Eager Sharing Protocol Implementation for GEMS

Vasily Tarasov
Computer Science
SUNY at Stony Brook
vtaras@cs.sunysb.edu

Dmytro Molkov
Computer Science
SUNY at Stony Brook
d_molkov@cs.sunysb.edu

ABSTRACT

In this paper we describe the implementation of Eager Sharing protocol using the Simics/GEMS combination. Eager Sharing is one of the cache coherency protocols which uses update instead of invalidate.

1. OVERVIEW

The main goal of the project was to implement the Eager Sharing protocol. This is a directory-based cache coherency protocol, with it's keystone being not using invalidate of the blocks in the caches, but rather update them if multiple caches are sharing them. The hope was that compared to the standard invalidation protocols update protocols will give us improvement in running time on multiple processor systems.

For the simulation of multiprocessor system running this cache coherence protocol implementation we used the Simics from Virtutech, which is full-system functional simulation infrastructure. Despite of its great functionality and reliability when simulating multiprocessor systems Simics lack the timing feature and has rather complicated tools for expanding and modifying functionality of the hardware simulation. That is why on top of Simics simulation we run the Ruby, which is a part of GEMS (General Execution-driven Multiprocessor Simulation) created in The University of Wisconsin. GEMS provide the Memory System Model, which gives the ability to get the timing of the simulation running and provides domain specific language for implementing cache coherency protocols.

During the work on the project we encountered quite a few problems, which we have managed to tackle and implement the basic model of the Eager Sharing protocol, which successfully runs on the simulation platform.

2. METHOD

First we will describe the Eager Sharing protocol in a nutshell. The main problem of all the protocols that use invalidation mechanism is that when the block is shared by multiple caches each time the value is being changed in one cache all the others caches have their copies of the value invalidated and as the result have to fetch the whole block from the memory again. Eager Sharing is an update, rather than invalidate, protocol. When a variable is shared, meaning it has been copied in more than one local cache, and has its value changed in one local cache by some CPU, the new value is passed up to the Eager Sharing global memory controller, which resends the value and address to all cache copies. On the other hand, changes to variables in cache-blocks copied into only one cache are quickly written into that local cache-block without accessing global memory. This change reduces the amount of data sent over the communication network since only the value that was changed is being sent across, not the whole block. The main states of the data blocks are the following:

- Cache-blocks that have no valid copy in the memory hierarchy have local state Invalid (I) if their tag appears in a local cache and global memory directory state Uncached (U).
- If there is only one copy of a variable, that is, its cache-block is present and valid in only one local cache, and if the values of all variables in its cache-block have not been changed from the values in global memory, the local cache-block copy has state Exclusive (E) and global memory directory state Read-only (R). A local CPU that uses that cache can change the

value of a variable in an Exclusive cache-block without sending the new value to global memory, but must change the cache-block's local state to Modified (M) and tell global memory to change its global state from Read-only (R) to Written (W).

- If there is only one copy of a variable and if some values in its local cache-block have been changed from what is in global memory, the local cache-block has state Modified M1 and global memory directory state Written (W). A CPU that uses the local cache can change the value of a variable in a Modified cache-block without sending the new value to global memory and without any other interaction with global memory.
- If there are multiple copies of a variable, including one copy in the local cache, the local cache-block has state Shared (S0 or S1) and global memory directory state Lots (L). All new values for addresses in the Shared cache-block have already been written to global memory. When a CPU that uses the local cache writes a new value for a variable in a Shared cache-block, the local cache does not immediately write the new value into its shared cache-block. Instead, the local cache sends the new value, the address, and the ID of the local cache to the global memory Eager Sharing controller, which multicasts the three data - the new value, its address, and the ID of the local cache that originated the new value - to all local caches that have a copy of this cache-block.

