# Limes: An Execution-driven Multiprocessor Simulation Tool for the i486+-based PCs

## - User's Guide -
## v1.0

Department of Computer Engineering,
Faculty of Electrical Engineering,
University of Belgrade,
Serbia, Yugoslavia

LIMES web site: http://galeb.etf.bg.ac.yu/~igi

Note: This guide pertains to the execution-driven simulator of LIMES v1.0, but it can be useful for understanding of trace-driven simulator of LIMES v1.1.

# Contents

# 1. Introduction

Limes is a tool for simulation of multiprocessors. It runs on a PC with an i486 or higher CPU, on the Linux operating system (hence the name: Linux Memory Simulator). Limes creates an illusion of N somehow connected processors executing a parallel application, on a single-processor machine. Processes of the parallel application executing on these virtual processors are manipulated in such way that the result of their work is as it would be if they were executed on a real multiprocessor system. The applications that can be executed are regular C or C++ programs that use certain (ANL) macros for expressing parallelism, which makes them portable to real multiprocessor machines. Limes uses the so-called "execution-driven" approach, which ensures maximum simulation speed. Existing applications do not require source code modifications in order to be compiled and run under Limes.

There are two possible purposes of Limes:

- **Architecture evaluation studies**: You can simulate the network that connects the processors by coding it in C or C++, following a simple programming interface. Limes includes a complete example of four snoopy cache coherence protocols; this example models N processors with caches implementing one of the snoopy protocols, connected via a bus. One can either extend/modify the code that describes such system, or write his own code, describing the desired shared address-space multiprocessor system. This gives an opportunity for testing performance of different parallel systems that would be executing a real parallel application.
  Of course, you need a real parallel application as well. You can write your own or use available ones, like those from the SPLASH-2 suite (see [COT+95] for details on SPLASH-2).
- **Parallel algorithms evaluation**: In this case, you may be interested only in investigating efficiency of certain algorithms for parallel processing. Then you can neglect the architecture details like cache size, memory latency etc. and instruct Limes to simulate a perfect shared-memory system. In this manner you can learn about the efficiency of your algorithm, like the value of communication-to-computation ratio against the number of processors and so on. You can also extend such minimal simulator for modeling network latencies etc.

(Several tools of similar purposes exist, but mainly for MIPS machines. One example is TangoLite, developed at the Stanford University. See [Her93] for TangoLite. However, certain experiments indicate that the simulation with this tool is not bound to the behavior of Intel processors; refer to section 4.1.2 for more details.)

In order to run Limes, you need a PC, with an i486 or higher CPU and at least 8MB of RAM, and, of course, Linux (a widely popular free UNIX clone). While the version of Linux kernel should not matter, Limes requires GNU CC version 2.6.3. If your Linux uses a higher version (which is probably the case), you need to copy GCC 2.6.3 files into a separate directory. (This minor inconvenience results from the fact that the tool does various manipulations over the assembly code of the parallel application, and is therefore bound to the version of the compiler. Fortunately, copying GCC files into a Limes-dedicated directory does not affect the rest of the system, except for up to 11Mbytes of disk space less. Limes runs without problems on ELF Linuxes, provided that a.out support is installed, which is the default). No special privileges or modifications in the operating system are required to install and run Limes.

The next subsection will describe briefly the purposes of certain components of the simulation.. It is more than likely that you will be familiar with most or all of the terms defined in the sections that follow, but they are nevertheless included for completeness. (Also, please do not find yourself offended with occasionally somewhat simplified explanation style, since this manual is also to be used by undergraduate students attending the course in VLSI and parallel processing at our university.)

## 1.1 Structure of a simulation

Before we really begin, it will be explained shortly how the entire process is organized. First, you have a **parallel application**, which is a program created to be run on a system with N processors, hoping that it would run up to N times faster than if it were executed on a single processor. The application consists of N *threads* which operate in parallel, where in the best case each thread executes on a distinct processor. A good example is the SPLASH-2 program for Fourier transformation. You choose among many pre-written applications or write your own. Parallel applications written in SPLASH-2 manner are executed on Limes without source code alterations.

Then, there is a choice of **memory simulators**. A memory simulator simulates the entire architecture that lies underneath the basic processors and concerns with the issues of memory operations. It may include caches, bus, main memory and so on, but it does not, for example, have to include TLBs if they are not of interest for the architecture whose performance you wish to evaluate. With basic Limes, you can choose between an ideal memory system, or some of the snoopy (bus-based) cache coherence protocols, where the behavior of the system is simulated to a full detail. Or, you can select your own simulator. The rules for writing a simulator are described later in this document.

