where page size is 4096 bytes. Now these shared arrays were annotated by a region ID. Since the size of the logically shared arrays varies on the input data set, the number of pages will also vary according to the input data size. For our benchmarks we got between 1 to 5 shared regions per benchmark, each of which is a

Cache	Total	Decode Tag	Wordline	Bitline	Sense_amp Tag
32KB 2way	0.112	0.016	0.00054	0.020	0.0754
32KB DM	0.107	0.016	0.00052	0.0192	0.0708

Figure 6.4: Energy statistics per access (nJ)

	Total Energy (nJ)	Shared Region Energy (nJ)	Percentage Energy Savings
fft sa	1597.49	888.42	44.39
fft dm	1651.02	863.79	47.68
lu sa	2024.18	1588.63	21.63
lu dm	2998.55	2386.99	20.40
radix sa	1159.85	316.18	72.74
radix dm	1212.87	353.99	70.8

Figure 6.5: Energy savings per benchmark

set of shared physical pages. The region ID's and the number of physical pages per region are listed in Figure 6.2. In our approach we are filtering the snoop requests based on the addresses of the read or write misses available on the shared bus. For this we will need to figure out how many of these addresses lie in a shared region. We compare the physical page number of the addresses on the bus with the physical page number of the shared arrays and then keep a count of the number of addresses that have the same page number as the shared arrays. We divide this count into shared read misses and shared write misses by taking into account the type of miss that was placed on the bus. These statistics are shown in Figure 6.3. With our approach we will probe the cache only if the addresses lie in a shared region.

The energy consumption per access for the tag arrays of data caches is measured using the CACTI tool [37]. We do it for our baseline architecture of 32KB cache. Figure 6.4 lists the numbers in nJ for the energy consumption of dedicated tag arrays used in the snoop activity. The total energy consumption reported is the energy dissipated by the tag arrays.

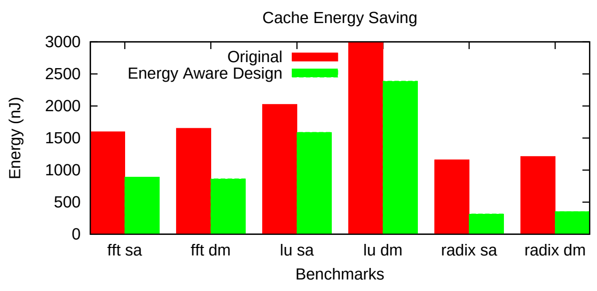


Figure 6.6: Energy savings

The percentage of energy savings per benchmark is calculated in Figure 6.5. These numbers are plotted in Figure 6.6. The higher bar represents the energy consumed in nJ for the baseline architecture and the second bar represents the energy consumed in nJ after filtering the unnecessary snoop activities. Here we show and compare the energy dissipated by the snoop activity. As we discussed earlier in the paper, the energy contribution of the snoop operations to the total energy of the memory system can be very high and depends on the cache/memory sizes and bus organization. This pattern is repeated for all the benchmarks. From the results we can see that we get the maximum reduction in FFT and $RADIX$ benchmarks. However, the energy reduction in LU is not that significant because of two reasons. Firstly, LU loops and threads operate on a large part of shared regions as compared to FFT or $RADIX$. Since the number of private regions in LU is small, the energy savings is not as high as the ones achieved for the other benchmarks. Secondly, our filtering process is conservative as it operates at page level granularity at the moment. We filter shared addresses based on the physical page number of the address, hence some private data, which belong to that page would also trigger a snoop cache probing.

Chapter 7

Conclusion

In this thesis, we have presented a low-power methodology for maintaining caches coherent in an embedded multi-processor system. The proposed approach exploits application information regarding shared memory regions of the communicating tasks in order to eliminate a large number of power consuming snoop-induced cache probing. The proposed methodology is very cost-efficient as the required additions to the system software and the hardware architectures are minimal and impose no performance or area overheads. Such an approach would be of great utility to a large number of modern embedded applications, for which both high-performance and low-power are of great importance.

