MICA: A Memory and Interconnect Simulation Environment for Cache-Based Architectures*

Hung-Chang Hsiao      Chung-Ta King
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
king@cs.nthu.edu.tw

Abstract

MICA is a new-generation simulation environment, which provides complete simulation facilities for simulating distributed shared memory (DSM) multiprocessors. It runs on the inexpensive Linux-based PCs. MICA uses application traces as inputs and provides a core scheduler and memory and interconnect interfaces. A rich set of synchronization algorithms and architecture simulators are also provided. In this paper, we introduce the MICA simulation environment and demonstrate its use in investigating the effectiveness of one-to-many (multicast) communications for write invalidation in DSM multiprocessors.

1. Introduction

With the vast advances of x86-compatible microprocessors, personal computers (PCs) can now perform many tasks which were only possible on high-end workstations just a few years ago. Computer architecture simulation is a typical example. Most tools in use today were developed on Unix workstations. Not until recently can we see tools which are executable on PCs emerge. Examples include Augmint [33] and Limes [31]. These PC-based simulation environments are most appealing for their low cost: The price of a PC is very cheap and is getting even more cheaper. The OS that these environments require are mostly Linux, which can be obtained for free. The simulation tools themselves are developed by academia and are also free. By integrating these components together, a very powerful architecture simulation environment can be constructed easily for one’s disposal. In this paper, we introduce such an environment, which is aimed at studying distributed shared-memory multiprocessors.

Distributed shared memory multiprocessors (DSM) have become very popular in high performance computing. Examples include Dash [9], Alewife [17], SGI Origin 2000 [28], Sequent NUMA-Q [30], Sun S3.mp [35], and HP/Convex Exemplar [3]. In this type of multiprocessors each processing node consists of a processor, a private cache and some local memory. The local memories in the nodes together form a global, shared addressing space. All the processing nodes are connected through a general interconnection network. To reduce the communication overhead in the network, remote memory blocks are often replicated in local caches. However this also creates the cache coherence problem, which is commonly solved on the DSM architecture with a directory-based coherence scheme [11].

There are a variety of simulation environments and simulators for studying DSM architectures. The RSIM system [15] runs on SPARC platforms and simulates processor and memory architectures for DSMs. Its interconnect simulator was modified recently by Bhuyan et al. to allow the study of network architectures [25]. The WWT system [6] is also a DSM simulator, which runs on the CM-5 [10]. It can be used to prototype an experimental DSM on a real machine, but its interconnect simulator is quite primitive and follows a simple approximation model. SimOS [38] is a complete system simulation environment and is based on the SGI IRIX operating system. Although it has been ported to the PC running the Linux, the simulated target is still the MIPS platform. In addition, it is mainly used for the design of processors and memories, but not interconnects. These simulation environments all share two common features: First, they were originally designed for a non-PC platform. Second, their focuses are on the memory hierarchy, and thus the interconnect simulation only plays a secondary role using very primitive models. This has limited researchers on doing serious network research.

As far as we know, there are two DSM simulators, which run on the PC platform and target at the x86 architecture: Augmint [33] and Limes [31]. The Augmint can generate the memory references induced by a group of corporating threads executing a SPMD program. However it does not provide other architectural components, such as caches and the interconnect. Limes, on the other hand, provides the cache and bus modules to investigate bus-based multiprocessors. However it does not provide necessary modules required to simulate a DSM multiprocessor, such as the directory and the interconnection network. As a result, Limes is only suitable for studying small scale and uniform memory access multiprocessors. Furthermore, Limes does not provide any resource scheduling routines, which simplifies the tasks of developing a network simulator.

In this paper, we introduce MICA — a memory and interconnect simulation environment for cache-based architecture [23]. MICA was developed for studying DSM multiprocessors. It runs on a PC and targets at the x86 architecture. It is built on top of CSIM [39], which allows the use of a variety of scheduling policies. Since CSIM provides user-level

*This work was supported in part by National Science Council under Grant NSC 89-2213-E-007-051 and 89-2219-E-007-002.
threads, we can thus simulate a set of computing nodes and communication activities through these threads. More importantly, a rich set of architectural modules (simulators) can then be developed to study the memory and interconnect architectures. Currently, MICA provides cache, bus, memory, directory, node controller/network interface and interconnect modules. With MICA, researchers can now focus on the interested and important portions of their architectures and skip over the parts of less interest.

