IT Master theses
UPTEC F 00 093

DSZOOM – Low Latency Software–Based Shared Memory

ZORAN RADOVIĆ

Supervisor: Professor Erik Hagersten

Sponsors: Sun Microsystems, PSCI

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

DSZOOM – Low Latency Software–Based Shared Memory

BY

ZORAN RADOVIĆ

December 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Master of Science in Technology
at Uppsala University 2000

DSZOOM – Low Latency Software–Based Shared Memory

Zoran Radović
`zoranr@it.uu.se`

Department of Computer Systems
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

`http://www.it.uu.se/`

ISSN 1401-5757
Printed by the Department of Information Technology, Uppsala University, Sweden

ABSTRACT

Software-implementations of shared memory are still far behind the performance of hardware-based shared memory and are not viable options for most fine-grain shared-memory applications. The major source for their inefficiency comes from the cost of interrupt-based asynchronous protocol processing, not from the actual network latency. As the raw hardware latency of inter-node communication decreases, the asynchronous overhead in the communication becomes more dominant. Elaborate schemes, involving dedicated hardware and/or dedicated protocol processors, have been suggested to cut the overhead.

This thesis describes how all the asynchronous overhead can be completely removed by running the entire coherence protocol in the requesting processor. This not only removes the asynchronous overhead, but also makes use of a processor that otherwise most likely would be stalled. The technique is applicable to both page-based and fine-grain software shared memory systems.

The proof-of-concept implementation presented in this thesis—DSZOOM—is a fine-grain software-based shared memory. It demonstrates a protocol-handling overhead below a microsecond for all the actions involved in a remote load operation, to be compared to the fastest implementation to date of around ten microseconds. The all-software protocol is implemented assuming only some basic low-level primitives in the cluster interconnect. The implementation is thread-safe and allows all processors in a node to simultaneously perform remote operations. Based on a remote atomic and simple remote put/get operations the requesting processor can take the role of the directory agent, traditionally assumed by a remote protocol agent in the home node in other implementations.

ACKNOWLEDGEMENTS

Many people have been extremely helpful and supportive during the development time of this thesis.

First of all, I would like to thank professor Erik Hagersten for introducing me to this fascinating world of high performance computing and computer architecture; and, of course, for giving me this great opportunity to be one of the coworkers in the newly formed Uppsala Architecture Research Team (UART) at the Department of Computer Systems, Uppsala University, Sweden. Erik has been enormous inspiration and support.

I would also like to thank Glenn Ammons, Computer Sciences Department, University of Wisconsin, for his excellent support, encouragement, and quick EEL updates during the whole DSZOOM implementation time. Many thanks goes to Sverker Holmgren and Henrik Löf from the Department of Scientific Computing, Uppsala University, for allowing/helping me to run the CPU-time-consuming benchmarks on their Sun Enterprise E6000™ SMP server.[1]

Finally, I would like to thank my family for their endless patience and understanding.

Zoran Radović
Uppsala, december 2000

[1] Well, in fact, it is a Sun-WildFire prototype SMP cluster consisting of two unmodified Sun Enterprise E6000™ machines running a slightly modified version of Solaris 2.6 operating system. Erik's design… [HK99]

CONTENTS

1 Introduction

Clusters of symmetric multiprocessors (SMPs) usually provide a powerful platform for executing parallel applications. To allow for shared-memory applications to run on such clusters, software distributed shared memory (SDSM) systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to hardware shared memory systems for executing certain classes of workloads (commercial and/or scientific). SDSM technology can also be used to connect several large hardware-coherent DSM systems and thereby extend their upper scalability limit.

Most SDSM systems keep coherence between page-sized coherence units [L88, LH89]. The normal per-page access privilege of the memory-management unit offers a cheap access control mechanism for these SDSMs. The large page-size coherence units created extra false sharing in the earlier SDSM systems and caused frequent transfer of large pages between the nodes. In order to avoid most of the false sharing, weaker memory models have been used to allow many update actions to be lumped to a specific point in time, such as the lazy release consistency (LRC) protocol [K95].

Fine-grain SDSMs with a more traditional cache-line-sized coherence unit have also been implemented. Here, the access control check is either done by altering of the error check code (ECC) [SFH+96] or by inline code snippets [SGT96]. The small cache line size reduces the false sharing for these systems, but the explicit access-control check adds extra latency for each remote load or store operation to global data. The most efficient access check reported to date is three extra instructions adding three extra cycles for each load to global data [SFH+98].

Today's implementations of SDSM systems suffer from quite long remote latencies. Thus, their scalability has never reached acceptable levels for general SMP shared-memory applications. This is especially important for the fine-grain SDSM systems. The coherence protocol is often implemented as communicating agents running in the different nodes sending requests and replies to each other, as illustrated in Figure 1. Each agent is responsible for accessing its local memory and for keeping a directory structure for "its part" of the shared address space. The agent where the directory structure for a specific coherence unit resides is called its home node. The interrupt cost for asynchronous protocol processing is the single largest component of the slow remote latency, not the actual wire delay in the network or the software actually implementing the protocol. To our knowledge, the shortest SDSM read latency to date is that of Shasta [SGA97]. The 15-microsecond roundtrip read latency is roughly divided into 5 microseconds, of "real" communication and 10 microseconds of interrupt and agent overhead [G00]. Most other SDSM implementations have substantially larger interrupt overheads, and latencies closer to 100 microseconds have been reported.

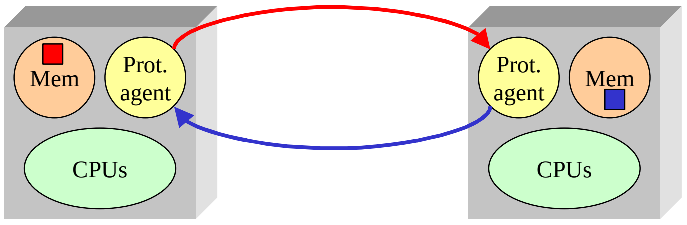


Figure 1: Protocol agent communication in traditional SDSM systems. The interrupt cost for asynchronous protocol processing is the single largest component of the slow remote latency.

This thesis suggests a new efficient approach to implement a global coherence protocol. While other work has proposed elaborate schemes for cutting down on the overhead associated with interrupting and/or polling caused by the asynchronous communication between the agents, e.g., [MFH+96, BLS99], DSZOOM has completely eliminated the protocol-agent interactions. Instead, the entire coherence protocol is implemented in the protocol handler running in the requesting processor, as shown in Figure 2. Rather than relying on a "directory agent," located in the home node, as the synchronization point for the coherence of a cache line, we use a remote atomic fetch-and-set operation to allow for protocol handlers running in any node, not just the home node, to temporarily acquire atomic access to the directory structure of the cache line. We believe that the solution presented here would be beneficial both for page-sized and fine-grain SDSMs, even though we will concentrate on fine-grain SDSMs in this thesis.

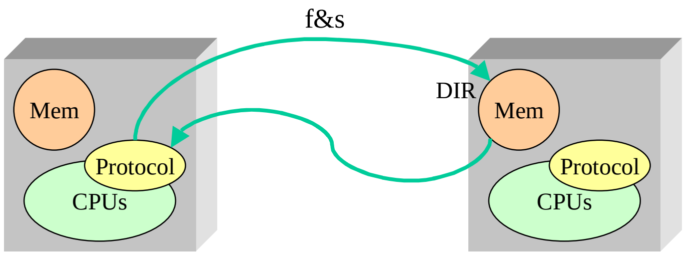


Figure 2: Our proposal. Coherence protocol is implemented in the protocol handler running in the requesting processor.

We have implemented the DSZOOM system—a proof-of-concept implementation—inside one Sun Enterprise E6000™ SMP server. We use an unmodified version of EEL [LS95] to instrument the benchmark binaries used in our study and to implement fine-grain SDSM between "virtual nodes," modeled as processes inside one SMP. SPARC's non-cacheable Block Load and Block Store instructions are used to model the remote put/get instructions between the different address spaces of a network. We have measured the actual protocol overhead to be less than a microsecond for a remote load. Latency loops have been inserted into our protocol in order to model the latency of realistic networks, such as the SCI-CLUSTER (described later in the thesis) or the emerging InfiniBand™ standard. We have also modeled latencies of more traditional interrupt-driven SDSM implementations. A total of eight completely unmodified SPLASH-2 programs [WOT$^+$95] that have been developed for hardware SMP multiprocessors using well-known PARMACS macros are studied.

The remainder of this thesis is organized as follows. Section 2 presents the basic idea and a couple of assumptions about the cluster interconnect for this thesis. The proof-of-concept implementation is described in Section 3 together with a short introduction to the SMP/PARMACS programming model. Section 4 presents the results of our performance study of the DSZOOM system and the description of the experimental environment. Related work is shortly discussed in Section 5. Section 6 presents the conclusions, and finally, in Section 7, the future work is presented.

2 Basic Idea

This section will give a brief introduction to the basic idea for this thesis, and a couple of assumptions about cluster interconnect used later on in the Proof-of-Concept Implementation section.

2.1 Cluster Networks with Put/Get Semantics

Put/Get cluster semantics is one of the simpler hardware solutions to cluster networking. One of the earliest implementation of put/get clustering architectures is the Scalable Coherence Interface (SCI) implementation by Dolphin.[2] This usage of a subset of the SCI standard protocol is referred to as *SCI-CLUSTER* in this thesis. Unlike hardware coherent interfaces, which need to be connected to the memory bus of a node, the SCI-CLUSTER is connected to an I/O bus, e.g., PCI or SBUS.

SCI-CLUSTER is widely used as the high-performance cluster interconnect by Sun Microsystems for large commercial and technical systems. A flavor of SCI-CLUSTER, *SCX Channel* [S95], is also used as CRAY's I/O-interconnect.

SCI-CLUSTER supports communication between cluster-nodes through put-and-get operations directly into remote nodes' memory, typically transferring a 64-byte data unit. The different cluster nodes run different kernel instances and do not share memory with each other in a coherent way; in other words, no invalidation messages are sent between the nodes to maintain coherence when replicated data are altered in one node. This removes the needs for the complicated coherence scheme of SCI, built on double-linked lists of sharing pointers and allows it to be connected to the I/O bus rather than the memory bus.

A cluster node can map remote memory into its I/O space. The remote memory can be accessed using ordinary load-and-store operations. In order to prevent a "wild node" from destroying crucial parts of other node's memories, the incoming transactions are sent through a network MMU (IOMMU). Each kernel needs to set up an appropriate IOMMU mapping to the remotely accessible part of its memory before the other nodes may access its memory. Given the correct initialization of the IOMMU, user-level accesses to remote memory with cache-line granularity have been enabled.

Typically, cluster communication is done through message-passing semantics implemented using a collection of put/get primitives to remote memory. This has allowed for low-latency implementations of MPI between workstations or servers.

[2] SCI is better known for its implementation of coherent shared memory than its non-coherent internode communication. In this thesis we only refer to its usage as a cluster interconnect.

A ping-pong round-trip latency of 5 microseconds, including MPI protocol overhead, has been demonstrated on a SCI network with a 2 microsecond raw read latency.

DSZOOM assumes a network with put/get semantics, similar to that of SCI-CLUSTER; accessible from user code with a round-trip read latency of about 2 microseconds.[3] The remote load-and-store operation are triggered by cache-line-sized load-and-store instructions issued by any processor to the I/O space, e.g., the Block Load and Block Store instructions of the Sun Microsystems UltraSPARC implementations. We further assume support for two new remote-access operations not currently supported by the SCI-CLUSTER: the half-word-wide *put16* and *fetch-and-set16*. The fetch-and-set16 operation is launched by a "normal" half-word load operation and the put16 is launched by a half-word store to the remotely mapped I/O space. The network interface detects the half-word load and converts it into a fetch-and-set. The fetch-and-set16 operation will return the 16 bits of data that was stored in the remote memory and also atomically set the most significant byte of the data in the same remote memory.

The emerging InfiniBand™ interconnect proposal also has an efficient way to access remote memory without the involvement of a remote node using RDMA-RD and RDMA-WR [IB00]. It also includes support for remote atomic operations like compare-and-swap and fetch-and-increment.