Finally, there is the simulation **kernel**. It is placed between the application and the simulator. The simulation is *execution-driven*. That means that the application executes on the host processor, while the simulation kernel catches only the events of interest. These may be references to shared data, or synchronization primitives, or all memory references. It is kernel's responsibility to stop and re-execute the threads as they generate events, and to call the actual memory simulator with events' descriptions at proper times. This approach ensures greatest efficiency, since machine instructions are not interpreted. It also ensures maximum accuracy, since the application code is compiled and optimized as it would normally be.

You can picture the relationship between the application, the kernel and the simulator like this:

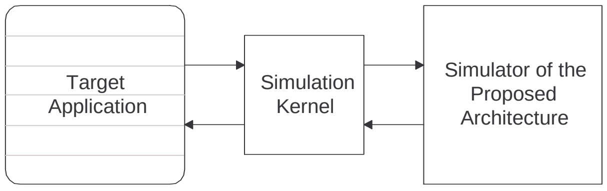


**Figure 1:** Relationship between the application, the kernel, and the simulator

Now it's easy to figure out what does the kernel do: it executes threads of parallel application, one thread at a time, with complete control over the execution, measuring the execution time for each thread in processor cycles. Whenever a thread tries to commit an operation which is of global significance (it may be shared, or even *any* memory reference, depending on your goal, but not a register operation, for example) it enters the kernel mode and the kernel decides who is the next to be scheduled, or whose memory request should be sent to the simulator to be simulated. When the simulator says that the request on processor #k is satisfied, then the kernel marks thread executing on that processor as "ready for execution". The event(s) which is the earliest in the global simulation time is scheduled, be it a request for a memory operation or a request of a ready-for-execution thread to return to user mode.

There is an analogy of the Limes kernel with the UNIX kernel: both control the execution of processes, and in the Limes case, memory operations that threads do can be viewed as system calls.

[Example: imagine that there exists a single process. At time, say, 1000 of some units (these units are, as will be seen, processor cycles) it creates a *thread* - a child process that shares the address space with its parent. Now there are two threads, each of which has reached the point t=1000 in time. They both operate on same data. Suppose that all we care about is shared memory references, since they affect both threads, and synchronization primitives. On a real multiprocessor, both threads would continue to execute, on different processors, in parallel. Assume that thread #0 would generate a shared read of variable V at time t=1050, and the thread #1 would generate a shared write to the same variable V at time t=1020. Between t=1000 and t=1050 thread #0 would work with registers and stack - none of which can affect thread#1, and the same applies to thread #1. So the kernel will schedule thread #0 for execution at t=1000. When the shared read is attempted at t=1050, the execution will stop and the kernel will resume control. Now it will see that the thread #1 is furthest behind in simulation time, and will activate him. At t=1020 he will try to do a shared write, and will be stopped. The kernel will conclude that the event {WRITE on processor #1, <write parameters> at time t=1020} is what would happen in reality, and will call the memory simulator, passing it the WRITE request parameters. Assume that the write finishes within 5 cycles. The actual write will be performed. Thread #1 will be executed again with time t=1025, and may generate a shared reference again at time, say, t=1100, and enter the kernel mode at that time. Now the kernel will call the simulator with

{READ on processor #0, <read parameters> at time t=1050}, and, when the read is simulated, the thread #0 will continue, having read the contents of the variable V written by thread #1 as it would have been if both threads had been executed on a real double-processor machine. Now extend 2 to N and this is how the N-processor system is simulated. But it will be described in much greater detail later.]

# 2. Quick start

This section will instruct you how to install the package and how to run a few examples. Having seen what it looks like, it will be easier to get more into details, described in section 3.

## 2.1 Installation

You do not need any privileges to install and run Limes, modulo the installation of version 2.6.3 of the GCC compiler, if it is not installed on your system already. The procedure is described below.

For the purposes of simplicity, it will be assumed that your username is **joe**, and that your home directory is **/home/joe**.

### 2.1.1 Unpacking the archive

Limes comes in a single archive, **limes*.tgz**, where * stands for the current version. While this procedure describes how to install Limes in a per-user manner, you can also choose to install it on the system to be globally accessible.

To unpack the archive, simply position in your home directory and do

```
tar xfz limes*.tgz
```

After this, a directory tree will be created.