The remainder of the paper is organized as follows. Section 2 describes the considerations in designing a DSM multiprocessor simulator. In Section 3 we introduce the MICA simulation environment. Section 4 presents a case study using MICA: a DSM simulator for multicast-based invalidation. The simulation results are presented in Section 4. Conclusions of the paper are given in Section 5 together with possible future works.

2. Background and design considerations of a DSM simulator

A DSM multiprocessor consists of several important components: caches, directory and node controller, and interconnection network. A complete DSM simulator should allow flexible simulation of these components in detail. This section gives some background information of these components and discusses the considerations in developing their simulation.

2.1. Cache memory subsystems

The cache memory is a critical component in DSM multiprocessors for shortening the long latency of memory accesses. Usually, a remote memory block can be replicated in a local cache so that the block is retrieved locally next time it is referenced\(^1\). Since a memory block may be cached in multiple nodes, consistency among the caches must be maintained. A cache coherence protocol thus is required. A DSM simulator therefore needs to provide at least one coherence protocol. A flexible simulator should provide more than one protocol and allow the users to choose one from them, perhaps adaptively.

A typical coherence protocol is the Illinois MESI protocol \([16]\). With such a protocol, each cache block is associated with some state bits to denote the block’s state. In MESI, there are four states: modified, exclusive, shared and invalid. The modified state indicates that the cached memory block is the most up-to-date copy and no one else shares such a memory block. The exclusive state means that the block is cached by only one node and the copy is consistent with the memory content. The shared state indicates that multiple nodes share the same memory block, while the invalid state is for the block whose content in the cache is invalid. A memory block is initially in the invalid state. If a node retrieves a memory block and this block is not cached by others, the cache state in that node will be set to exclusive. It is possible for some nodes to access that memory block. If so, the state will be changed to shared. Once one of these nodes updates that memory block, the node owning the most up-to-date memory block changes the state to modified and forces the other sharing nodes to invalidate their cached copies.

2.2. Directory and local memory subsystems

The MESI protocol was originally developed for a bus-based multiprocessor, where the bus provides a medium to broadcast coherence transactions. On scalable DSM multiprocessors which consist of hundreds or thousands of computing nodes, it will be very inefficient to provide such a broadcast medium. Typical cache coherence protocols on DSM multiprocessors thus follow a directory-based approach \([2]\).

A directory-based protocol associates a directory entry with each memory block to indicate who share such a block. Such information is often stored in a structure called the presence vector or directory pointers for that block. State bits are also used to track a block’s status. The node maintaining the directory entry is called the home node of the memory block.

On a read miss, the missing node sends a message to request the block from the home node. The home node replies the block if it is shared or uncached by other nodes. The directory entry in the home node is updated accordingly. If the requested block is dirty in another node’s local cache, the home node will ask the latter to forward the most up-to-date copy to the requesting node. On a write miss, the operations generally follow those for a read miss. One complication is that if the block is shared by several nodes, then the home node needs to invalidate those sharing nodes first before it allows the missing node to write into the block.

On DSM multiprocessors, the latency of memory requests heavily depends on the design of the cache, page allocation policy, directory, and local memory subsystems. For caches, the cache associativity, block size and capacity are the three main factors to consider during their design. They collectively affect various misses in the caches: conflict, cold, capacity, and coherence misses. These misses contribute to the memory latency. The page allocation policy determines the location of the home node for a given memory block \([5]\), which in turn determines the distance between the requesting node and the home node. A good page allocation policy should strive to allocate the pages in such a way that the remote memory access overhead and network traffic can be minimized. The directory structure in the DSM multiprocessor also affects greatly the performance of the system. Different directory protocols and directory structures, such as the linked list \([24]\) and the limited pointers \([17]\), introduce different coherence behavior and result in different memory latency.

With the huge design space resulted from the above design options, a DSM simulator should incorporate these options as many as possible and allows very flexible investigation of the effects of their combinations. This entails providing in the simulator various modules, such as cache modules, directory modules, page allocation modules and interconnect modules.

2.3. Interconnect subsystems

The interconnect subsystem determines how fast the nodes in a DSM multiprocessor can exchange their data and coordinate their operations. The design space of an interconnect includes the router/switch architecture, the switching techniques and the routing algorithms \([14]\). These design parameters significantly affect the communication efficiency of the system, which is often measured in terms of the communication latency and bandwidth. These parameters should be modeled accurately and incorporated into an interconnect simulator.