This approach however has an overwhelm when it comes to copies of the data that are not being used anymore but have not yet been replaced in the cache by some other blocks. We will call them orphan copies. To handle this problem Eager Sharing is modified to have periodic invalidate messages, when time-to-time it sends half-invalidating messages along with its update message. If the value is being invalidated twice the value is marked as invalid and will have to be fetched again when the CPU tries to access it. The implementation of the half-invalidation part of the protocol needs additional "half-invalidated" states E0 and S0 introduced. The fully valid states will then be called E1 and S1.

The full specification of the Eager Sharing Protocol that we used for our work can be found in [7].

To achieve the goal of the project we basically needed to introduce custom cache coherency protocol into the existing implementation of Simics and GEMS. As a tool for doing this we chose the SLICC (Specification Language including Cache Coherence), the domain specific language provided by the GEMS, which was designed to implement new cache coherence protocols and, while all the protocols existing in the system have their sources available in SLICC, also updating and improving them.

The main accent in implementing the protocol using this language is made on States, Events and Transitions, where the above work together in the following manner: States are the states of a basic State Machine that cache and directory use, Events are everything that influences and works with cache and Transitions define the state changes of the state machine according to events that occur in the system. Actions is the set of the actions performed on each transition from state to state. Moreover, SLICC provides developers with a set of tools to implement the communications between the caches, and between caches and the directory, this communication is done with the use of Messages of the format that can be defined by the developer to best fit the protocol which is being implemented.

Given all of this the very first thing to do is to reformat the definition of the Eager Sharing protocol into the States/Events/Transitions/Actions/Messages format.

While providing very powerful and useful domain-specific language GEMS have one big

flaw that makes it harder to implement the Eager Sharing protocol. The granularity of memory operations is the cache block. So the Eager Sharing protocol had to be modified in a way to satisfy this restriction. While it seems like the main idea of the protocol is being destroyed with this restriction, seems like there is more that one way to skin a cat. One of the workarounds that can be introduced to have the keystone of the protocol, update instead of invalideate, work is to introduce different latencies for the fetch of the block and update messages since update messages in the Eager Sharing protocol are carrying only one value that is being updated instead of the whole block and so have to be transmitted faster.

The Eager Sharing protocol is very similar to the MESI protocol, however it does have some very important differences. Let's walk through it step by step.

The state machines for the Directory and the Caches are totally different so the specification will be separated in two parts.

2.1 Cache

2.1.1 States

First we need to describe all the states that will be used by our machines. This list of states should not only include the base states of the protocol but also the transient states of the state machine, which are used when we are in the middle of the transition, i.e. cache sent request to the directory and is awaiting response.

The base states for the caches in the implementation of the Eager Sharing coherence protocol are:

- I – the invalid state of the cache block, basically means that this block is not yet in the cache, or became obsolete already, either way needs to be fetched from the memory.
- E – the state that specifies that the cache is the only owner of the block. In the advanced implementation of the protocol it is divided into two states E0 and E1, which are used for half invalidation procedure.
- M – the state, while in which the cache is the only owner of the data, that means that the memory contains the old version of the data and all writes and reads are done exclusively in the cache.
- S – the state that specifies that multiple caches are sharing the block, in this case both caches and directory should work on keeping the caches coherent. In the advanced implementation S is just like E divided into two states S0 and S1 for half invalidation.

According to the way of organization of the states we, as was mentioned before have to introduce transient states for the state machine. Those are described below:

- RS – the state that shows that the cache just did the read miss on the block and send a request to the memory but hasn’t got the response with the bock of the data.
- WS – the state for the write miss of the cache, when it has already sent a request and is waiting for the data to be delivered from the memory
- ST – the state of the cache block when it was marked as shared among multiple caches and the cache is writing the value, but hasn’t received the acknowledgment from the directory yet.
- EM – the transient state between the Exclusive and Modified states, i.e. block is writing the value into the Exclusively owned cache block and notified directory of its actions but hasn’t received the acknowledgment yet.
- MT – the state of the block that was exclusively modified inside of the cache, but is now sent back to the directory to be written by some other processor.

This is the full list of the cache block states that are used in the Simics/GEMS implementation of the Eager Sharing protocol.