2.2 DSZOOM Node Model

Each DSZOOM node consists of an SMP multiprocessor, e.g., the Sun Enterprise E6000™ SMP with up to 30 CPUs or the Pentium Pro Quad with up to four CPUs. The SMP hardware keeps coherence among the caches and the memory within each SMP node. The SCI-CLUSTER or InfiniBand™-like interconnect, as described above, connects the nodes. We further assume that the write order between any two endpoints in the network is preserved.

2.3 DSZOOM Blocking Directory Protocol Overview

The coherence protocol could easily be designed based on the original directory-based proposal [CF78] if it was not for the race conditions. However, the race conditions, caused by multiple simultaneous requests for the same cache line in combination with an unordered network, greatly complicate the protocol implementation. Blocking directory coherence protocols have been suggested to simplify the design and verification of hardware DSMs [HK99]. Typically, the directory blocks new requests to a particular cache line until all previous coherence activity to that cache line has ceased. This eliminates all the race conditions since

[3] We see no reason why this latency could not easily be cut in half using today's technology, and see this as a conservative number.

there only can be one cache line at a time that changes the directory state. The requesting node sends a completion signal to mark the completion of an ongoing activity.

The DSZOOM protocol is implemented in a distributed version of a blocking directory protocol. A processor that has detected the need for global coherence activity will first acquire a lock associated with the cache line before starting the coherence activity. A remote fetch-and-set16 operation to the corresponding directory entry in the home node will bring the directory entry to the processor and also atomically acquire the cache line's "lock" by setting the most significant byte (MSB) of the cache entry to all ones. If the MSB byte of the directory entry returned is already set, the cache line is "busy" by some other coherence activity. The fetch-and-set16 operation is repeated until the most significant byte is zero.[4] Now, the processor has acquired the exclusive right to perform coherence activities on the cache line and has also retrieved the necessary information in the directory entry using a single, 2-microseconds, operation. The processor now has the same information as, and can assume the role of, the "directory agent" in the home node of a more traditional SDSM implementation. Once the coherence activity is completed, the lock is released and the directory is updated by a single put16 transaction. No memory barrier is needed after the put16 operation since any other processor will wait for the most significant byte of the directory entry to become zero before the directory entry can be used again. Thus, the latency of the remote write will not be visible to the processor.

To summarize, we have enabled the requesting processor to momentarily assume the role of a traditional "directory agent," including access to the directory data, at the cost of one remote latency and the transfer of two small network packages. This has the advantage of removing the need for asynchronous interrupts in foreign nodes and also allows us to execute the protocol in the requesting processor that most likely would be idle waiting for the data. A further advantage is that the protocol execution is divided between all the processors in the node, not just one processor at a time as suggested in other proposals, e.g., [MFH$^+$96].

2.4 Protocol Details

The SMP hardware keeps the coherence within the node, on top of which the global SDSM protocol has been added. The SDSM state must be explicitly represented by data structures in the node. All the coherence activities and state names discussed in this thesis apply to the SDSM protocol.

[4] A random back-off scheme can be used to avoid a livelock situation, but has not been employed in DSZOOM yet.

The DSZOOM directory supports a MSI[5] protocol and has eight presence bits per cache line, i.e., can support up to eight SMP nodes. Figure 3 shows the example of the memory map for every process that is executed in the home node (NODE_ID = 0) together with the directory entry for one particular cache line. By convention, all cache lines in state INVALID store the "magic" data value BADBEEF,[6] as independently suggested by [SFH$^+$96, SGT96]. This value is checked for after each load to quickly determine if some global coherence activity is needed.[7]

Each node has 2 bytes of MTAG associated with each group of eight cache lines, divided into one lock byte and eight write-enable bits. Before each global store operation, the MTAG byte is locked by a local atomic operation, similar to the directory lock above. If the write-enable bit is set, the cache line is in state MODIFIED and the lock byte is cleared again directly after the store has been issued (assuming total store ordering (TSO), or processor consistency). A cleared MTAG bit indicates that the cache line is in either state INVALID or SHARED why some global coherence is needed (exemplified later in the thesis).

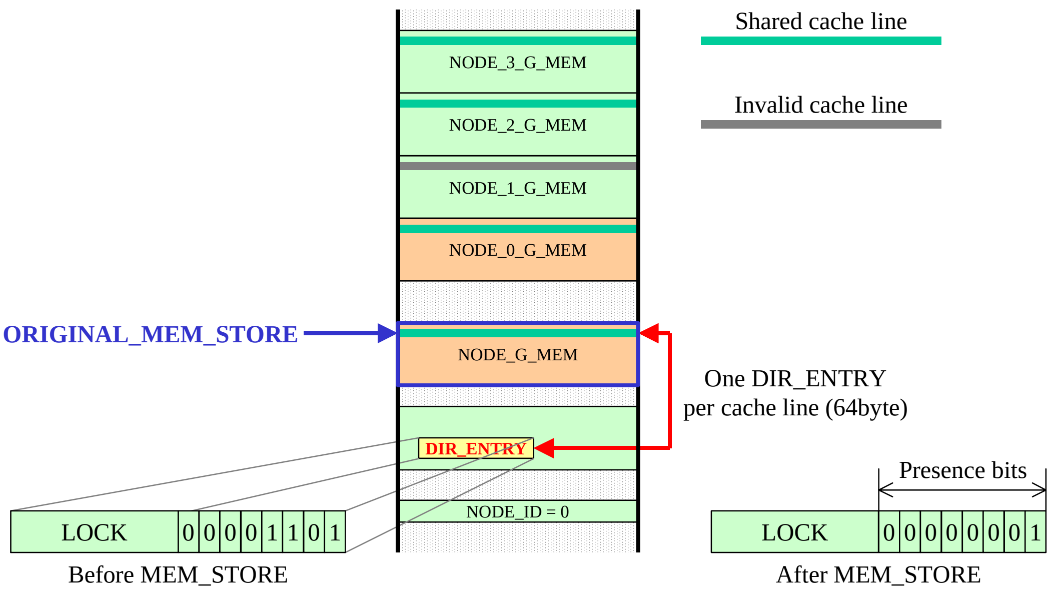


Figure 3: MSI protocol in "action." This figure assumes a four-node configuration.

MODIFIED cache lines in remote nodes are exclusively read and invalidated by initially locking the remote MTAG (fetch-and-set16) and a (prefetch) get data is issued back-to-back. After it has been determined that the MTAG has been locked,

[5] Modified-Shared-Invalid (MSI) is a classical three-state cache coherence protocol. More details about the MSI protocols can be found in many computer architecture textbooks, e.g., [CSG99].

[6] In this implementation the BADBEEF "magic" value is equal to a constant binary notation bit-pattern: 01010101…

[7] If the case the BADBEEF value was the intended data value, some unnecessary global activities have been created. This has proven to be a very rare event in all our studied applications.

BADBEEF is written to the remote data location using a put operation directly after which the MTAG is released by issuing a remote put16 (back-to-back) with the corresponding write-enable bit cleared the lock byte and set to zero.

The synchronization primitives of PARMACS macros are implemented directly using fetch-and-set16 operations rather than on top of the SDSM coherent memory.

3 Proof-of-Concept Implementation

This section describes our proof-of-concept implementation inside one SMP server. Because of the obvious reasons/assumptions described in the previous sections, we are currently unable to make a DSZOOM implementation on a real cluster interconnect. Instead, we logically divided one SMP into several "imaginary" SMPs, and rely on the DSZOOM protocol to keep the coherence between the imaginary SMPs. To make this kind of implementation more realistic we model the network delays for our virtual cluster as well by inserting some extra "dummy-loops" into the coherence protocol code routines. Our approach for the proof-of-concept implementation is illustrated in Figure 4. The unmodified SMP application is compiled with a standard gcc compiler and an m4 macro preprocessor using the modified/extended PARMACS macros (called PARMACZ in this implementation). The resulting file, the "(Un)executable," is than passed to our binary modification tool—Zeel. Zeel will insert the cache coherence protocol code into binaries by statically analyzing and modifying executables. Finally, the Zecutable (the executable running inside the virtual SMP cluster) is produced and can be used as if it was executed inside one SMP.

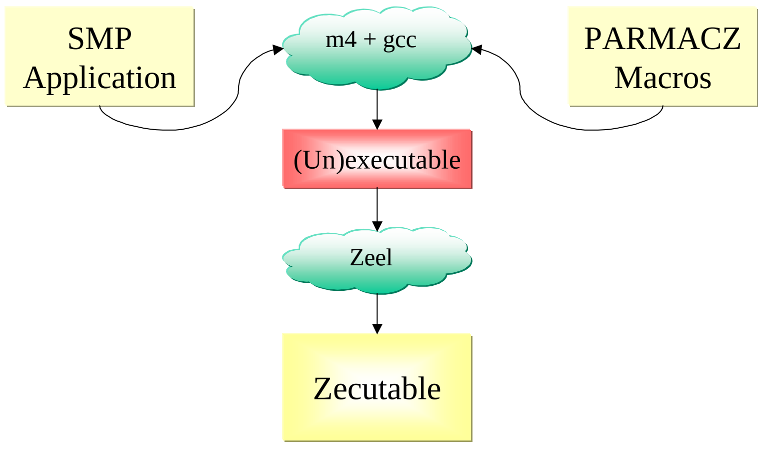
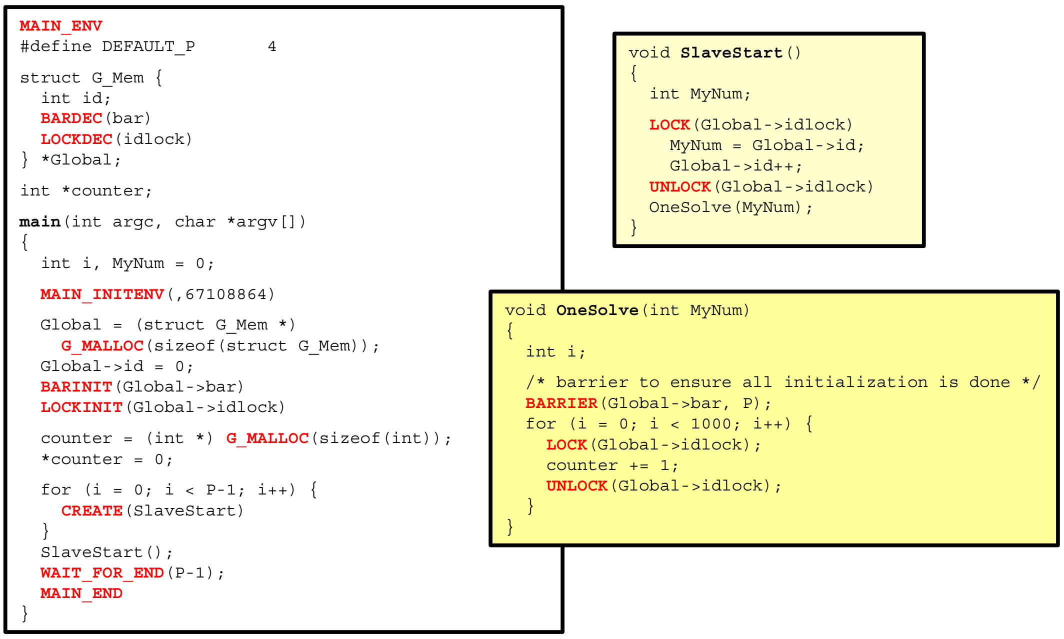


Figure 4: Our approach for the proof-of-concept implementation of a DSZOOM system. The Zecutable is an internal name for the executable running inside the virtual SMP cluster.