---

\(^1\)Some memory blocks are not cached, e.g., those blocks storing the synchronization variables for, say, the test-and-set \([40]\) primitive.
2.3.1. Router/Switch architectures

The router/switch architecture can be generalized, and Figure 1 depicts such a model. It consists of flit buffers, virtual channels, a crossbar switch, and routing/control and physical links. Most router/switch architectures in DSM multiprocessors can be modeled and simulated by changing the configuration of this general model.

![Figure 1. A general router/switch architecture](image)

The flit buffers are FIFO buffers for storing message flits during transmission. Alternative buffering architectures are input buffering and output buffering [14]. Input buffering associates the FIFO buffers with physical input links, while output buffering associates the flit buffers with output links.

Virtual channels [13] are used to multiplex message flits onto a physical link. It boosts the router’s transmission bandwidth and makes better use of the physical links, although the switching delay may be increased. Virtual channels are also used to avoid deadlock in the system [14] by providing sufficient flit buffers to break the hold-and-wait condition. The additional routing lanes supported by virtual channels also allows messages to be routed adaptively [22].

The crossbar switch is responsible for connecting the input buffers to the output buffers, under the control of the routing/control unit. The connection is according to the message header in the input buffer. After the connection is established, the message flits stored in the input buffer can be routed to the output buffer without delay. If multiple messages contend for the same output buffer, an arbitration will take place. Typical arbitration schemes include the round-robin policy and the FCFS policy [20].

2.3.2. Switching techniques

Switching methods generally fall into three categories: circuit switching, virtual cut-through, and wormhole switching [14]. Since the coherence messages are quite short in a DSM multiprocessor, the latter two approaches are more suitable, because they do not have to establish a priori a path between the sender and the receiver. On the other hand, if a dedicated network is used for synchronization traffic, the circuit switching mechanism may have an edge [41].

To model virtual cut-through in the router/switch model shown in Figure 1, we need only enlarge the input/output buffers to accommodate a complete coherence message, when the requested output buffer is busy. This is very practical because coherence messages in a DSM multiprocessor are quite short and can be buffered entirely in the input/output buffer. Modeling wormhole switching is even simpler: we only have to provide a one-flit buffer per virtual channel in the input/output buffer.

2.3.3. Routing algorithms

According to the paths that a message may be routed between a given pair of source and destination nodes, routing algorithms can be categorized into deterministic, partially adaptive, and fully adaptive [14]. A deterministic routing algorithm always supplies the same path for a given source and destination nodes. Partially and fully adaptive routing provide more than one path, with the latter imposes almost no restriction.

According to the length of the routing path, routing algorithms can be classified as minimal or non-minimal. A non-minimal routing algorithm allows the message to be routed away from its destination. Thus mechanisms must be designed to avoid the message being routed in the network forever.

Most routers or switches are designed to make their routing decision locally with without global network information. Two possible implementations are to look at a table (table-lookup) or to consult a finite state machine. To aid their routing decisions, many systems employ source routing, in which the routing path is determined in the source node before the message is injected into the network.

2.3.4. Collective communication

Communications in the network can be categorized according to the parties participating in the communication. If there are more than one source or destination nodes, the communication is called collective. For DSM multiprocessors, there are one-to-many (multicast) and many-to-one collective communications. For example, multicasts can be applied to the invalidation traffic [12] and the many-to-one communication can be employed to support barrier synchronization [37, 41].

It is possible to use pure software [29] to do collective communication. This is done by employing multiple one-to-one messages to form a tree-like structure to route the messages. It is not necessary to change the interconnect infrastructure [43]. Hardware-based collective communication schemes can be categorized into path-based and tree-based [29, 32]. They all require that the router/switch be able to replicate a message from one input buffer to multiple output buffers. The replication is based on the multiple destination addresses carried in the message header. With this hardware capability, the sender only has to send one multistination message to reach all the destinations [36]. To avoid possible deadlocks in cut-through networks, the path-based multicast links the destinations in one or a few long paths so that no cycle can be formed in the network. In the tree-based scheme, the path length is shorter but deadlocks are hard to avoid. Thus it is often used with switching techniques such as store-and-forward, virtual cut-through, or some variations of them.