2.1.2 Events

Here we describe the list of events that are driving the state changes for cache-blocks. These events can be divided into three parts: events originating in the processor, events that are driven by requests from the directory and events that are responses to the requests of the caches.

Processor Events:

- Load – instruction from the processor stating it needs the data from the specific address in the memory to be loaded.
- Ifetch – same as above but operates with the instructions instead of the data.
- Store – Processors write instruction.

Directory Request Events:

- FetchReadBlock – the event that states that a specified block that is exclusively modified in the cache is needed by the other cache for read and so has to be fetched into the memory.
- FetchedWriteBlock – same as above, but the cache block is requested for write not for read.
- UpdateWithAddress – event that occurs when the block that is being shared by multiple caches is being written by one of them. All the sharers receive the new value with this event saying they need to modify their instance of the block.
- ChangeLocalAddress_S – this event comes up when the block that was exclusively owned by one cache is being fetched by another processor and so the former exclusive owner has to change the state to shared.
- ChangeLocalAddress_E – the event notifying the cache that it is the only owner of the block left in the system since all other owners have replaced their instances with other blocks.
- Invalidate – event that is used by the directory to invalidate the block in the caches. While the Eager Sharing protocol uses update instead of invalidation this event is needed for the half-invalidation procedure.

Directory Response Events:

- NewBlockAddress_S – is sent by directory in response to the Read Miss or Write Miss of the cache, means that the block is present in other caches and so has to be marked as shared.
- NewBlockAddress_E – is sent by the directory in response to cache's fetch request, saying the cache is the first to access the block and so can work with it in Exclusive state.
- NewBlockAddress_M – is sent by the directory when the cache made a write miss and becomes the first owner of the block, i.e. the block is not present in any other cache in the system.

2.2 Directory

2.2.1 States

The situation with the states of the directory state machine is much simpler, while it has the same number of base states only one transient state has to be introduced:

- U – state of the directory that describes the block as Uncached, i.e. no cache in the system has the instance of the block.
- R – state of the directory for describing the block as being held by exactly one cache for exclusive read needs.
- W – this state specifies that the only valid copy of the block is held in exactly one cache and the block copy there is dirty, so directory doesn't has the current value of the block.
- L – this is the state specifying multiple caches are sharing the block at the time.
- WI – this is a transient state, directory enters it when it needs to get back the block that was exclusively modified by one of the caches, sent the request to the owner, but didn't receive the response yet.

2.2.2 Events

Directory Events are split into two groups: events that originated in the cache and events that are reactions to directory's requests to the cache.

Cache Requests:

- GETR – read miss occurred in the cache and this event appears in the directory.
- GETW – same as above but for the write miss in the cache.
- CHNG2W – the cache that was using the block as Exclusive now wants to write to it and so changes the state to Modified and is notifying the directory about it so directory could change the state of this block to Written.
- PUTL – the block was unloaded from one of the caches and the count of left sharers of the block is either 1 or 0. Directory should acknowledge the last sharer of the fact that it can now switch to Exclusive.
- PUTNL – the block was unloaded but there are still multiple sharers of the block.
- WRT2SHRD – this event takes place when one of the multiple sharers of the block sends request to the directory for the write into this block.

There is only one Cache Response Event:

- DTFRC – This is the response from the cache that had the dirty block and directory sent request for it.

Transitions for both the Directory and the Cache state machines are not specified here because of the size, but are included into appendix for the big picture.

There are about a dozen of already implemented cache coherence protocols for the GEMS, we choose one of them "MOSI_SMP_directory_1level" as the one that was closest by the states and transitions to the Eager Sharing, so we would not have to implement the whole system from scratch. The reason for this decision was basically the absence of the comprehensive up-to-date documentation for the SLICC language, so we basically had to learn by doing and having a template for it was very useful. The MOSI_SMP_directory_1level implements directory+snoopy protocol for the SMP system with only L1 caches, that is very close to the protocol that we were implementing with the exception of the snoopy part, but getting rid of it was not the hardest part. The transitions and the states of the protocol however took quite a while to get converted to our standard.