3.1 SMP/PARMACS Programming Model

One simple SMP application written with PARMACS macros is shown in Figure 5. These macros were developed at the Argonne National Laboratory. The original specification for PARMACS macros can be found in [BBD⁺87, LO87]. There are many different PARMACS macro implementations, implemented for the different programming models and for the different computer architectures, based on several execution and synchronization models, from classical Unix processes to multithreaded systems. Thus, example application in Figure 5, or any other PARMACS application for that matter, can easily be ported to many parallel computer architectures. Ideally, any modification/extension of any of the PARMACS macros should be reflected to all different PARMACS implementations as well. In this thesis, the target machines for the proof-of-concept implementation of a DSZOOM system are the Sun Enterprise SMP servers running Solaris 2.6 operating system, and therefore many of the macro modifications/extensions will only be valid for that type of architecture. One of the reasons why the SMP/PARMACS programming model is used in this thesis is that all of the popular SPLASH-2 applications from the benchmark suite are implemented with that relatively simple and portable model.



```
MAIN_ENV
#define DEFAULT_P        4

struct G_Mem {
  int id;
  BARDEC(bar)
  LOCKDEC(idlock)
} *Global;

int *counter;

main(int argc, char *argv[])
{
  int i, MyNum = 0;

  MAIN_INITENV(,67108864)

  Global = (struct G_Mem *)
    G_MALLOC(sizeof(struct G_Mem));
  Global->id = 0;
  BARINIT(Global->bar)
  LOCKINIT(Global->idlock)

  counter = (int *) G_MALLOC(sizeof(int));
  *counter = 0;

  for (i = 0; i < P-1; i++) {
    CREATE(SlaveStart)
  }
  SlaveStart();
  WAIT_FOR_END(P-1);
  MAIN_END
}
```

```
void SlaveStart()
{
  int MyNum;

  LOCK(Global->idlock)
    MyNum = Global->id;
    Global->id++;
  UNLOCK(Global->idlock)
  OneSolve(MyNum);
}
```

```
void OneSolve(int MyNum)
{
  int i;

  /* barrier to ensure all initialization is done */
  BARRIER(Global->bar, P);
  for (i = 0; i < 1000; i++) {
    LOCK(Global->idlock);
    counter += 1;
    UNLOCK(Global->idlock);
  }
}
```

Figure 5: Simple SMP/PARMACS application.

Short description of a PARMACS macros used in a SPLASH-2 benchmark suite can be found in Appendix A.

3.2 PARMACZ: Setting up the Memory-Mapped Communication

Modified/Extended PARMACS macros (called PARMACZ in this implementation) are responsible for, among many other things, setting up the memory-mapped communication between the processes inside one SMP. Address space layout and attachment of the shared memory objects for every process running in the home node (NODE_ID = 0) is shown in Figure 6. Shared memory objects with shared memory identifiers *A*, *B*, *C*, and *D* represents the physical shared/global memory of every node in the cluster. Shared memory identifier *E*, which is attached to the G_NODE_DATA area, contains the shared directory structure for the virtual SMP cluster and the global DSZOOM run-time system data. Local DSZOOM data for every process (e.g., NODE_ID) is stored in a privately mapped L_NODE_DATA area in a current implementation.

The main problem with spawning processes is that they are very expensive to create and destroy, and context switches (which involve changing the address space) also have a high cost. Nowadays, multithreaded operating systems and microkernels offer threads as a lightweight method to exploit parallelism. DSZOOM run-time system is currently implemented only with a fork-exec model, as shown in Figure 7. It should be possible to use some other lightweight method (e.g., POSIX-threads, Solaris-threads, or LWP-processes) inside one node at least, to better exploit parallelism in this type of systems.

The reason why we only use fork-exec model in this implementation is that it is quite easy to collect different type of statistics on "per-thread" basis, and that we currently do not run applications with huge number of threads, i.e., maximum number of threads is the number of CPUs in a system.

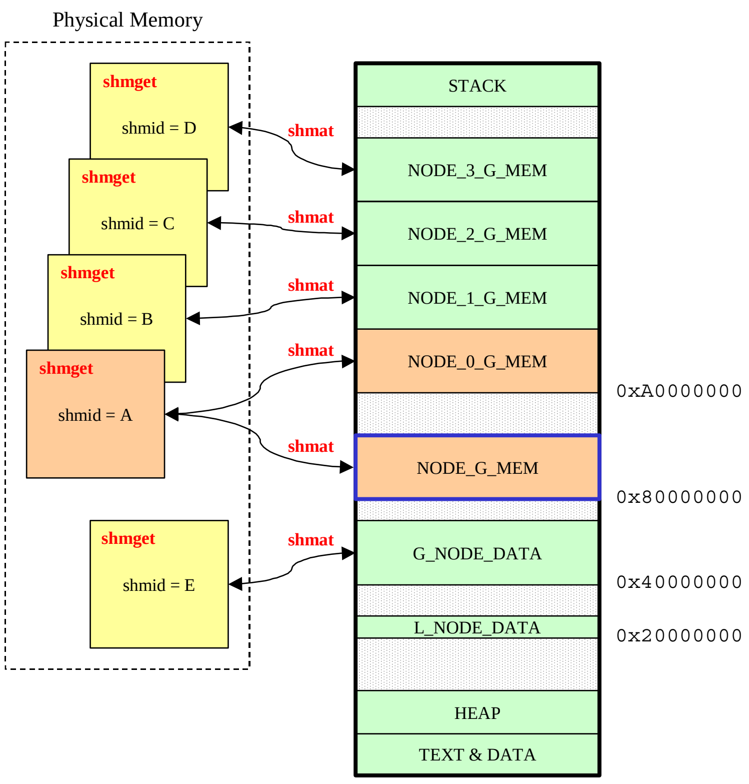


Figure 6: Address space layout and attachment of the shared memory objects for home node, NODE_ID = 0. This figure assumes that virtual SMP cluster consists of four nodes.

More information about modifications/extensions of the original PARMACS implementation can be found in Appendix A together with some brief description of the PARMACS macros used in the SPLASH-2 benchmark suite.

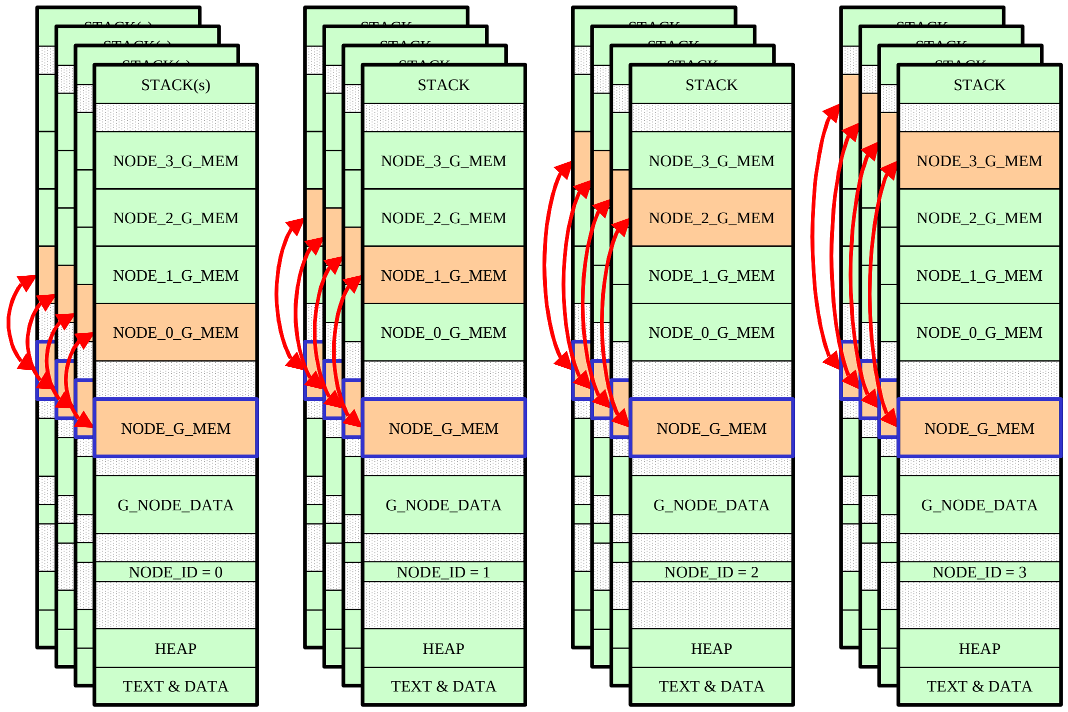


Figure 7: Four-Node virtual SMP Cluster example, with total of 16 CPUs. Fork-exec model is used here to exploit parallelism. Different type of counters are easily implemented and used on "per-thread" basis.

3.3 Zeel: Inserting Cache-Coherence Protocol Code into Binaries

There are at least three different ways of inserting cache-coherence protocol code into benchmark binaries. The most classical way of doing this is to make a compiler modification to perform that particular task, e.g., create a compiler backend that is capable of exchanging all relevant loads and stores with corresponding code snippets for those loads and stores. Dynamic code instrumentation (e.g., *JiTI*: a Robust Just in Time Instrumentation Technique, [RB00]) is another technique that also is capable of performing this task. The third alternative, the static binary instrumentation, is a technique usually described as a low-cost, medium-effort approach of inserting sequences of machine instructions into a program in executable or object format. Many tools have been written in the past that uses this technique for various purposes. All these tools work essentially in the same way, performing a sequence of three phases (discussed more in [AF96]):

1. Analysis
    a. Break code into functions and basic-blocks
    b. Analyze control transfers (jumps, branches, calls)
    c. Produce control flow graph (CFG)

2. Instrumentation
    a. Insert, change, and/or remove code
3. Regeneration
    a. Compute address translation
    b. Produce translation table if necessary
    c. Regenerate each basic-block, patching control transfers
    d. Update symbol table and relocation information if necessary

There are several successful applications of binary modification systems around (with both static and dynamic approaches); QPT [BL94] – a basic block counting tool for SPARC machines, Pixie [S91], Mahler [WP87], Epoxy [W92], EEL [LS95] – an executable-editing library, ATOM [SE94] and Alto [BD96] for Alpha machines, ETCH [RVL$^+$97] for Intel machines running Microsoft Windows NT operating system, and Paradyn for multiple architectures [MCC$^+$95, ZML99, XMN99].

For our purposes, EEL—a C++ library for machine-independent executable editing—seemed to be a good match, it was successfully used in several similar projects based on the UltraSPARC architecture, e.g., Blizzard-S [SFL$^+$94] and Sirocco-S [SFH$^+$98]. The source code is also freely available.

3.3.1 EEL Basics

EEL (Executable Editing Library) encapsulates the analysis (phase 2 in the previous subsection) and regeneration (phase 3 in the previous subsection) phases of binary instrumentation, allowing the tool programmer to focus on only the instrumentation phase. EEL's abstractions are similar to those found in compilers. The library handles executables (statically and dynamically linked), as well as object files. EEL presents the executable (or object file) to the programmer as a set of *routines*, each consisting of a collection of *basic-blocks*. A basic-block consists of a single sequence of straight-line *instructions*. Routines, basic-blocks, and instructions are some of the EEL abstractions (in form of C++ classes) that hide machine and system specific details from the tool programmer. For example, there are operations to obtain the registers that are read and written by an instruction. On the other hand, machine specific details are available if required, such as for example the binary opcode of an instruction.

EEL instrumentation is usually performed using *snippets* (another major EEL abstraction represented as a C++ class). A snippet is simply a sequence of binary machine instructions. Snippets can be inserted between any[8] instructions in basic-blocks, or along control flow edges.

Most of the Zeel implementation, in particular code snippets, is performed with the machine specific instructions (see Appendix B.1.2 and B.3.2 for examples how to create code snippets with EEL). The code in Figure 8 shows how easy it is to perform instrumentation on each routine in an executable.

```
int
main(int argc, char* argv[])
{
  executable* e = exec_kit()->open(argv[1], NULL);
  exec-read_contents();

  routine* r;
  e->read_contents();
  e->routines()->sort(cmp_routine_by_start);
  FOREACH_ROUTINE (r, e->routines())
  {
    instrument(r);

    while (!e->hidden_routines()->is_empty())
    {
      r = e->hidden_routines()->first();
      e->hidden_routines()->remove(r);
      e->routines()->add(r);
      instrument(r);
    }
  }

  addr new_start = exec->start_address();
  if (new_start != 0)
  {
    new_start = exec->edited_address(new_start);
  }
  exec->write_edited_executable(cat_string(argv[1], ".ins"),
                                new_start);
  return (0);
}
```

Figure 8: Example of how to instrument every routine in an executable with EEL.

