CS 740, Fall 1998
Assignment 6:
Part I: Cache Protocol Verification with SMV

Assigned: Thurs., Nov. 19
Due: Fri., Dec. 4

Logistics

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page. There is no electronic handin for this assignment.

In the following, *AFS740* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15740-f98/public`

The files you need for this assignment are in the directory *AFS740*`/asst/asst6/cache`.

The problems in this assignment all make use of the model checker SMV. This program has been installed as *AFS740*`/bin/smv-`*SYS* for several CPU types: Solaris (*SYS* = `solaris`) and Linux (*SYS* = `linux`). If you want to run on a different machine, retrieve the file *AFS740*`/src/smv.r2.5.3.1b.tar` and do your own installation. This code is unlikely to work on a 64-bit machine, so don't try using an Alpha.

The problems in this assignment require running SMV many times, making incremental changes to the files (this is the easiest way to gain an understanding of the program). For each problem, show the fragment of the SMV file you are modifying, and indicate the response produced by SMV, i.e., that the verification succeeded, or describing the counterexample it produced. You *should not* include the detailed trace file. A verbal or pictorial explanation would be more useful.

One useful technique for maintaining multiple versions of an SMV description is to create a single "master" copy containing C-style conditional compilation directives. You then generate the different versions by running the C preprocessor using the appropriate set of command line arguments. For example, the version of `snoopy.smv` shown in the assignment directory was generated from the master version `snoopy.master`. The appropriate incantations are stored in the `Makefile`, including some good parameters for running SMV.

Exploring the Cache Protocol

The file `snoopy.smv` is an SMV description of a write-invalidate cache protocol. This protocol is similar to the one described in Section 8.3 of Hennessy and Patterson but is somewhat more complete in its details.

Problem 1: Tracing Interesting Behaviors

When SMV is run on the cache model, it simply returns a message stating that the verification succeeded. Often we'd like to be convinced that the protocol has the potential to do interesting things, as well. The best way to do this is to purposely run SMV with an incorrect specification where the desired behavior is a counterexample.

Show the following behaviors can actually occur. Give the temporal logic formulas you use to generate the behaviors, and describe the sequence generated:

1. We can reach a state where exclusive copies of blocks A and B are held in different caches.
2. An `invalidate` operation occurs on the bus.
3. A cache must give up its exclusively-held copy of a block due to a write miss by its processor.
4. A cache must give up its exclusively-held copy of a block due to a write miss by some other processor.

Eliminating XRead

Our protocol uses the a special "exclusive-read" (xread) operation to atomically read a block and invalidate all other copies. Alternatively, we could implement this process with two bus transactions: the first to get a read-only copy of the block and the second to invalidate all other copies.

Problem 2: Safety of Read-Invalidate

Modify the cache model so that it never issues an `xread` bus operation. Use SMV to show that the resulting protocol still satisfies the safety conditions.

Problem 3: Liveness Problems with Read-Invalidate

Describe the counterexample generated when we attempt to verify the liveness conditions for the modified protocol.

Problem 4: Making Read-Invalidate Fair

Devise a modification to the bus arbitration scheme that will guarantee that the modified protocol is fair. Use SMV to verify this.

Prefetch Operations

Most modern processors implement "prefetch" instructions, which provide a warning that some address is likely to be referenced in the near future. The processor may then attempt to move

the corresponding block into its cache in anticipation of the future read or write access. These instructions should never cause a memory exception, and ideally they should not impede the performance of other memory operations.

Suppose we wish to add a new instruction `prefetch` as one of the possible processor operations in our cache model. This operation would be implemented much like a read operation—attempting to get a copy of the block in the local cache. However, the processor would not be stalled while the operation is under way. If the processor issues a different memory operation while the prefetch is pending, the cache would abort the prefetch and move on to the new operation.

Adding Prefetching to the Cache Model

Show how you could modify the processor and cache models to support the prefetch instruction.

Use SMV to verify that the cache model is still correct.

Showing Interesting Prefetch Behaviors

Use the counterexample facility to generate the following scenarios:

- A successful prefetch occurs—there is a prefetch followed by a read hit.
- A prefetch is aborted.
- Prefetching by one processor impedes the performance of a write by another.