3. The MICA simulation environment

In this section, the MICA simulation environment and the associated simulators are presented. The simulation environment is shown in Figure 2. It is built on top of CSIM [39]. Its input is the application trace generated by a modified version of Limes [31]. Major components in MICA include the input traces, the core scheduler, and the computing and communication threads. A computing thread represents a computing node in the simulated DSM multiprocessor. Also provided are a synchronization library and a rich set of architectural simulators. The simulators can be invoked by the memory and interconnect interfaces.
3.1. Application traces

The trace of the given application is collected by using a modified version of the Limes memory reference generator. Since multiple threads will be created to perform a single task for an application, each trace record should include the thread id, the access address, the type of the memory request, the segment of the accessed data, the size of the accessed data, and the pseudo time of the thread under the PRAM model [19]. Detail information of each field is given below.

- **Thread id:** Each trace record contains a thread id, indicating which thread induces such a memory request.
- **Access address:** This field denotes the address to which the memory request is accessing.
- **Type of memory request:** Memory requests in MICA can be READ, WRITE, LOCK, UNLOCK, BARRIER and NOTIFICATION. Among them, LOCK and UNLOCK are used to perform locking and unlocking on a synchronization variable. BARRIER is used to synchronize all running threads, and NOTIFICATION is used by the last thread committing a barrier operation.
- **Segment of accessed data:** The accessed data block may be in the CODE_SEG, DATA_SEG or STACK_SEG segment. CODE_SEG indicates that the accessed data is in the code segment. Some self-modifying code may induce such a request. If the accessed memory block is in DATA_SEG, the block could be shared by allocating it with G_MALLOC() for a piece of the global shared memory. If it is in STACK_SEG, the data should be local variables.
- **Size of accessed data:** The size of the accessed data may be BYTE, INT or FLOAT.
- **Pseudo time stamp:** The PRAM model assumes that each memory request takes one cycle to complete. In MICA, each trace record is associated with a time stamp to indicate the time of a memory request under the PRAM model. All non-memory requests are assumed to take one processor cycle. The time stamp has taken the time to perform such non-memory instructions into consideration. Also, the idle time to perform LOCK, UNLOCK and BARRIER are all accumulated into the time stamp.

3.2. The core scheduler

The core scheduler is the main core of MICA. It is responsible for dispatching memory requests to the computing threads. However, primitive lock and unlock synchronization algorithms are provided in MICA. They are accessible for all threads and are defined in the memory/interconnect interfaces. Users can easily develop their architectural modules, e.g., the cache module, without significantly modifying the simulator programs.

For each computing thread that is in the idle state, the core scheduler dispatches the memory request whose trace record contains the matching thread id field and time stamp is the smallest to that thread. A computing thread has its own clock, and the clocks of the threads can be synchronized through the BARRIER operation. Note that the time stamp in the system is increased by performing events. The events can be a read, a write, a lock, or a barrier. In this way, the simulation is driven by events rather than system clocks, which in turn speeds up the simulation while preserving accuracy.

3.3. Computing and communication threads

There are two kinds of threads in MICA. One is the computing thread and the other is the communication thread. The computing threads, as mentioned above, are used to represent all computing nodes in the simulated DSM multiprocessor. It consists of the cache module, the directory module, the page allocation module and the node controller module. It receives memory requests from the scheduler and simulates their operations by invoking corresponding modules. The interfaces to invoke these modules are defined in the memory/interconnect interfaces. It is thus possible to redesign these modules individually without changing the interfaces. If a module has an empty architectural behavior code, the computing thread will behave as if running under the PRAM model. All computing threads deliver their communication requests by invoking the node controller modules, which then invokes the interconnect interfaces. The communication threads are used to simulate message transmissions in the interconnect. A communication thread denotes a message worm. It can acquire and release interconnect resources, such as flit buffers, virtual channels (external communication links), ejection/injection virtual channels (internal communication links), and a computing thread. Synchronization among computing threads and communication threads are through SEND and RECEIVE primitives. For example, a computing thread may wait for a communication thread to transmit/receive coherence messages to/from the home node. They use the RECEIVE and SEND primitives respectively to achieve the synchronization.

3.4. Memory and interconnect interfaces

MICA provides a set of memory and interconnect interfaces that can be invoked by the computing threads. Memory interfaces include MEM_READ(), MEM_WRITE(), MEM_LOCK(), MEM_UNLOCK(), and MEM_BARRIER(). Interconnect interfaces include SEND_MESSAGE(), RECEIVE_MESSAGE(), MESSAGE_WORM(), RELEASE_INPORT(), RELEASE_OUTPORT(), RELEASE_LINK(), ROUTE(). Through these interfaces, users can easily develop their architectural modules, e.g., the cache module, the interconnect module, etc., without significantly modifying the simulator programs.