Initially, the set of routines in an executable are defined by the functions appearing in the symbol table (if present), or by the entry point into the executable. Then, EEL performs call graph analysis incrementally, on a per routine basis. Analysis of a routine containing a call may expose the entry point of a new routine. The code in Figure 8 shows that after instrumenting a routine, the set of

[8] There are some places where snippets cannot be inserted, as, e.g., after a control transfer instruction. Typically, delay slot instructions are difficult to replace/modified. In this thesis, only non-optimized executables are instrumented with EEL, i.e., there are no delay slots what so ever.

hidden_routines is consulted, to see if any new routines were discovered. If so, the new routines are added to the set of regular routines and then instrumented as well.

More information about EEL details and how to use the library can be found in [L97].

3.3.2 Zeel Details

Zeel—our binary modification tool—has been developed with a version 4.0.1 of the EEL library and has been compiled with a gcc version 2.8.1.[9] In the current implementation, Zeel does not need to instrument accesses to non-shared, i.e., private data, which includes all stack and static data. Table 1 shows which SPARC instructions are instrumented with a Zeel tool. Every single instruction from the table below is replaced with the corresponding cache coherence protocol code snippet.

MEM_LOAD		**MEM_STORE**		**MEM_LOAD_ STORE**
Load Integer	**Load Floating-Point**	**Store Integer**	**Store Floating-Point**	**Load-Store Unsigned Byte**
LDSB	LDF	STB	STF	LDSTUB
LDSH	LDDF	STH	STDF	
LDSW	LDQF	STW	STQF	
LDUB		STX		
LDUH		STD		
LDUW				
LDX				
LDD				

Table 1: SPARC instrumentation.

The purpose of the proof-of-concept DSZOOM implementation is to demonstrate the implementation of a low-overhead global SDSM protocol, which is applicable to both page-based SDSMs and fine-grain SDSMs. The actual implementation of the low-level fine-grain instrumentation is still far from optimal. The code in Figure 9 shows the code snippet replacing each global 32-bit load instruction, lduw (usually only written as ld) to be more precise. See Appendix B.1.1 for a corresponding C routine—DSZOOM_MSI2_mem_load—that contains the cache coherence source code for global integer and/or floating-point loads. Examples of more efficient instrumentation can be found in both the Shasta [SG97] and the Sirocco-S [SFH+98]. For more details about the code instrumentation see Appen-

[9] Currently, it is not possible to compile the EEL library with more modern gcc compilers.

dix B.1 (for examples of instrumenting global integer loads), B.2 (for examples of instrumenting global floating-point loads), and B.3 (for examples of instrumenting global integer and/or floating-point stores).

```
     ld     [%o1 + 64], %DEST_REG          //original LD
     sethi  %hi(._start_FFT.EXE), %temp    //prepare for jmpl
     srl    %DEST_REG, 24, %g5             //mask BADBEEF
     cmp    %g5, 0xAA                      //check if BADBEEF
     bne    hit:                           //if not, it is a hit
     mov    %DEST_REG, %g6                 //prepare jmpl
     jmpl   %temp, %o7                     //jump link to C routine
     add    %o1 + 64, %g5                  //pass addr to C routine
     mov    %g6, %DEST_REG                 //move LD value from C
hit:
```

Figure 9: Replacing one load instruction, `lduw` to be more precise (partly-optimized). The `%temp` register is allocated with EEL at the insertion point. The `%g5` and `%g6` registers are global SPARC registers, currently not used or modified by any standard compiler.

3.3.3 Modeling the Network Delays

SDSM is run very efficiently in a single SMP node. In order to model a more realistic set-up with real network delay, the remote protocol implemented in C code have extra latency loops inserted. A remote access has about 2 microseconds of extra latency added.

We also wanted to compare our SDSM implementation to one with a more common, still short, remote latency caused by the extra protocol overhead. We have used the shortest latency reported to date as our benchmark number: 15 microseconds (Shasta [SG97, G00]). This is simply modeled as extra network delay. The extra CPUs occupancy by the protocol agent in the remote end have not been taken into account, nor have we modeled any contention effects from single threaded agent in that scheme.

4 Performance Study

4.1 Experimental Setup

All experiments in this thesis are performed on Sun Enterprise E6000™ SMP server running version 2.6 of Solaris operating system. Server has 16 UltraSparcII (250MHz) processors and 4GB of primary storage. Each processor has a 32kB L1 cache, and 4MB L2 cache respectively.

4.2 Benchmarks

The benchmarks we use in this study are well-known scientific workloads from the SPLASH-2 benchmark suite [WOT+95]. We study a total of eight completely unmodified SPLASH-2 applications from the original Stanford distribution, which were originally developed for hardware multiprocessors. The applications are:

- *Barnes-Hut* – This application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical N-body method.
- *FFT* – The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [B90]. This kernel is optimized to minimize interprocessor communication.
- *LU* – A classical method for blocked LU decomposition. It factors a dense matrix into the product of a lower triangular and upper triangular matrix. See [WSH94] for more details.
- *CLU* – Blocked LU decomposition with contiguous allocation of data. More optimized version of LU.
- *Radix* – Integer Radix sort kernel.
- *Radiosity* – This program computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [HSA91].
- *Water-nsq* – Water simulation without spatial data structure. This application evaluates forces and potentials that occur over time in a system of water molecules.
- *Water-sp* – Water simulation with spatial data structure. This application solves the same problem as Water-nsq, but uses a more efficient algorithm.

The benchmarks were compiled with System V IPC version of the PARMACS shared-memory macros used by Artiaga et al. [ANM+97, AMB+98]. The macro library was modified in several ways, e.g., we use user-level synchronization through test-and-set locks rather a System V IPC semaphore library calls. We also began all measurements at the start of the parallel phase to avoid measuring fork-system calls and DSZOOM run-time system initialization. The reason why we cannot run the entire SPLASH-2 application suite is only because of the different semantics for the System V fork-system call, and Solaris 2.6 fork-system call respectively. Solaris 2.6 fork-system call semantics are copy-on-write (COW), and five out of thirteen SPLASH-2 applications use global variables that are not allocated with G_MALLOC PARMACS-macro. If any child process modifies one global variable it will get its own copy of that particular variable and that change will not be visible for any other process in the running system. Of coarse, it should be possible to manually modify some of the SPLASH-2 programs to correctly declare and allocate every single global variable in the application to get rid of this problem.

The data-set sizes and uniprocessor-execution times for the studied SPLASH-2 applications are presented in Table 2.

Program	Problem Size	Non-Instrumented Sequential Time [s]
Barnes-Hut	16,384 bodies (32.8 MB)	36.91
FFT	1,048,576 points (48.1 MB)	15.72
LU	1024×1024, block 16 (8.0 MB)	86.07
CLU	1024×1024, block 16 (8.0 MB)	74.38
Radiosity	Test (29.4 MB)	8.79
Radix	4,194,304 items (36.5 MB)	29.18
Water-nsq	2197 molecules, 2 steps (2.0 MB)	86.73
Water-sp	2197 molecules, 2 steps (1.5 MB)	23.08

Table 2: Data-set sizes and sequential-execution times for the studied SPLASH-2 applications.

4.3 DSZOOM Performance Overview

Sequential-execution times for the instrumented SPLASH-2 programs are shown in Table 3. Efficiency overhead is between 1.42 and 1.93 for all of the studied applications, i.e., instrumented code takes between 42% and 93% longer time to execute the programs.

Program	% LD	% ST	Instrumented Sequential Time [s]	Efficiency Overhead
Barnes-Hut	52.1	51.9	65.59	1.78
FFT	40.5	23.5	22.96	1.46
LU	38.8	17.1	154.33	1.79
CLU	40.2	14.8	142.03	1.91
Radiosity	41.4	35.2	14.66	1.67
Radix	46.1	17.3	41.31	1.42
Water-nsq	45.0	32.5	166.21	1.92
Water-sp	43.7	27.7	44.54	1.93

Table 3: Sequential-execution times for instrumented SPLASH-2 applications with an efficiency overhead shown in the last column. Second and third columns show percentage of instrumented loads, and stores respectively. Both loads and stores are instrumented.

Table 4 shows the sequential-execution times if only global loads are instrumented. Note that the range check overhead is included for non-global (i.e., local/heap loads) loads. For integer intensive applications, e.g., Radix integer sort, the estimated cost is quite small, only 14 nanoseconds in this case. Floating-Point intensive applications/kernels has much larger cost, as expected, because BADBEEF-tests for floating-point loads are much more expensive. See Appendix B.1 and B.2 for more details about BADBEEF-test implementations.

Program	# Global Loads	Instrumented Sequential Time Loads [s]	Efficiency Overhead	Estimated Cost [ns]
Barnes-Hut	134,674,212	61.82	1.67	185
FFT	62,785,566	16.86	1.07	18
LU	741,587,877	119.28	1.39	45
CLU	744,920,712	100.07	1.35	34
Radiosity	50,794,436	14.04	1.60	103
Radix	554,534,181	36.94	1.27	14
Water-nsq	528,036,759	162.83	1.88	144
Water-sp	139,133,720	42.56	1.84	140

Table 4: Instrumenting only loads. Number of global loads is shown in the second column. Last column shows the estimated cost per load (in nanoseconds).

Table 5 shows the sequential-execution times if only global stores are instrumented. Note that the range check overhead is included for non-global (i.e., local/heap stores) stores. As expected, in the current implementation, the stores are generally more expensive then the loads because there are no speculative and/or optimistic tests (compare with the BADBEEF-tests for different type of global loads). See Appendix B.3 for details about how store snippets are currently implemented.

Program	# Global Stores	Instrumented Sequential Time Stores [s]	Efficiency Overhead	Estimated Cost [ns]
Barnes-Hut	3,131,395	40.20	1.09	1051
FFT	58,591,241	21.76	1.38	103
LU	360,012,816	128.86	1.50	119
CLU	360,021,011	107.13	1.44	91
Radiosity	2,268,365	9.14	1.04	154
Radix	293,703,383	33.65	1.15	15
Water-nsq	3,203,727	91.59	1.06	1517
Water-sp	3,068,968	25.01	1.08	629

Table 5: Instrumenting only stores. Number of global stores is shown in the second column. Last column shows the estimated cost per load (in nanoseconds).

Table 6 shows the results of our performance study for the three different configurations; (*i*) 8 CPU runs on SMP server, (*ii*) DSZOOM 2×4 (i.e., 2 nodes, 4 CPUs per node) with 3 microseconds network delay, and (*iii*) DSZOOM 2×4 with 15 microseconds network delay. Note that speedups for DSZOOM system are calculated relatively the DSZOOM execution times from Table 3.

Program	SMP 8 CPU Runs [time/speedup]	DSZOOM 2×4 (3 µs delay) [time/speedup]	DSZOOM 2×4 (15 µs delay) [time/speedup]
Barnes-Hut	5.11/7.2	8.95/7.3	9.65/6.8
FFT	2.45/6.4	3.65/6.3	6.02/3.8
LU	13.91/6.2	23.12/6.7	24.07/6.4
CLU	15.21/4.9	19.65/7.2	20.97/6.8
Radiosity	1.17/7.5	1.98/7.4	2.58/5.7
Radix	4.26/6.8	5.98/6.9	8.28/5.0
Water-nsq	11.95/7.3	23.11/7.2	25.06/6.6
Water-sp	3.82/6.0	6.52/6.8	7.57/5.9

Table 6: Execution times/Speedups for SMP server with 8 CPU runs, DSZOOM 2×4 (3 µs network delay), and DSZOOM 2×4 (15 µs network delay). NOTE: speedups for DSZOOM systems are calculated relatively the DSZOOM execution times from Table 3.

Figure 10 shows a graphical representation of the execution times for all configurations in this study.