Appendix A

Data Structures

The data structures declared in `CacheMemory.h` are explained in this chapter. These data structures are declared as either public or private variables members of class `CacheMemory`.

```
int *m_rd_miss;
int *m_wr_miss;
int *m_rd_hits;
int *m_wr_hits;
int *m_snoop_miss;
int *m_snoop_hits;

int *m_shared_write;
int *m_shared_readonly;
int *m_rd_hits_inMstate[2];
int *m_rd_miss_inMstate[2];
int *m_wr_hits_inMstate[2] ;
int *m_wr_miss_inMstate[2] ;
int m_total_miss;
int m_region_size;

struct physical_stat {
        char varname[20];
        int offset;
        unsigned long pa;
        int region;
};

list<struct physical_stat> pa_list;
typedef std::list<struct physical_stat> physical_stat_t;

struct rd_wr_stat {
        unsigned long priv_miss;
        unsigned long shared_rd_miss;
        unsigned long shared_wr_miss;
        unsigned long page;
};
struct rd_wr_stat rw_page_stats[NUM_PROCS];

struct snoop_stat {
        unsigned long hits[NUM_PROCS];
        unsigned long misses[NUM_PROCS];
};
struct snoop_stat snoop_stats_gets[NUM_PROCS], snoop_stats_getx[NUM_PROCS];

struct region_stat {
        unsigned long gets_hits;
        unsigned long gets_miss;
        unsigned long getx_hits;
        unsigned long getx_miss;
};
struct region_stat *per_reg_stat[NUM_PROCS];
```

Appendix B

Useful Functions

The following functions inside `CacheMemory.h` operate on the data structures mentioned in the previous chapter.

```
bool isReadHit(const Address& address, CoherenceRequestType type, MachineID m);
void printOurStats(ostream &out);
void printPhysicalAddrStats(ostream &out, physical_address_t pa, int rw);
void intoMstate(MachineID m);
void outMstate(MachineID m);
bool getphysicaladdr(const Address& address, CacheRequestType type);
bool rangeCheck(const Address& address, CoherenceRequestType type, MachineID m);
```

Appendix C

Recompiling Simics

Simics is installed on `dogbert.eng.umd.edu`. This chapter shows how to run simics on dogbert as user-id `res`. The following commands need to be run from `/home/res/TESTDIR`.

C.1 Compiling Simics

Simics needs to be recompiled if there are any modifications done on the sources. The following commands can be used to recompile simics assuming there are some changes made to ruby, the memory module simulator:

1. `cd /home/res/ruby/TESTDIR/ruby`
2. `make PROTOCOL=`

   `MOSI_SMP_bcast_1level DESTINATION=MOSI_SMP_bcast_1level`

Note that the above commands are also encapuslated inside a script called `runmake` under `/home/res/ruby/TESTDIR`.

C.2 Running Simics

1. `cd /home/res/TESTDIR/simics/home/MOSI_SMP_bcast_1level`
2. `setenv SIMICS_EXTRA_LIB ./modules`

3. `cd /home/res/TESTDIR/simics/home/MOSI_SMP_bcast_1level`

4. `./simics`

At this point simics should startup with the simics prompt. The following simics commands need to be issued at simics prompt in order to configure simics before running any simulations.

1. `read-configuration`

   `../../../../GEMS/simics/home/sarek/four_cpu_checkpoint`

2. `instruction-fetch-mode instruction-fetch-trace`

3. `istc-disable`

4. `dstc-disable`

5. `load-module ruby`

6. `ruby0.setparam g_NUM_PROCESSORS 4`

7. `ruby0.init`

8. `c`

At this point a console window should pop-up simulating the boot sequence of a solaris 5.8 OS running on four processor SPARC system.

BIBLIOGRAPHY

[1] M. Ekman, F. Dahlgren and P. Stenstrom, .TLB and snoop energy-reduction using virtual caches in lowpower chip-microprocessors., in ISLPED, pp. 243. 246, August 2002.

[2] http://www.cis.upenn.edu/ milom/papers/milo_martin_phd.pdf

[3] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, third edition, 2003.

[4] K. Hwang and F. A. Briggs. Computer Architecture and Parallel Processing. McGraw-Hill, Inc. 1984.

[5] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability Port: A Coherent Interface for Shared Memory Multiprocessors. In Proceedings of the 10th Hot Interconnects Symposium, pages 6570, Aug. 2002. URL http://www.hoti.org/archive/ hoti10/program/Kumar_ScalabilityPort.pdf.

[6] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. EXA Cache/Scalability Controllers. In IBM Enterprise X-Architecture Technology: Reaching the Summit, pages 3750. International Business Machines, 2002.

[7] A. Charlesworth. Starfire: Extending the SMP Envelope. IEEE Micro, 18(1):3949, Jan/Feb 1998

[8] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 241-251, June 1997.

[9] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. IEEE Computer, 29(12):66-76, Dec. 1996.

[10] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, C-28(9):690-691, Sept. 1979.

[11] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In Proceedings of the 13th Annual International Symposium on Computer Architecture, pages 414423, June 1986.