3.5. The synchronization library

A variety of synchronization algorithms are provided in MICA. Primitive lock and unlock synchronization algorithms include test-and-set, test-test-and-set [40], and queue-based locking [26]. Barrier synchronization in
MICA can be software-based or hardware-based. Software-based barrier synchronizations are built on top of lock and unlock, and they include the counter-based barrier and tree combining barrier [40]. Hardware-based barriers include the hardware-based tree-combing [41] and the message-passing-based schemes, e.g., the one using multideestination worms [37].

3.6. Architectural modules

All modules in MICA are classified as memory-related and interconnect-related. Memory-related modules include the cache modules, the directory modules, the page allocation modules, and the node controller modules. The cache module models write-back or write-through caches with parameters indicating their associativity, block size and capacity. The directory module supports the full-bit-vector and the limited directory schemes [2]. There are two page allocation policies in MICA: first-touch and round-robin [5]. The node controller could be purely hardware-based [9] or thread-based [34].

Interconnect-related modules include the flit buffers, virtual channels, switching schemes, point-to-point routing algorithms and collective routing algorithms. The flit buffers are implemented by using the CSIM facility. One buffer is emulated by one facility, and it is abstracted as FLIT BUFFER, which serves as the basic data structure for the interconnect in MICA.

The virtual channels are implemented on top of FLIT BUFFER, which are denoted as VIRTUAL_LANE. Multiplexing on multiple virtual channels can be done by defining their scheduling policy. Currently, MICA provides first-come-first-serve (FCFS), priority, and round-robin (RR) scheduling policies. As for switching techniques, MICA supports virtual cut-through and wormhole switching, which are the switching schemes used in the state-of-the-art DSM multiprocessors.

The routing algorithms in MICA heavily depend on the users’ requirements and the network topology. MICA currently implements k-ary n-cube networks and dimension-order routing. Since it also supports source routing, irregular networks can be simulated as well. Communications between nodes in MICA are through point-to-point (unicast) messages by default. MICA also supports collective communications. For invalidation traffic, it is possible to use path-based and tree-based multicasts in MICA. For the reduction phase of barrier synchronization, many-to-one collective communication can be used.

Within a computing node, MICA provides a bus interconnect, which is used to communicate between the local memory, the cache, the processor and the node controller.

4. A case study

In this section, we present an example of using MICA to study the invalidation traffic in a DSM multiprocessor. In contrast to the traditional designs, which used point-to-point invalidation messages (unicast-based invalidation scheme), we study the effects of using multicasts to implement invalidation. With the help of MICA, we evaluated the effectiveness of two multicast algorithms: dual-path [29] and tree-based [32]. The evaluated applications and the architectural models are described in Section 4.1.2. The simulation results are presented in Section 4.2.

4.1. Applications and architectural models

4.1.1. Applications

Six representative benchmark programs were chosen in the evaluation. Ocean, Radix and LU are from the SPLASH-2 suites [7], Gauss and SOR are from Presto [42], and Spark is from [4]. The input data sets and a brief description of the programs are given in Table 1. Due to limitations of the available computation resources and simulation time, small input data sets were chosen for these applications. This results in small working sets, as shown in Table 2. All benchmark programs were compiled with the GNU C compiler version 2.7 on the Linux operating system with an –O2 optimization level.

Table 1. Benchmark programs used in the evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Input data set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss</td>
<td>160x 35 matrix</td>
<td>Gauss elimination</td>
</tr>
<tr>
<td>LU</td>
<td>262144 keys, 512 radix</td>
<td>Factorization of a dense matrix into the product of a lower triangular and an upper triangular matrix</td>
</tr>
<tr>
<td>Ocean</td>
<td>66 grid, 1e-07 error tolerance</td>
<td>Simulation of large scale ocean movements based on eddy and boundary currents</td>
</tr>
<tr>
<td>Radix</td>
<td>128x128 keys, 512 radix</td>
<td>Implementations of Successive over-relaxation algorithm with an initialization boundary value of 1</td>
</tr>
<tr>
<td>SOR</td>
<td>128x 128 matrix, 1% tolerance</td>
<td>Implementation of Successive over-relaxation algorithm with an initialization boundary value of 1</td>
</tr>
<tr>
<td>Spark</td>
<td>Tiny, 2.5 pack, 20 iterations</td>
<td>Three-dimensional unstructured finite element simulation of carbonate ground motion</td>
</tr>
</tbody>
</table>