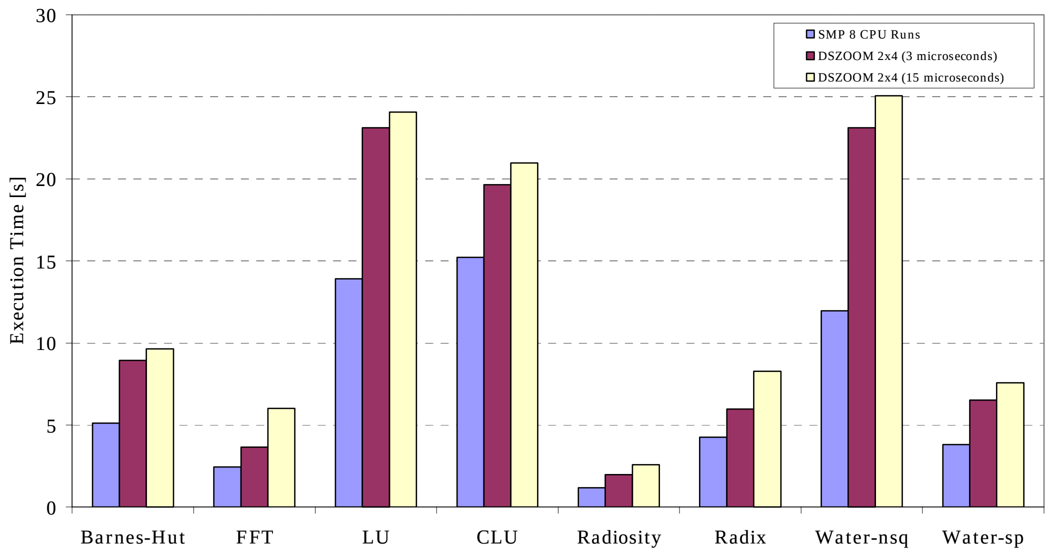


Figure 10: Execution times for the studied SPLASH-2 applications.

FFT and Radix are two of the applications that perform very well on the DSZOOM system compared with more traditional SDSM systems.

5 Related Work

Many different SDSM implementations have been proposed over the years (in alphabetical order): Blizzard-S [SFL$^+$94], Brazos [SB97], Cashemere-2L [SDH$^+$97, DGK$^+$99], CRL [JKW95], CVM [K96], GeNIMA [BLS99], Ivy [L88, LH89], MGS [YKA96], Munin [BCZ90, CBZ91], Shasta [SGT96, SGA97, SG97, SG97:2, DGK$^+$99], Sirocco-S [SFH$^+$98], SoftFLASH [ENC$^+$96], and TreadMarks [KCD$^+$94]. Most of them suffer from synchronous interrupt protocol processing. We see the work presented in this thesis as a complement to these activities and believe that most of these implementations would benefit from a more efficient protocol implementation.

The GeNIMA proposal is closest to our work. GeNIMA paper proposes a protocol and network solution to avoid some of the asynchronous overhead. A processor starting a synchronous communication event, e.g., the requesting processor initiating some coherence actions, checks for incoming messages at the same time. This avoids some of the asynchronous overhead in the home node, but will also add some extra delay while waiting for a synchronous event to happen in the node. The protocol is still implemented as communicating protocol agents.

6 Conclusions

We have demonstrated how asynchronous protocol processing can be completely avoided at the cost of some extra remote transactions – trading bandwidth for efficiency. The entire protocol processing for remote SDSM load operation on our DSZOOM implementation has been measured to be below 800 nanoseconds on a 400MHz UltraSparcII SMP system (Sun Enterprise E450 with 4GB of main memory). We believe that the total round-trip SDSM latency can be kept below three microseconds once the raw latency of a modern interconnects has been added. We demonstrate a substantial improvement in speedup for many of the SPLASH applications when we compare a modeled three microsecond SDSM system with the current state-of-the-art 15-microsecond.

The protocol technique described in this thesis is applicable to the emerging InfiniBand™ I/O interconnect proposal. We believe a protocol, such as the one we describe, could speed up many of the existing SDSM implementations on such interconnect.

7 Future Work

We plan to continue with this work in several different directions. First, cache-coherence protocol code optimizations are quite straightforward to implement and will give direct impact on the performance of the DSZOOM system. More detailed performance study, e.g., collecting the statistics per instruction type, is also straightforward to perform and will hopefully give us more information to be able to better understand the system and the behavior of the studied applications.

Second, we plan to port a DSZOOM to a real cluster interconnect with remote load/store semantics and remote fetch-and-set capabilities (e.g., to begin with, we could at least easily and more accurately model up to four-node configuration on the Sun-WildFire [HK99] prototype SMP cluster by binding the processes to nodes).

Third, instead of modeling the network delays by inserting the extra loops into cache coherence protocol routines, more accurate simulations could be produced with full system- and instruction-level simulators, e.g., simics [MDG$^+$98] and/or SimOS [RHW$^+$95]. That kind of simulators could also be used to more accurately model the contention effects from single threaded agents in the SDSM systems. Thus, we could make comparisons between DSZOOM and some other SDSM systems fairer.

Finally, to make this kind of system more usable it is desirable to make a POSIX-threads implementation as well because, currently, most of the commercial workloads are implemented with that popular programming model.

REFERENCES

[AF96] Arpaci, R. and M. Fähndrich. *RevEELing Solaris.* IRAM project, University of California of Berkley, May 1996. Available from: http://www.cs.wisc.edu/~remzi/papers.html.

[AMB+98] Artiaga, E., X. Mortorell, Y. Becerra, and N. Navarro. *Experiences on Implementing PARMACS Macros to Run the SPLASH-2 Suite on Multiprocessors.* In Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing, January 1998.

[ANM+97] Artiaga, E., N. Navarro, X. Mortorell, and Y. Becerra. *Implementing PARMACS Macros for Shared Memory Multiprocessor Environments*. Technical Report UPC-DAC-1997-07, Polytechnic University of Catalunya, Department of Computer Architecture, January 1997.

[B90] Bailey, D. H. *FFT's in External or Hierarchical Memory.* Journal of Supercomputing, 4(1):23–35, March 1990.

[BBD+87] Boyle, J., R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, 1987.

[BCZ90] Bennett, J. K., J. B. Carter, and W. Zwaenepoel. *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence.* In Proceedings of the Conference on Principles and Practice of Parallel Programming, 1990.

[BD96] Bosschere, K. and S. Debray. *Alto: A Link-Time Optimizer for the DEC Alpha.* Technical Report TR 96-15, Computer Science Department, University of Arizona, 1996.

[BL94] Ball, L. and J. R. Larus. *Optimally Profiling and Tracing Programs.* ACM Transactions on Programming Languages and Systems, 16(4):1319–1360, July 1994.

[BLS99] Bilas, A., C. Liao, and J. P. Singh. *Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems*. In Proceedings of 26th International Symposium on Computer Architecture, May 1999.

[CBZ91] Carter, J. B., J. K. Bennett, and W. Zwaenepoel. *Implementation and Performance of Munin*. In Proceedings of the 13th ACM Symposium on Operating System Principles, pages 152–164, October 1991.

[CF78] Censier, L. M. and P. Feautrier. *A New Solution to Coherence Problems in Multicache Systems*. IEEE Transactions on Computers, C-27(12):1112–1118, December 1978.

[CSG99] Culler, D. E., J. P. Singh, and A. Gupta. *Parallel Computer Architecture A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1st edition, 1999.

[DGK$^+$99] Dwarkadas, S., K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. *Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory*. In Proceedings of the 5th International Symposium on High-Performance Computer Architecture, pages 260–269, January 1999.

[ENC$^+$96] Erlichson, A., N. Nuckolls, G. Chesson, and J. Hennesssy. *SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory.* In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 210–220, October 1996.

[G00] Gharachorloo, K. Personal Communication, October 2000.

[HK99] Hagersten, E. and M. Koster. *WildFire: A Scalable Path for SMPs*. In Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture, pages 172–181, February 1999.

[HSA91] Hanrahan, P., D. Salzman, and L. Aupperle. *A Rapid Hierarchical Radiosity Algorithm.* In Proceedings of SIGGRAPH 1991.

[IB00] InfiniBand Architecture Specification, Release 1.0. InfiniBand(SM) Trade Association, October 24, 2000. Available from: http://www.infinibandta.org.

[JKW95] Johnson, K. L., M. F. Kaashoek, and D. A. Wallach. *CRL: High-Performance All-Software Distributed Shared Memory.* Operating Systems Review, 29(5):213–228, December 1995.

[K95] Keleher, P. *Lazy Release Consistency for Distributed Shared Memory*. Ph.D. thesis, Rice University, January 1995.

[K96] Keleher, P. *The Relative Importance of Concurrent Writers and Weak Consistency Models.* In Proceedings of the 16th International Conference on Distributed Computing Systems, May 1996.

[KCD+94] Keleher, P., A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. In Proceedings of the 1994 Winter USENIX Conference, pages 115–132, January 1994.

[L88] Li, K. *IVY: A Shared Virtual Memory System for Parallel Computing*. In Proceedings of the International Conference on Parallel Processing, pages 94–101, 1988.

[L97] Larus, J. R. *EEL Guts: Using the EEL Executable Editing Library.* Computer Sciences Department, University of Wisconsin-Madison, 1997. Available together with the EEL software distribution.

[LH89] Li, K. and P. Hudak. *Memory Coherence in Shared Virtual Memory Systems*. ACM Transactions on Computer Systems, 7(4):321–359, November 1989.

[LO87] Lusk, E. L. and R. A. Overbeek. *Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-Scheduling DO-Loop and Askfor Monitors.* Technical Report ANL-84-51, Revision 1, Argonne National Laboratory, June 1987.

[LS95] Larus, J. R. and E. Schnarr. *EEL: Machine-Independent Executable Editing*. In Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, pages 291–300, June 1995.

[MCC+95] Miller, B., M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. *The Paradyn Parallel Performance Tools.* IEEE Computer, 28(11):37–44, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.

[MDG+98] Magnusson, P. S., F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. *SimICS/sun4m: A Virtual Workstation*. In USENIX Annual Technical Conference, June 1998.

[MFH+96] Mukherjee, S. S., B. Falsafi, M. D. Hill, and D. A. Wood. *Coherent Network Interfaces for Fine-Grain Communication*. In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 247–258, April 1996.

[RB00] Ronsse, M. and K. De Bosschere. *JiTI: a Robust Just in Time Instrumentation Technique.* In Proceedings of the Workshop on Binary Translation, October 2000.

[RHW+95] Rosenblum, M., S. A. Herrod, E. Witchel, and A. Gupta. *Complete Computer Simulation: The SimOS Approach.* In IEEE Parallel and Distributed Technology, Fall 1995.

[RVL+97] Romer, T., G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. *Instrumentation and Optimization of Win32/Intel Executables Using Etch.* In Proceedings of the USENIX Windows NT Workshop, pages 1–7, August 1997.

[S91] Smith, M. D. *Tracing with Pixie.* Memo from Center for Integrated Systems, Stanford University, April 1991.

[S95] Scott, S. *The SCX Channel: A New, Supercomputer-Class System Interconnect*. In Proceedings of the Hot Interconnects III, 1995.

[SB97] Speight, E. and J. K. Bennett. *Brazos: A Third Generation DSM System.* In Proceedings of the 1st USENIX Windows NT Symposium, August 1997.

[SDH+97] Stets, R., S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. *Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network*. In Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997.

[SE94] Srivastava, A. and A. Eustace. *ATOM: A System for Building Customized Program Analysis Tools.* In Proceedings of the SIGPLAN ’94 Conference on Programming Language Design and Implementation, pages 196–205, June 1994.

[SFH+96] Schoinas, I., B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. *Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations.* Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[SFH+98] Schoinas, I., B. Falsafi, M. Hill, J. R. Larus, and D. A. Wood. *Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory*. In Proceedings of 6th International Conference on Parallel Architectures and Compilation Techniques, October 1998.

[SFL+94] Schoinas, I., B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. *Fine-Grain Access Control for Distributed Shared Memory*. In Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 297–307, October 1994.

[SG97] Scales, D. J. and K. Gharachorloo. *Design and Performance of the Shasta Distributed Shared Memory Protocol*. In Proceedings of the 11th ACM International Conference on Supercomputing, July 1997. Extended version available as technical report 97/2, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[SG97:2] Scales, D. J. and K. Gharachorloo. *Towards Transparent and Efficient Software Distributed Shared Memory*. In Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997.