After the implementing of the protocol using the SLICC language is done, ruby has all the tools for compiling and building it into the part of Simics for further usage (look for details on building in the Appendix).

3. YET TO DO

Because of limited time, the project was originally oriented towards creating only minimal working implementation of Eager Sharing protocol. This goal was completely achieved. However a wide range of problems is not addressed in our work, so there is a vast expanse for future work. Section 6 describes main directions to follow for further improvements.

4. DIARY

The work on the Project can be logically divided in five major stages:

- Installing Simics and understanding how it works;
- Installing GEMS, getting it to work and understanding its principles;
- Eager Sharing protocol comprehension;
- Understanding SLICC syntax and semantic;
- Protocol implementation;

The description of these sections and timing in full workdays for each of them provided below.

4.1 Simics

Simics is a commercial product and requires a license. Thankfully academic license is freely available. However it takes several workdays to

get such license, so it should be done in advance. After you have a license it is easy to install Simics and get it to work. Some time is required to understand Simics scripting model and internal commands. The big thing is to feel how objects inside Simics are interconnected and how you can construct different systems just by connecting different components in different order. Note, that we used Simics 3.0.

Time Spent: 1-2 days plus time to obtain the license.

4.2 GEMS

We used GEMS 2.0, which was released only several months ago, so it is pretty crude. Moreover we installed it on Simics 3.0, what is not completely supported. These facts combined, lead to the necessity of relatively long time to get Simics + GEMS to work together. During setup procedure we had problems related to too new compilers, too old libraries, just library dependencies hell. Initially we wanted to work on x86_64 host, but as we found out Simics and GEMS do detect such hosts differently (Simics as x86_64, while GEMS as x86), it breaks all building process. So finally we were able to do it only on 32-bit host.

Time Spent: 3-4 days

4.3 Eager Sharing Protocol

Eager Sharing is not very hard, but it is still harder when usual MSI/MOSI/MESI/MOESI snooping protocols. So it takes some time to understand how it works in details.

Time Spent: 2 days

4.4 SLICC

GEMS uses special language called SLICC to describe cache coherence protocols. SLICC syntax is C-like, but it implies that the programmer knows a lot about the process of converting this SLICC code into the C++. It helps very much that GEMS has a lot of examples of already implemented protocols. Still it takes a considerable amount of time to stably understand all its features.

Time Spent: 2-3 days

4.5 Implementation

Implementation is definitely the hardest stage, because a lot of peculiarities that were not noticed before appear. Time is required to construct GEMS model that properly corresponds to the verbal description of a protocol. Debugging is one more time-consuming action, because debug methods in GEMS are not very developed.

Time Spent: 10 days.

5. IMPROVEMENTS

Simics and GEMS are easily available and can be installed on common computers and it is a big advantage of working on this project: no additional hardware or software facilities are required.

Authors believe that the work on the project of such difficulty requires more time. Having more time will allow to get a mature results at the end of the project, so for better achievements the work on this project has to be spread on the whole semester.

6. FUTURE WORK

First, protocol should be tested much more extensively than when it was done by authors. GEMS has a special tester module inside its framework, that allows testing cache coherence protocols in various marginal conditions. Such testing can discover deadlocks, race conditions, falling into not expecting states and other bugs that lead to wrong behavior of the protocol. Of course all uncovered problems should be fixed.

Second, the timing should be adjusted. As was already sad GEMS supports only block (not part of block) transferring. Therefore, to make the timing model work properly, we have to adjust virtual latencies carefully. At our current view, latencies of the messages must depend on the percent of changes that the message introduces to the block. Probably it will require some slight changes to the general protocol model.

Third, it is highly necessary to get real numbers from the simulation. The best way here is to develop an evaluation suite. Such suite should produce the special workload that will

allow noting the difference between protocols. Evaluation suite can be either internal or external relative to the Simics. During this work additional neglects of implementation can be uncovered and hopefully fixed.