Table 2. Working sets in the evaluated applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Gauss</th>
<th>LU</th>
<th>Ocean</th>
<th>Radix</th>
<th>SOR</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working set (KB)</td>
<td>96</td>
<td>512</td>
<td>32</td>
<td>96</td>
<td>96</td>
<td>32</td>
</tr>
</tbody>
</table>

Characteristics of these benchmark programs are further analyzed below. All the statistics were collected from MICA. In Table 3, the bandwidth requirements of the evaluated applications are shown. It indicates that Gauss, Ocean, SOR and Spark are communication-intensive applications. Radix has a medium communication requirement and LU and Spark are computation-intensive applications. These data conform to the observations made by Abandah et al. [1]. In Figure 3, a histogram of number of messages vs. execution time of the evaluated applications are shown. It shows that LU and Radix induce bursty communications during their execution. Bursty communications may impact system performance even for computation-intensive applications such as LU. Figure 3 also shows the amount of invalidation messages transmitted along time. We can see that Gauss, Ocean, SOR and Spark have quite a few invalidation messages. Although for LU and Radix, only a few invalidation messages are transmitted, they appear in periods of bursty communications and thus still have some impacts on system performance.

In Figure 4, the invalidation degrees for the evaluated benchmarks are shown. The invalidation degree indicates the number of nodes which need to be invalidated during an invalidation. All the evaluated benchmarks have a low degree of invalidation, especially for LU and Radix. For Spark, the invalidation degree is quite high, and it is expected to be improved by multicast invalidation, although Spark is a computation-
intensive application and does not generate any burst communication.

Figure 4. The invalidation patterns

4.1.2. Architectural models
We model a conventional DSM multiprocessor. There are 32 nodes in the system, connected through a 4 × 8 mesh network. The processing nodes consist of a processor, a L2 cache, some memory and a node controller/network interface. The processor is clocked at 200 MHz to attend a reasonable computation/communication ratio for the evaluated applications under the given input data sets.

Access time to the L2 cache takes 1 cycle. The cache is write-dominated and copy-back. Each cache block has a size of 32 bytes. The L2 cache is 64K bytes in size and uses two-way set-associativity. The cache size was scaled to 64K bytes to match the small working sets in the evaluated applications (see Table 2). Such a size is sufficient to capture all the shared memory blocks. In choosing the cache size we follow the methodology in Chapter 4 of [11], which scales the caches according to the workload.

The bus arbitration time is assumed to take 2 cycles and the bus width is 64 bits. In general, to transfer a cache block, the acquired data word is transferred first and thus needs to join the bus arbitration. The remaining words in the cache block are transferred next without any bus arbitration. Cache coherence in the system is maintained with a directory-based protocol, similar to that of the Dash [9] multiprocessor. The directory employs the full-bit-vector scheme, and the directory entries can be cached in the directory cache of the node controller. Access time to the directory is 2 processor cycles. In addition, access time to the DRAM memory takes 20 processor cycles.

The modeled multiprocessor supports sequential consistency [21]. The default page allocation scheme is round-robin and the page size is set to 512 bytes [27], again due to the small problem size used in the evaluation. Synchronization is based on the test-test-and-set primitive, which is used to implement the lock/unlock operations [26]. The synchronization variables can be cached for local spinning. Barrier synchronization is counter-based with exponential backoff [40].

The node controller/network interface is modeled at 100 MHz. The node controller overhead is estimated to about 12 protocol processor cycles, without considering the time to access off-chip interface registers (e.g., bus and memory interface registers) and the time to access DRAM memory. Also, we estimate the time for sending and receiving a message to be 9 protocol processor cycles. These values were set according to those reported in Flash [18].

The network was simulated to the flit-level, with a flit size of 16 bits. It uses the wormhole switching [14], which is similar to that in Dash. Routing through the 2D mesh network is based on the X-Y routing. The routing delay is set to 4 cycles, following the SGI Spider [20] and the Intel Cavallino [8]. Each link is 16-bit wide and takes 1 cycle delay. The switching delay is also 1 cycle. There are four virtual channels per each physical link, which is the same as that used in the Spider and Cavallino. Channels belonging to the same physical link should be multiplexed onto this physical link. Flit buffering follows input buffering and link arbitration is FCFS. The size of a control message is 4 flits, while that of a data message is 20 flits, because it has to carry a memory block. The core of the router is assumed to run at 100 MHz. Finally in our experiments, we did not apply multicasts and invalidations to synchronization, because they can be implemented with better solutions, such as queue-based locking schemes [26].