[SGA97] Scales, D. J., K. Gharachorloo, and A. Aggarwal. *Fine-Grain Software Distributed Shared Memory on SMP Clusters*. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[SGT96] Scales, D. J., K. Gharachorloo, and C. A. Thekkath. *Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory*. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pages 174–185, October 1996.

[WG94] Weaver, D. L. and T. Germond. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[WOT$^+$95] Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 24–36, June 1995.

[W92] Wall, D. W. *Systems for Late Code Modification.* Technical Report 92/3, Western Research Laboratory, Digital Equipment Corporation, May 1992.

[WSH94] Woo, S. C., J. P. Singh, and J. L. Hennessy. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors.* In Proceedings of the 6th International Conference on Architectural Support for Programming Languages, pages 219–229, October 1994.

[WP87] Wall, D. W. and M. L. Powell. *The Mahler Experience: Using an Intermediate Language as the Machine Description.* In Proceedings of the 2nd International Symposium on Architectural Support for Programming Languages and Operating Systems, pages 100–104, October 1987.

[XMN99] Xu, Z., B. Miller, and O. Naim. *Dynamic Instrumentation of Threaded Applications.* In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, May 1999.

[YKA96] Yeung, D., J. Kubiatowicz, and A. Agarwal. *MGS: A Multigrain Shared Memory System.* In Proceedings of the 23rd International Symposium of Computer Architecture, pages 44–45, May 1996.

[ZML99] Zandy, V., B. Miller, and M. Livny. *Process Hijacking.* In Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, August 1999.

APPENDIX A

The PARMACS macros used in the SPLASH-2 applications are shown in Table 7 together with the short descriptions. More detailed information about these macros can be found in [AMN+97, LO87].

Macro	Description
MAIN_ENV EXTERN_ENV	Variables and symbol definitions for the PARMACS environment.
MAIN_INITENV MAIN_END	Initialization and termination of the PARMACS environment.
CLOCK	Get current time.
CREATE WAIT_FOR_END	Create a new process, starting in the specified routine. Wait for children processes to finish
G_MALLOC G_FREE	Allocate and deallocate shared memory.
LOCKDEC LOCKINIT LOCK UNLOCK	Declaration, initialization and usage of binary semaphores.
ALOCKDEC ALOCKINIT ALOCK AULOCK	Declaration, initialization and usage of arrays of binary semaphores. Note that they not provide atomic operations on several semaphores in the array simultaneously.
BARDEC BARINIT BARRIER	Declaration, initialization and usage of barriers.
GSDEC GSINIT GETSUB	Global subscripts management for self-scheduled loops. Each call to GETSUB returns a unique subscript from 0 to the maximum value specified. At the end of the loop, -1 is returned and each process waits for the others.
PAUSEDEC PAUSEINIT SETPAUSE CLEARPAUSE WAITPAUSE PAUSE EVENT	Operations for synchronization via event. PAUSEDEC declares an array of events. Each event can be set or cleared using SETPAUSE and CLEARPAUSE. The rest of the operations block processes waiting for a certain event to be set or cleared. PAUSE and EVENT reset the event when the caller is awakened.

Table 7: PARMACS macros used in the SPLASH-2 applications. From [AMB+98].

The benchmarks in our study were compiled with System V IPC version of the PARMACS shared-memory macros used by Artiaga et al. [ANM+97, AMB+98]. The macro library was modified in several ways, e.g., we use user-level synchronization through test-and-set (or to be more specific, a “fake” version of the fetch-and-set16 as described in the Basic Idea section) locks rather then System V IPC semaphore library calls. We also began all measurements at the start of the parallel phase to avoid measuring fork-system calls and DSZOOM run-time system initialization.

One typical implementation of the test-and-set spin-lock for the SPARC architecture using load-store unsigned byte (LDSTUB) instruction is shown in Figure 11. The LDSTUB copies a byte from memory into destination register, and then re-writes the addressed byte in memory to all ones. Typically, a nonzero value for the lock represents the locked condition, while a zero value means that the lock is free. Code busy waits by doing loads to avoid generating expensive stores to a potentially shared location. The reason to insert `membar #StoreStore` in `UnLockWithLDSTUB` is to make sure that pending stores are completed before the store that frees the lock. In DSZOOM implementation we do not use memory barriers at all because we rely on the global store order in the system. Note that this type of locking mechanism can only be used locally, not remotely. For remote locking we must rely on the fetch-and-set16 operation as described in the Basic Idea section.

```
LockWithLDSTUB(lock)

retry:
   ldstub     [lock], %l0
   tst        %l0
   be         out
   nop
loop:
   ldub       [lock], %l0
   tst        %l0
   bne        loop
   nop
   ba,a       retry
out:
   membar     #LoadLoad | #LoadStore

UnLockWithLDSTUB(lock)
   membar     #StoreStore    ! RMO and PSO only
   membar     #LoadStore     ! RMO only
   stub       %g0, [lock]
```

Figure 11: Lock and Unlock using LDSTUB. From [WG94], page 327.

Modifications/Extensions of the original PARMACS macros that are usually used for the SMP-programming are summarized in Table 8.

Macro	Modifications/Extensions
MAIN_ENV EXTERN_ENV	Variables and symbol definitions for the DSZOOM run-time system are added, e.g., TOTAL_NODES, DIR_ENTRY structure, …
MAIN_INITENV MAIN_END	Initialization and termination of the DSZOOM run-time system is added. All shared memory objects are created, attached, and initialized in the master process. Presence bits for every directory entry are set to 00000001_2, thus, it is only master process that allocate and initialize the global memory until the first occurrence of a CREATE macro is reached.
CLOCK	None.
CREATE WAIT_FOR_END	CREATE macro distribute processes to virtual nodes by setting the NODE_ID values in the L_NODE_DATA area. It will also, still sequentially, make a copy of a home nodes' memory to all other nodes in a system using the ordinary `memcpy` system call. Every directory entry is set to 11111111_2 before the start of a parallel execution, i.e., all cache lines are in the state SHARED.
G_MALLOC G_FREE	G_MALLOC macro will start memory allocation at 0x80000000 with a cache line size alignment (64bytes). G_FREE macro is not implemented.
LOCKDEC LOCKINIT LOCK UNLOCK	The user-level synchronization through test-and-set locks is used instead of a System V IPC semaphore library calls. Otherwise unchanged.
ALOCKDEC ALOCKINIT ALOCK AULOCK	The user-level synchronization through test-and-set locks is used instead of a System V IPC semaphore library calls. Otherwise unchanged.
BARDEC BARINIT BARRIER	The user-level synchronization through test-and-set locks is used instead of a System V IPC semaphore library calls. Otherwise unchanged.
GSDEC GSINIT GETSUB	The user-level synchronization through test-and-set locks is used instead of a System V IPC semaphore library calls. Otherwise unchanged.
PAUSEDEC PAUSEINIT SETPAUSE CLEARPAUSE WAITPAUSE PAUSE EVENT	The user-level synchronization through test-and-set locks is used instead of a System V IPC semaphore library calls. Otherwise unchanged.

Table 8: Modifications/Extensions to the original PARMACS implementation.

APPENDIX B

B.1 Instrumentation of the Global Integer Loads

Global integer load instructions are replaced with the following non-optimized code snippet (original integer load is typed in bold):

```
2c24c:  a2 02 00 09       add          %o0, %o1, %l1
2c250:  d0 02 00 09       ld           [%o0 + %o1], %o0
2c254:  8c 10 20 ff       mov          255, %g6
2c258:  8c 09 80 08       and          %g6, %o0, %g6
2c25c:  80 a1 a0 aa       cmp          %g6, 0xAA
2c260:  12 80 00 0a       bne          0x2c288
2c264:  0d 02 00 00       sethi        %hi(0x8000000), %g6
2c268:  8b 34 60 1a       srl          %l1, 26, %g5
2c26c:  80 a1 60 20       cmp          %g5, 32
2c270:  12 80 00 06       bne          0x2c288
2c274:  8c 11 a1 40       or           %g6, 320, %g6
2c278:  9f c1 a0 00       jmpl         %g6, %o7
2c27c:  8a 14 60 00       or           %l1, 0, %g5
2c280:  d0 04 60 00       ld           [%l1], %o0
2c284:  cc 31 60 00       sth          %g6, [%g5]
2c288:  ...
```

In the example above, the EEL library reserves the `%l1` register at the insertion point and uses it as a temp-register, to temporarily store the address for this particular global integer load. BADBEEF-Test is performed between the line $2c254_{16}$ and $2c260_{16}$. Range-Check is carried out between the line $2c268_{16}$ and $2c270_{16}$. At the line $2c278_{16}$, the `jmpl` instruction will jump to a cache coherence routine written in C, `DSZOOM_MSI2_mem_load`, shown in Appendix B.1.1. Finally, the line $2c284_{16}$ will unlock the directory entry and, at the same time, write back the current presence bits for that particular cache line. Appendix B.1.2 contains the EEL source code for a snippet generation.

B.1.1 DSZOOM_MSI2_mem_load

This is the cache coherence routine written in C that is called from the code snippets for global integer or floating-point loads in case when the global cache coherency action is needed. The routine below is compiled with the following gcc options (see gcc manual for descriptions):

```
   -r -O0 -mno-epilogue -Wa,-xarch=v8plus -mno-app-regs
```

```
void
DSZOOM_MSI2_mem_load(void)
{
  register unsigned int reg_i0 asm("i0");
  register unsigned int reg_i1 asm("i1");
  register unsigned int reg_i2 asm("i2");
  register unsigned int reg_i3 asm("i3");
  register unsigned int reg_i4 asm("i4");
  register unsigned int reg_i5 asm("i5");
  register unsigned int reg_i6 asm("i6");
  register unsigned int reg_i7 asm("i7");

  register unsigned int reg_g5 asm("g5");
  register unsigned int reg_g6 asm("g6");
  register unsigned int reg_g7 asm("g7");

  register unsigned char lock;
  register unsigned char node_id;
  register unsigned char my_mask;
  register unsigned char bits;
  register unsigned int cache_line_nmbr;
  register int i;

  struct DSZOOM_node_data *NODE_DATA;
  register struct DIR_ENTRY *DIR;

#ifdef PROFILE_FLAG
  struct DSZOOM_counters *NODE_COUNTER;
  unsigned long tick_start;
  unsigned long tick_end;
  unsigned long tick_total;
  double        time;

  GetTicks(tick_start);
#endif /* PROFILE_FLAG */

  lock = 0;

  NODE_DATA = (struct DSZOOM_node_data *) L_NODE_DATA_START_ADDR;
  DIR       = (struct DIR_ENTRY *) DIR_START_ADDR;

  node_id = NODE_DATA->node_id;
  cache_line_nmbr = (reg_g5 - NODE_G_MEM_START_ADDR) / CACHE_LINE_SIZE;
  my_mask = 1 << node_id;
  DIR += cache_line_nmbr;

  // Lock directory entry
  //---------------------
#ifdef NETWORK_DELAY
  if (node_id != HOME_NODE) {
    register int i;
    for (i = 0; i < LOOP_DELAY; i++) ;
  }
#endif /* NETWORK_DELAY */
  while (1) {
    asm("ldstub %1,%0" : "=r" (lock) : "m" (DIR->lock));
    if (lock == 0) break;
    while (DIR->lock);
  }

  bits = DIR->presence_bits;

  // Where is the data?
  // If not in my node:
  if (!(bits & my_mask)) {

    // Get cache line from another node (64byte)
    // Put local data (64byte)
    for (i = 0; i < TOTAL_NODES; i++) {
      if (bits & (1 << i)) {

#ifdef NETWORK_DELAY
        if (node_id != HOME_NODE) {
          register int i;
          for (i = 0; i < LOOP_DELAY; i++) ;
        }
#endif /* NETWORK_DELAY */

        memcpy((void *)(NODE_G_MEM_START_ADDR +
                        cache_line_nmbr * CACHE_LINE_SIZE),
               (void *)(G_BIG_MEM_START_ADDR +
                        i * NODE_G_MEM_MAX_SIZE +
                        cache_line_nmbr * CACHE_LINE_SIZE),
               CACHE_LINE_SIZE);

        break;
      }
    }
  }

#ifdef PROFILE_FLAG
  GetTicks(tick_end);
  tick_total = tick_end - tick_start;
  time = tick_total/CPU_FREQ;

  NODE_COUNTER = (struct DSZOOM_counters *) COUNTERS_START_ADDR;
  NODE_COUNTER->mem_load[NODE_DATA->proc_id]++;

  if (tick_total < NODE_COUNTER->mem_load_min_ticks[NODE_DATA->proc_id])
    NODE_COUNTER->mem_load_min_ticks[NODE_DATA->proc_id] = tick_total;

  if (tick_total > NODE_COUNTER->mem_load_max_ticks[NODE_DATA->proc_id])
    NODE_COUNTER->mem_load_max_ticks[NODE_DATA->proc_id] = tick_total;

  NODE_COUNTER->mem_load_tot_ticks[NODE_DATA->proc_id] += tick_total;
#endif /* PROFILE_FLAG */

  // Return DIR->lock addr via %g5
  reg_g5 = (unsigned int) &DIR->lock;
  // Return DIR_ENTRY via %g6
  reg_g6 = (unsigned int) bits | my_mask;

#ifdef NETWORK_DELAY
  if (node_id != HOME_NODE) {
    register int i;
    for (i = 0; i < LOOP_DELAY; i++) ;
  }
#endif /* NETWORK_DELAY */
}
```