[12] D. E. Culler and J. Singh. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, Inc., 1999.

[13] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In Proceedings of the 26th Annual International Symposium on Computer Architecture, pages 294304, May 1999.

[14] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination- Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors. In Proceedings of the 30th Annual International Symposium on Computer Architecture, pages 206217, June 2003.

[15] B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of ACM, pages 13(7):422-426, July 1970.

[16] Moshovos, A., RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. Proceedings of the 32nd Annual International Symposium on Computer Architecture(ISCA). 2005.

[17] Moshovos, A., Memik, G., Falsafi, B., and Choudhary, A.JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. Proceedings of 7th International Symposium on High-Performance Computer Architecture(HPCA), 2001.

[18] C. Saldanha and M. H. Lipasti, Power Efficient Cache Coherence, High Performance Memory Systems, edited by H. Hadimiouglu, D. Kaeli, J.Kuskin, A. Nanda, and J. Torrellas, Springer-Verlag, 2003.

[19] C. Saldanha and M. Lipasti, Power Efficient Cache Coherence, Workshop on Memory Performance Issues, in conjunction with ISCA, June 2001

[20] Magnus Ekman, Fredrik Dahlgren and Per Stenstrm, Workshop on Duplicating, Deconstructing, and Debunking in conjunction with ISCA-2002

[21] Magnus Ekman, Fredrik Dahlgren and Per Stenstrm, "TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip Multiprocessors", International Symposium on Low-Power Electronics and Design, 2002

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. Proceedings of the 22th International Symposium on Computer Architecture, pages 24-36, June 1995.

[23] C. Kulkarni, F. Catthoor, and H. De Man, Cache Optimization for Multimedia Compilation on Embedded Processors for Low Power, ACM/IEEE Proc.

Parallel Processing Symp. (IPPS), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 292-29

[24] P.R. Panda, N.D. Dutt, and A. Nicolau, Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration, Kluwer Academic Publishers, Norwell, Mass., 1999.

[25] W.-T. Shiue and C. Chakrabarti, Memory Exploration for Low Power Embedded Systems, Proc. Design Automation Conf., ACM Press, New York, 1999, pp. 140-145.

[26] P.R. Panda, N.D. Dutt, and A. Nicolau, Local Memory Exploration and Optimization in Embedded Systems, IEEE Trans. Computer-Aided Design, vol. 18, no. 1, Jan. 1999, pp. 3-13.

[27] P. Stenstrom, A Survey of Cache Coherence Schemes for Multiprocessors, IEEE Computer, Vol. 23, No. 6, June 1990, pp. 1224.

[28] S. Richardson, MPOC: A Chip Multiprocessor for Embedded Systems,, HP Technical Report, HPL-2002-186, July 2002.

[29] L Benini, GD Micheli, Networks on Chips: A new paradigm for System on Chip Design DATE, 2002

[30] J. Choi, J. Lee, S. Jeong, S. Kim and C. Weems,.A low power TLB structure for embedded systems.,2002.

[31] Johnson Kin, Munish Gupta and William H.Mangione-Smith, .The Filter Cache: An Energy Efficient Memory Structure., in International Symposium on Microarchitecture, pp. 184.193, 1997.

[32] D. H. Albonesi, .Selective Cache Ways: On-Demand Cache Resource Allocation., in Proceedings of the 32nd International Symposium on Microarchitecture, pp. 248.259, November 1999.

[33] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, .Implementing a cache consistency protocol., in ISCA 85: Proceedings of the 12th annual international symposium on Computer architecture,pp. 276.283, Los Alamitos, CA, USA, 1985,IEEE Computer Society Press.

[34] Mirko Loghi, Martin Letis, Luca Benini and Massimo Poncino, .Exploring the energy efficiency of cache coherence protocols in single-chip multi-processors., in ACM Great Lakes Symposium on VLSI, pp. 276. 281, 2005.

[35] P.S. Magnusson et al., .Simics: A full system simulation platform., IEEE Computer, vol. 35, n. 2, pp. 50.58, February 2002.

[36] Milo M. K. Martin et al., .Multifacet's general execution-driven multiprocessor simulator (GEMS)toolset., SIGARCH Comput. Archit. News, vol. 33,n. 4, pp. 92.99, 2005.

[37] Premkishore Shivakumar and Norman P. Jouppi,.CACTI 3.0: AnIntegrated Cache Timing, Power,and Area Model., Technical Report WRL-2001-2, Hewlett Packard Laboratories, December 28 2001.