4.1.3. Performance metrics
Our evaluation of the invalidation schemes uses the following performance metrics:

- **Invalidation time:** Invalidation time is the time to perform the invalidations and to collect the acknowledgments for a write operation.

- **System traffic:** System traffic is the total amount of messages transferred in the interconnection network. It includes the messages for processing memory read, write and synchronization operations.

- **Write stall time, read stall time, and synchronization stall time:** These are the time to perform write, read, and synchronization (including lock/unlock and barrier) operations, respectively.
• **Total execution time:** This is the total time required to execute the given application. This metric is the ultimate index of the effectiveness of multicast-based invalidation.

### 4.2. Simulation results

In this section, the simulation results are presented. Data of different invalidation schemes reported here will be normalized respect to those of the unicast-based invalidation.

**Table 4. Invalidation time, invalidation traffic, write stall time, read stall time, synchronization stall time, and total execution time of the baseline configuration**

<table>
<thead>
<tr>
<th>Applications</th>
<th>Gauss</th>
<th>LU</th>
<th>Ocean</th>
<th>Radix</th>
<th>SOR</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invalidation time</strong></td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
</tr>
<tr>
<td>Unicast</td>
<td>[70]</td>
<td>[74]</td>
<td>[74]</td>
<td>[81]</td>
<td>[89]</td>
<td>[74]</td>
</tr>
<tr>
<td>Dual-path</td>
<td>[70]</td>
<td>[74]</td>
<td>[81]</td>
<td>[89]</td>
<td>[74]</td>
<td>[74]</td>
</tr>
<tr>
<td>Pruning</td>
<td>[70]</td>
<td>[74]</td>
<td>[81]</td>
<td>[89]</td>
<td>[74]</td>
<td>[74]</td>
</tr>
<tr>
<td><strong>Write stall time</strong></td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
</tr>
<tr>
<td>Unicast</td>
<td>[81]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[86]</td>
</tr>
<tr>
<td>Dual-path</td>
<td>[81]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[86]</td>
</tr>
<tr>
<td>Pruning</td>
<td>[81]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[91]</td>
<td>[86]</td>
</tr>
<tr>
<td><strong>Read stall time</strong></td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
</tr>
<tr>
<td>Unicast</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
</tr>
<tr>
<td>Dual-path</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
</tr>
<tr>
<td>Pruning</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
<td>[99]</td>
</tr>
<tr>
<td><strong>Synchronization stall time</strong></td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
</tr>
<tr>
<td>Unicast</td>
<td>[66]</td>
<td>[96]</td>
<td>[76]</td>
<td>[100]</td>
<td>[69]</td>
<td>[72]</td>
</tr>
<tr>
<td>Dual-path</td>
<td>[66]</td>
<td>[96]</td>
<td>[76]</td>
<td>[100]</td>
<td>[69]</td>
<td>[72]</td>
</tr>
<tr>
<td>Pruning</td>
<td>[66]</td>
<td>[96]</td>
<td>[76]</td>
<td>[100]</td>
<td>[69]</td>
<td>[72]</td>
</tr>
<tr>
<td><strong>Overall execution time</strong></td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
<td>[100]</td>
</tr>
<tr>
<td>Unicast</td>
<td>[87]</td>
<td>[90]</td>
<td>[95]</td>
<td>[100]</td>
<td>[90]</td>
<td>[95]</td>
</tr>
<tr>
<td>Dual-path</td>
<td>[87]</td>
<td>[90]</td>
<td>[95]</td>
<td>[100]</td>
<td>[90]</td>
<td>[95]</td>
</tr>
</tbody>
</table>

Performance of the invalidation schemes on the baseline multiprocessor is summarized in Table 4. From the table we can see that multicasts improve the invalidation time by 11% (in Radix) to 41% (in SOR). For the communication-intensive applications, e.g., Gauss and SOR, the improvement due to multicast-based invalidation is very significant — up to 30% and 41% improvement for Gauss and SOR, respectively. For Spark, it is a computation-oriented application but has a high invalidation degree. Thus, multicast-based invalidation also improves its performance.