B.1.2 DSZOOM_MSI2_load_snippet

This is how the code snippets for both global integer and global floating-point loads can be generated with EEL library:

```
code_snippet*
DSZOOM_MSI2_load_snippet(executable* exec,
                         const instruction* inst,
                         routine* r,
                         addr pc)
{
  const int max_inst   = 100;
  mach_inst* code      = new mach_inst[max_inst];
  int_reg_set* alloc   = new int_reg_set;
  mach_inst mi         = *inst->bits();

  int_reg_set* free_regs = new int_reg_set;
  int_reg_set* reserved_regs = new int_reg_set;

  const int_reg_set* int_regs;

  int_reg reg;
  int_reg TEMP_REG, ORIG_ADDR;

  int label;

  free_regs->add(REG_L0);
  free_regs->add(REG_L1);
  free_regs->add(REG_L2);
  free_regs->add(REG_L3);
  free_regs->add(REG_L4);
  free_regs->add(REG_L5);
  free_regs->add(REG_L6);
  free_regs->add(REG_L7);

  int_regs = inst->reads();
  FOREACH_REG(reg, int_regs)
    {
      reserved_regs->add(reg);
    }

  int_regs = inst->writes();
  FOREACH_REG(reg, int_regs)
    {
      reserved_regs->add(reg);
    }

  reg = inst->result_reg(r, pc);
  if (reg != NO_REG)
    {
      reserved_regs->add(reg);
    }

  reserved_regs->add(REG_G5);
  reserved_regs->add(REG_G6);
  reserved_regs->add(REG_G7);
  reserved_regs->add(REG_O7);

  free_regs->remove(reserved_regs);

  TEMP_REG = free_regs->first();
  free_regs->remove(TEMP_REG);

  ORIG_ADDR = free_regs->first();
  free_regs->remove(ORIG_ADDR);

  label = 0;

  switch (inst_category(inst->bits())) {

  case MEM_LOAD_INT:

    //test_bad_beef:
    if (IS_IMM(mi))
      {
        code[label] = ADD | OP2;
        code[label] = SET_RD(code[label], ORIG_ADDR);
        code[label] = SET_RS1(code[label], RS1(mi));
        code[label] = SET_IMM_BIT(code[label], IMM_BIT);
        code[label] = SET_IMM(code[label], (mi & IMM_MASK));
        label++;
      }
    else {
      code[label++] = make_2arg_inst(ADD | OP2, RS1(mi), RS2(mi), ORIG_ADDR);
    }

    code[label++] = mi;

    code[label++] = make_imm_inst(OR | OP2, REG_G0, 0xff, REG_G6);

    code[label++] = make_2arg_inst(AND | OP2, REG_G6, RD(mi), REG_G6);
```

```
    code[label++] = make_imm_inst(SUBcc | OP2, REG_G6, BADBEEF_1BYTE, REG_G0);

    code[label] = BNE | ICC;
    code[label] = SET_BR_ADDR(code[label], 10); // goto end:
    label++;

    code[label++] = make_sethi(REG_G6,
                               DSZOOM_MSI2_mem_load_routine_start_addr);

    //range_check:
    code[label++] = make_imm_inst(SRL | OP2, ORIG_ADDR, RANGE_CHECK_RIGHT_SHIFT,
                                  REG_G5);

    code[label++] = make_imm_inst(SUBcc | OP2, REG_G5, RANGE_CHECK_BITS,
                                  REG_G0);

    code[label] = BNE | ICC;
    code[label] = SET_BR_ADDR(code[label], 6); // goto end:
    label++;

    code[label++] = make_imm_inst(OR | OP2,
                                  REG_G6,
                                  DSZOOM_MSI2_mem_load_routine_start_addr &
                                  IMM_MASK,
                                  REG_G6);

    //jmpl_DSZOOM_MSI2_mem_load:
    code[label++] = make_imm_inst(JMPL | OP2, REG_G6, 0, REG_O7);

    code[label++] = make_move(ORIG_ADDR, REG_G5);

    code[label] = OP3 | (mi & OP3_MASK);
    code[label] = SET_RD(code[label], RD(mi));
    code[label] = SET_RS1(code[label], ORIG_ADDR);
    code[label] = SET_IMM_BIT(code[label], IMM_BIT);
    code[label] = SET_IMM(code[label], 0x0);
    label++;

    //unlock_dir_entry:
    code[label++] = make_imm_inst(STH | OP3, REG_G5, 0, REG_G6);

    //end:

    alloc->add(ORIG_ADDR);
    break;

  case MEM_LOAD_F_P:

    //test_bad_beef:
    if (IS_IMM(mi))
      {
        code[label] = ADD | OP2;
        code[label] = SET_RD(code[label], REG_G5);
        code[label] = SET_RS1(code[label], RS1(mi));
        code[label] = SET_IMM_BIT(code[label], IMM_BIT);
        code[label] = SET_IMM(code[label], (mi & IMM_MASK));
        label++;
      }
    else {
      code[label++] = make_2arg_inst(ADD | OP2, RS1(mi), RS2(mi), REG_G5);
    }

    code[label++] = mi;

    code[label++] = make_imm_inst(STF | OP3, REG_FP, -4, RD(mi));

    code[label++] = make_imm_inst(LDUB | OP3, REG_FP, -4, REG_G6);

    code[label++] = make_imm_inst(SUBcc | OP2, REG_G6, BADBEEF_1BYTE, REG_G0);
    code[label] = BNE | ICC;
    code[label] = SET_BR_ADDR(code[label], 10); // goto end:
    label++;

    code[label++] = make_sethi(TEMP_REG, DSZOOM_MSI2_mem_load_routine_start_addr);

    //range_check:
    code[label++] = make_imm_inst(SRL | OP2, REG_G5, RANGE_CHECK_RIGHT_SHIFT,
                                  REG_G6);

    code[label++] = make_imm_inst(SUBcc | OP2, REG_G6, RANGE_CHECK_BITS,
                                  REG_G0);

    code[label] = BNE | ICC;
    code[label] = SET_BR_ADDR(code[label], 6); // goto end:
    label++;

    code[label++] = make_imm_inst(OR | OP2,
                                  TEMP_REG,
                                  ZSDSM_MSI2_mem_load_routine_start_addr &
                                  IMM_MASK,
                                  TEMP_REG);

    //jmpl_DSZOOM_MSI2_mem_load:
    code[label++] = make_imm_inst(JMPL | OP2, TEMP_REG, 0, REG_O7);

    code[label++] = NOP;

    code[label++] = mi;

    //unlock_dir_entry:
    code[label++] = make_imm_inst(STH | OP3, REG_G5, 0, REG_G6);

    //end:

    alloc->add(TEMP_REG);
    break;

  default:
    break;
  }

  code_snippet* s = new code_snippet(code,
                                     label * sizeof(mach_inst),
                                     alloc,
                                     reserved_regs);

  delete[] code;
  return (s);
}
```

B.2 Instrumentation of the Global Floating-Point Loads

Global floating-point load instructions are replaced with the following non-optimized code snippet (original floating-point load is typed in bold):

```
        30f80:  8a 02 22 08        add          %o0, 0x208, %g5
        30f84:  c9 1a 22 08        ldd          [%o0 + 520], %f4
        30f88:  c9 27 bf fc        st           %f4, [%fp - 4]
        30f8c:  cc 0f bf fc        ldub         [%fp - 4], %g6
        30f90:  80 a1 a0 aa        cmp          %g6, 0xAA
        30f94:  12 80 00 0a        bne          0x30fbc
        30f98:  21 02 00 00        sethi        %hi(0x8000000), %l0
        30f9c:  8d 31 60 1a        srl          %g5, 26, %g6
        30fa0:  80 a1 a0 20        cmp          %g6, 32
        30fa4:  12 80 00 06        bne          0x30fbc
        30fa8:  a0 14 21 40        or           %l0, 320, %l0
```

```
30fac:  9f c4 20 00        jmpl         %l0, %o7
30fb0:  01 00 00 00        nop
30fb4:  c9 1a 22 08        ldd          [%o0 + 520], %f4
30fb8:  cc 31 60 00        sth          %g6, [%g5]
30fbc:  ...
```

In the example above, the `%g5` register will temporarily store the address for this particular global floating-point load. BADBEEF-Test is performed between the line 30f88$_{16}$ and 30f94$_{16}$. Range-Check is carried out between the line 30f9c$_{16}$ and 30fa4$_{16}$. At the line 30fac$_{16}$, the `jmpl` instruction will jump to a cache coherence routine written in C, `DSZOOM_MSI2_mem_load`, shown in Appendix B.1.1. Finally, the line 30fb8$_{16}$ will unlock the directory entry and, at the same time, write back the current presence bits for that particular cache line.

B.3 Instrumentation of the Global Stores

Global store instructions are replaced with the following non-optimized code snippet (original store is typed in bold):

```
2c2a0:  8a 02 60 a8        add          %o1, 168, %g5
2c2a4:  8d 31 60 1a        srl          %g5, 26, %g6
2c2a8:  80 a1 a0 20        cmp          %g6, 32
2c2ac:  12 80 00 18        bne          0x2c30c
2c2b0:  21 02 00 00        sethi        %hi(0x8000000), %l0
2c2b4:  0d 0f 00 00        sethi        %hi(0x3c000000), %g6
2c2b8:  8b 31 60 06        srl          %g5, 6, %g5
2c2bc:  8a 01 40 05        add          %g5, %g5, %g5
2c2c0:  8c 01 80 05        add          %g6, %g5, %g6
2c2c4:  ca 69 a0 00        ldstub       [%g6], %g5
2c2c8:  80 90 00 05        orcc         %g0, %g5, %g0
2c2cc:  02 80 00 07        be           0x2c2e8
2c2d0:  01 00 00 00        nop
2c2d4:  ca 09 a0 00        ldub         [%g6], %g5
2c2d8:  80 90 00 05        orcc         %g0, %g5, %g0
2c2dc:  12 bf ff fe        bne          0x2c2d4
2c2e0:  01 00 00 00        nop
2c2e4:  30 bf ff f8        ba,a    0x2c2c4
2c2e8:  ca 09 a0 01        ldub         [%g6 + 1], %g5
2c2ec:  80 a1 40 07        cmp          %g5, %g7
2c2f0:  02 80 00 04        be           0x2c300
2c2f4:  a0 14 22 8c        or           %l0, 652, %l0
2c2f8:  9f c4 20 00        jmpl         %l0, %o7
2c2fc:  8a 02 60 a8        add          %o1, 168, %g5
2c300:  d0 22 60 a8        st           %o0, [%o1 + 168]
2c304:  ce 31 a0 00        sth          %g7, [%g6]
2c308:  30 80 00 02        ba,a    0x2c310
2c30c:  d0 22 60 a8        st           %o0, [%o1 + 168]
2c310:  ...
```

In the example above, the range-check code is performed between the lines 2c2a0$_{16}$ and 2c2ac$_{16}$. Locking the directory entry is performed between the lines 2c2b0$_{16}$ and 2c2e4$_{16}$. Lines 2c2e8$_{16}$ – 2c2f0$_{16}$ will check if the current cache line is in the state MODIFIED (exclusive) or not (register `%g7` contains the presence bits mask for the current NODE_ID, e.g., if NODE_ID = 2, the bit-mask will be: 00000100$_2$). At the line 2c2f8$_{16}$, the `jmpl` instruction will jump to a cache coherence routine written in C, `DSZOOM_MSI2_mem_store`, shown in Appendix B.3.1. Finally, the line 2c304$_{16}$ will unlock the directory entry and, at the same time, write back the current presence bits for that particular cache line. Appendix B.3.2 contains the EEL source code for a snippet generation. Note that the instrumentation overhead shown in an example above can be much smaller if the range-check and locking mechanism are moved to the cache coherency routine instead.