Fourth, at the moment our model supports half-invalidated states (S0, E0), but periodical unload of orphan blocks is not supported. Implementation of this feature will allow measuring how it helps in increasing performance of the system.

The modifications needed for the half-invalidate part of the protocol to work can be done in either the EAGER_SHARING-cache.sm or EAGER_SHARING-dir.sm files. There is an event Invalidate already defined for the invalidation to take effect, all the states are already implemented too, so the only thing left is ***periodical*** invalidation. This can be done by holding the value in the directory corresponding to each block and on each write into the block while in shared this value would be decremented. When it reaches zero, directory should send the invalidation message (has to be further defined in the EAGER_SHARING-msg.sm as one of RequestTypes).

7. CONCLUSION

There is a belief that Eager Sharing protocol can increase the performance of massive parallel architectures under some workloads [1]. By implementing an Eager Sharing protocol for GEMS framework the authors give a mean to prove or disprove this position.

In addition GEMS became a universal instrument for measuring performance of various low level designs. Because of it, a lot of different protocols were implemented for GEMS during last years. However all implemented protocols before Eager Sharing were invalidate protocols. So this work shows a flexibility of GEMS infrastructure and gives the first example of update protocol implementation for GEMS.

8. REFERENCES

[1] "Eager Sharing for Efficient Massive Parallelism", Larry D. Wittie, Gudjon Hermannsson, Ai Li, Proceedings of Parallel Processing Conference, 1992

[2] "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill and David A. Wood, Computer Architecture News, September 2005

[3] "GEMS - ISCA 2005 Tutorial Slides", Mike Marty, Brad Beckmann, Luke Yen, Alaa Alameldeen, Min Xu, Kevin Moore, 2005

[4] Simics User Guide for Unix, Virtutech AB, 2006

[5] Simics/SunFire Target Guide, Virtutech AB, 2006

[6] http://www.cs.wisc.edu/gems/doc/wiki/moin.cgi, Multifacet GEMS Wiki

[7] "Eager Sharing Protocol Draft", Larry D. Wittie, 2007

[8] "Optimistic Synchronization in Distributed Shared Memory", Larry D. Wittie, Gudjon Hermannsson, International Conference on Distributed Computing Systems, 1994

APPENDIX

1. SUPPORTING MATERIALS DESCRIPTION

The archive submitted with the report consists of the following parts:

1.1 Eager Sharing Specs

In the root of the archive you can see the file called EagersharingHideMPDeepLatencyV1, which holds the specification for Eager Sharing protocol.

1.2 Eager Sharing implementation for GEMS

Files of the Eager Sharing implementation are placed in the *sources* folder, a brief description for each of them is provided:

1.2.1 EAGER_SHARING-dir.sm

This is a file that describes the behavior of the directory for the Eager Sharing protocol.

1.2.2 EAGER_SHARING-cache.sm

This file contains implementation of the state machine for the cache part of the Eager Sharing protocol.

1.2.3 EAGER_SHARING-msg.sm

The format of the messages is defined in this file.

1.2.4 EAGER_SHARING.slicc

The file that is used by GEMS build script when building the protocol, it contains the list of all the files that contain implementation of the protocol.

1.3 Documentation for the EAGER_SHARING

The second folder, *docs,* contains the automatically generated documentation of the implemented protocol. It has the tables describing State/Event/Transition model of the Eager Sharing implementation.

2. BUILDING AND RUNNING *EAGER_SHARING* FOR GEMS.

1. Extract the EAGER_SHARING implementation files from the zip archive into the $GEMS_HOME/protocols directory.
2. Go to the $GEMS_HOME/ruby folder and execute the following command:

   $ make PROTOCOL=EAGER_SHARING DESTINATION=EAGER_SHARING

   If GEMS was configured properly the compilation will succeed and as a result you will get a ruby module ready to be loaded into the Simics framework. Our configuration was SIMICS 3.0 + GEMS 2.0, so some differences may arise when compiling on different versions.
3. Once you compiled the module you have to import it into the Simics simulation. The detailed description of loading routine is described on the GEMS wiki [6]