It is interesting to see that, although LU and Radix have a low invalidation degree, multicasts can still improve the invalidation time by 26% and 11% respectively. This can be attributed to the bursty communications in these two programs, which stress the interconnect. With multicast-based invalidation, fewer network resources are occupied and the stress can be relieved. To see the effects on bursty communications more closely, we plot the time for an invalidation during the execution of LU and Radix in Figure 5. We can see that the average invalidation time per invalidation message for LU jumps to about 25000 processor cycles under unicast-based invalidation, when the program runs for 3800000 cycles. Note that LU causes a bursty communication at that time (see Figure 3). The invalidation time can be reduced to about 10000 cycles using dual-path and pruning at that instance. These data confirm that multicast-based invalidation can reduce the communication latency significantly during bursty communications. Radix shows a similar behavior, but the effects are less obvious because of the relatively smaller amount of invalidation messages induced.

**Figure 5. The time for an invalidation during the execution of LU and Radix**

We also measured the network traffic induced by invalidation and the results are depicted in Figure 6. The figure shows that using multicast-based invalidation schemes can reduce about 7% (in Ocean) to 30% (in SOR) the control messages in invalidation. For SOR, the invalidation traffic can be reduced up to 30% by using pruning. This helps to explain why multicasts for invalidation have a large impact on the invalidation time in SOR. Note that, according to our baseline configuration, most shared memory blocks can be held in the local caches and traffic to the network will be dominated by invalidations. For SOR, invalidations are to the boundary data of the nodes. Thus improving invalidations with multicast messages in SOR will improve the system traffic even with a low invalidation degree. For LU and Radix, it is not surprising that multicast-based invalidation reduces the system traffic modestly due to their low invalidation degrees.

**Figure 6. System traffic in the baseline configuration**

Now, let us turn to the effects of multicast-based invalidation on write and read operations. The write stall time reported in Table 4 includes the invalidation time as well as the time for actual write. From the table we can see that the write stall time can be improved by using multicast-based invalidation schemes due to a reduction in the invalidation time and traffic. The improvement is from 4% (in LU) to 14% (in Spark), excluding Radix. The table also shows that the time for read and synchronization can also be improved. This is because using multicasts for invalidation reduces network load, which in turn smooths the transmissions of read and synchronization messages. Overall, the execution time of the benchmark programs is improved.
using multicast-based invalidation. The improvement is from 5% (in Ocean) to 18% (in Gauss), excluding LU and Radix.

Comparing the two multicast methods, we can see from Table 4 and Figure 6 that dual-path always outperforms pruning, except in network traffic. Our explanation is that dual-path only generates one or two worms and thus has only one or two message headers for routing and reserving resources. When the header reserves necessary resources in a router, the remaining flits, including other destination addresses and the invalidation information, can follow. On the other hand, pruning will branch out several heads, each needs to be routed and will experience routing delays.

Although under our experimental settings, dual-path performs better than pruning, this is by no means conclusive. We have only demonstrated a very small design space of DSM multiprocessors by using MICA. To compare with the results reported by Malumbres et al. [32], our results were obtained from a smaller network (a 4×8 mesh network) as opposed to an 8×8 mesh network there. In our small network, the effects of path length is not very significant, which then favors dual-path. Other differences between our experiments and those in [32] include different assumptions in routing delay (1 cycle there versus our 4 cycles), message length, and number of sharers to be invalidated. In [32], synthetic workloads were used assuming an invalidation degree of 4~25.

5. Concluding Remarks

In this paper, we have introduced the MICA simulation environment and demonstrated how it can be used to investigate the effectiveness of multicast communications for write invalidation in DSM multiprocessors. The MICA simulation environment is built on top of CSIM and running on a Linux PC. The input of the application traces are generated from a modified version of Limes. The MICA simulation environment includes the application traces as its input, the core scheduler, and the threads simulating the computing nodes and the communication messages. The simulator modules provided by MICA can be invoked through the memory and the interconnect interfaces. They include the cache, bus, directory, node controller, and interconnect modules. To support the shared programming paradigm, a rich set of synchronization algorithms is supported.

MICA represents a new set of architecture simulation tools, which run on low-cost PCs and simulate a complete multiprocessors, including its memory hierarchy and interconnect. By using MICA, researchers can reuse and reconfigure the simulators already developed and focus on the parts they are interested the most, without significantly rewriting other architectural modules. This reduces significantly the efforts in developing architecture simulators.

Since MICA does not provide processor models for the node processor simulated, we cannot study the effects of using different node processors. We plan to support this feature in the future, especially for the x86-compatible processor model. In addition, we will also add support for irregular networks and related routing algorithms, especially for adaptive routing.

References