B.3.1 DSZOOM_MSI2_mem_store

This is the cache coherence routine written in C that is called from the code snippets for global stores in case when the global cache coherency action is needed. The routine below is compiled with the following gcc options (see gcc manual for descriptions):

```
   -r -O0 -mno-epilogue -Wa,-xarch=v8plus -mno-app-regs
```

```
void
DSZOOM_MSI2_mem_store(void)
{
  register unsigned int reg_i0 asm("i0");
  register unsigned int reg_i1 asm("i1");
  register unsigned int reg_i2 asm("i2");
  register unsigned int reg_i3 asm("i3");
  register unsigned int reg_i4 asm("i4");
  register unsigned int reg_i5 asm("i5");
  register unsigned int reg_i6 asm("i6");
  register unsigned int reg_i7 asm("i7");

  register unsigned int reg_g5 asm("g5");
  register unsigned int reg_g6 asm("g6");
  register unsigned int reg_g7 asm("g7");

  register unsigned char node_id;
  register unsigned char my_mask;
  register unsigned char bits;
  register unsigned int cache_line_nmbr;
  register int i;

  struct DSZOOM_node_data *NODE_DATA;
  register struct DIR_ENTRY *DIR;

#ifdef PROFILE_FLAG
  struct DSZOOM_counters *NODE_COUNTER;
  unsigned long tick_start;
  unsigned long tick_end;
  unsigned long tick_total;
  double        time;

  GetTicks(tick_start);
#endif /* PROFILE_FLAG */

  NODE_DATA = (struct DSZOOM_node_data *) L_NODE_DATA_START_ADDR;
  DIR       = (struct DIR_ENTRY *) DIR_START_ADDR;

  node_id = NODE_DATA->node_id;
  cache_line_nmbr = (reg_g5 - NODE_G_MEM_START_ADDR) / CACHE_LINE_SIZE;
  my_mask = 1 << node_id;
  DIR += cache_line_nmbr;

  bits = DIR->presence_bits;

#ifdef NETWORK_DELAY
  if (node_id != HOME_NODE) {
    register int i;
    for (i = 0; i < LOOP_DELAY; i++) ;
  }
#endif /* NETWORK_DELAY */
```

```
  // Check if this node is the only node with the current cache line
  if (!(bits ^ my_mask)) {
    // Only this node has now the valid cache line
    // DIR->presence_bits = my_mask;

#ifdef NETWORK_DELAY
    if (node_id != HOME_NODE) {
      register int i;
      for (i = 0; i < LOOP_DELAY; i++) ;
    }
#endif /* NETWORK_DELAY */

    return;
  }
  else {

    // Check if this node has the current cache line
    if (!(bits & my_mask)) {
      // This node has not the current cache line
      // Get cache line from another node (64byte)
      // Put local data (64byte)
      for (i = 0; i < TOTAL_NODES; i++) {
        if (bits & (1 << i)) {

#ifdef NETWORK_DELAY
          if (node_id != HOME_NODE) {
            register int i;
            for (i = 0; i < LOOP_DELAY; i++) ;
          }
#endif /* NETWORK_DELAY */

          memcpy((void *)(NODE_G_MEM_START_ADDR +
                          cache_line_nmbr * CACHE_LINE_SIZE),
                 (void *)(G_BIG_MEM_START_ADDR +
                          i * NODE_G_MEM_MAX_SIZE +
                          cache_line_nmbr * CACHE_LINE_SIZE),
                 CACHE_LINE_SIZE);

          break;
        }
      }
    }
  }

 invalid_all:
  for (i = 0; i < TOTAL_NODES; i++) {
    if ((i != node_id) && (bits & (1 << i))) {
      // Invalidate

#ifdef NETWORK_DELAY
      if (node_id != HOME_NODE) {
        register int i;
        for (i = 0; i < LOOP_DELAY; i++) ;
      }
#endif /* NETWORK_DELAY */

      memset((void *)(G_BIG_MEM_START_ADDR +
                      i * NODE_G_MEM_MAX_SIZE +
                      cache_line_nmbr * CACHE_LINE_SIZE),
             BADBEEF_1BYTE,
             CACHE_LINE_SIZE);
    }
  }

#ifdef PROFILE_FLAG
  GetTicks(tick_end);
  tick_total = tick_end - tick_start;
  time = tick_total/CPU_FREQ;
  NODE_COUNTER = (struct DSZOOM_counters *) COUNTERS_START_ADDR;
  NODE_COUNTER->mem_store[NODE_DATA->proc_id]++;

  if (tick_total < NODE_COUNTER->mem_store_min_ticks[NODE_DATA->proc_id])
    NODE_COUNTER->mem_store_min_ticks[NODE_DATA->proc_id] = tick_total;

  if (tick_total > NODE_COUNTER->mem_store_max_ticks[NODE_DATA->proc_id])
    NODE_COUNTER->mem_store_max_ticks[NODE_DATA->proc_id] = tick_total;

  NODE_COUNTER->mem_store_tot_ticks[NODE_DATA->proc_id] += tick_total;
#endif /* PROFILE_FLAG */

  // Only this node has now the valid cache line
  // DIR->presence_bits = my_mask;

#ifdef NETWORK_DELAY
  if (node_id != HOME_NODE) {
    register int i;
    for (i = 0; i < LOOP_DELAY; i++) ;
  }
#endif /* NETWORK_DELAY */
}
```

B.3.2 DSZOOM_MSI2_store_snippet

This is how the code snippets for the global stores can be generated with EEL library:

```
code_snippet*
DSZOOM_MSI2_store_snippet(executable* exec,
                         const instruction* inst,
                         routine* r,
                         addr pc)
{
  const int max_inst   = 100;
  mach_inst* code      = new mach_inst[max_inst];
  int_reg_set* alloc   = new int_reg_set;
  mach_inst mi         = *inst->bits();

  int_reg_set* free_regs = new int_reg_set;
  int_reg_set* reserved_regs = new int_reg_set;

  const int_reg_set* int_regs;

  int_reg reg;

  int_reg TEMP_REG;

  int label = 0;

  free_regs->add(REG_L0);
  free_regs->add(REG_L1);
  free_regs->add(REG_L2);
  free_regs->add(REG_L3);
  free_regs->add(REG_L4);
  free_regs->add(REG_L5);
  free_regs->add(REG_L6);
  free_regs->add(REG_L7);

  int_regs = inst->reads();
  FOREACH_REG(reg, int_regs)
    {
      reserved_regs->add(reg);
    }

  int_regs = inst->writes();
  FOREACH_REG(reg, int_regs)
```

```
    {
      reserved_regs->add(reg);
    }

  reg = inst->result_reg(r, pc);
  if (reg != NO_REG)
    {
      reserved_regs->add(reg);
    }

  reserved_regs->add(REG_G5);
  reserved_regs->add(REG_G6);
  reserved_regs->add(REG_G7);
  reserved_regs->add(REG_O7);

  free_regs->remove(reserved_regs);

  TEMP_REG = free_regs->first();
  free_regs->remove(TEMP_REG);

  //range_check:
  if (IS_IMM(mi))
    {
      code[label] = ADD | OP2;
      code[label] = SET_RD(code[label], REG_G5);
      code[label] = SET_RS1(code[label], RS1(mi));
      code[label] = SET_IMM_BIT(code[label], IMM_BIT);
      code[label] = SET_IMM(code[label], (mi & IMM_MASK));
      label++;
    }
  else
    {
      code[label++] = make_2arg_inst(ADD | OP2, RS1(mi), RS2(mi), REG_G5);
    }

  code[label++] = make_imm_inst(SRL | OP2, REG_G5, RANGE_CHECK_RIGHT_SHIFT,
                               REG_G6);

  code[label++] = make_imm_inst(SUBcc | OP2, REG_G6, RANGE_CHECK_BITS, REG_G0);

  code[label] = BNE | ICC;
  code[label] = SET_BR_ADDR(code[label], 24); // goto nomiss:
  label++;

  code[label++] = make_sethi(TEMP_REG, DSZOOM_MSI2_mem_store_routine_start_addr);

  //spin_lock:
  code[label++] = make_sethi(REG_G6, 0x3C000000);

  code[label++] = make_imm_inst(SRL | OP2, REG_G5, 6, REG_G5);

  code[label++] = make_2arg_inst(ADD | OP2, REG_G5, REG_G5, REG_G5);

  code[label++] = make_2arg_inst(ADD | OP2, REG_G6, REG_G5, REG_G6);

  //spin_lock_retry:
  code[label++] = make_imm_inst(LDSTUB | OP3, REG_G6, 0, REG_G5);

  code[label++] = make_2arg_inst(ORcc | OP2, REG_G0, REG_G5, REG_G0);

  code[label] = BE | ICC;
  code[label] = SET_BR_ADDR(code[label], 7); // goto spin_lock_out:
  label++;

  code[label++] = NOP;

  //spin_lock_loop:
  code[label++] = make_imm_inst(LDUB | OP3, REG_G6, 0, REG_G5);

  code[label++] = make_2arg_inst(ORcc | OP2, REG_G0, REG_G5, REG_G0);
  code[label] = BNE | ICC;
  code[label] = SET_BR_ADDR(code[label], -2); // goto spin_lock_loop:
  label++;

  code[label++] = NOP;

  code[label] = BA | ICC;
  code[label] = SET_BR_ADDR(code[label], -8); // goto spin_lock_retry:
  code[label] = SET_ANNUL_BIT(code[label], ANNUL_BIT);
  label++;

  //spin_lock_out:
  //check_if_exclusive:
  code[label++] = make_imm_inst(LDUB | OP3, REG_G6, 1, REG_G5);

  code[label++] = make_2arg_inst(SUBcc | OP2, REG_G5, REG_G7, REG_G0);

  code[label] = BE | ICC;
  code[label] = SET_BR_ADDR(code[label], 4); // goto orig_store:
  label++;

  code[label++] = make_imm_inst(OR | OP2,
                              TEMP_REG,
                              ZSDSM_MSI2_mem_store_routine_start_addr &
                              IMM_MASK,
                              TEMP_REG);

  //jmpl_DSZOOM_MSI2_mem_store:
  code[label++] = make_imm_inst(JMPL | OP2, TEMP_REG, 0, REG_O7);

  if (IS_IMM(mi))
    {
      code[label] = ADD | OP2;
      code[label] = SET_RD(code[label], REG_G5);
      code[label] = SET_RS1(code[label], RS1(mi));
      code[label] = SET_IMM_BIT(code[label], IMM_BIT);
      code[label] = SET_IMM(code[label], (mi & IMM_MASK));
      label++;
    }
  else
    {
      code[label++] = make_2arg_inst(ADD | OP2, RS1(mi), RS2(mi), REG_G5);
    }

  //orig_store
  code[label++] = mi;

  //unlock_dir_entry:
  code[label++] = make_imm_inst(STH | OP3, REG_G6, 0, REG_G7);

  code[label] = BA | ICC;
  code[label] = SET_BR_ADDR(code[label], 2); // goto end:
  code[label] = SET_ANNUL_BIT(code[label], ANNUL_BIT);
  label++;

  //nomiss:
  code[label++] = mi;

  //end:

  alloc->add(TEMP_REG);
  code_snippet* s = new code_snippet(code,
                                     label * sizeof(mach_inst),
                                     alloc,
                                     reserved_regs);
  delete[] code;
  return (s);
}
```