#### 2.1.1.1 The Limes directory tree

After unpacking, the created tree will look like this:

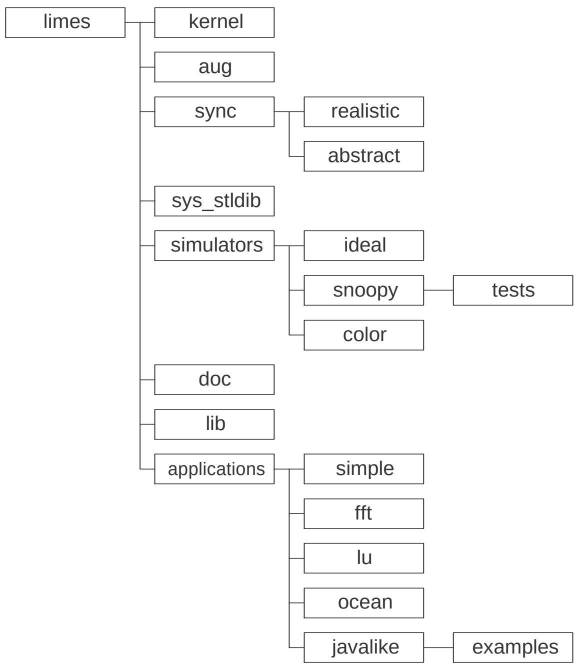


**Figure 2:** Limes directory tree

- The **kernel** directory contains all the kernel source files and the include files.
- **aug** contains utilities for instrumentation of the assembly code of the parallel application being compiled.

- **sync** is the parent directory for all the implementations of synchronization primitives; two implementations are provided with Limes: *realistic* and *abstract*, and the user can put his own here
- **sys_stdlib** includes certain standard library functions that need to be instrumented
- **simulators** comprise of a perfect memory simulator, four snoopy protocols (namely Berkeley, Dragon, WTI and WIN), and one simple for visualization of data accesses. The root of this sub-tree contains some utilities for developing your own simulators. If you develop your own simulators, they will be stored in directories like **yoursim**.
- **applications** are three SPLASH-2 applications and a simple custom application. They are all ready to be compiled for simulation. The **javalike** directory contains an implementation of threads and monitors for C++, similar to those in the Java language; the examples below it include a couple of simple parallel applications written in that manner. See Appendix D for details.
- **lib** is the directory where intermediate results of previous compilations are stored to speed up the compilation process.

There may be other directories, too. All the directories are accompanied with a README file where the purpose of each file in the directory and the directory itself is explained, along with some hints on usage.

### 2.1.2 Setting up the environment

Limes scripts and makefiles must know where the whole tree is. To communicate this information, set the environment variable **LIMESDIR** to point to the root of the tree. In the example, do

```
export LIMESDIR=/home/joe/limes      # if you use bash, or
setenv LIMESDIR /home/joe/limes      # if you use tcsh
```

(You can put them in your .profile or .login for convenience).

### 2.1.3 Installing the GNU C v2.6.3 compiler

Check the version of your GCC compiler with “gcc -v”. If it reports **2.6.3** or lower, the installation procedure is over. Otherwise, you will probably have to be root (or to ask one) to copy the development environment for the GCC 2.6.3 version on your filesystem. **Note: do not really *install* it, for it will overwrite your current GCC! Just read below how to *copy* it onto your filesystem, in a separate, harmless directory! Limes will know how to find and execute it.**

As explained, Limes does various things with the application’s assembly code, and is therefore strongly dependent on the version of the compiler. But the 2.6.3 version does not have to be the default compiler on your system, of course. It will reside in a sub-directory and be called only by the Limes. Here is how to do it:

1) First, get the Development disk series of the Linux distribution that contains GCC 2.6.3. An example is Slackware 2.2.0.1

2) You will need the following archives: **gcc263.tgz**, **include.tgz**, **libc.tgz**, **libgxx.tgz**, **lx128_2.tgz**, **binutils.tgz**.

3) create a directory, say, **/oldgcc**. Change the ownership of this directory to make it belong to a non-privileged user. Now login as that user and cd to /oldgcc.

4) From now on, everything will be done from the directory /oldgcc as the current, and all the paths will be relative to that directory.

5) unpack the archive d1/gcc263.tgz (like `tar xfz /cdrom/slackware/d1/gcc263.tgz`)

6) unpack d6/lx128_2.tgz. Do

```
(cd usr/src/linux-1.2.8/include; ln -sf asm-i386 asm)
   (cd usr/src; ln -sf linux-1.2.8 linux)
```

7) unpack d2/include.tgz. Do

```
(cd usr/include; ln -sf /oldgcc/usr/src/linux/include/linux linux)
   (cd usr/include; ln -sf /oldgcc/usr/src/linux/include/asm asm)
```

8) unpack d5/libc.tgz. Do `sh install/doinst.sh`

9) unpack d4/libgxx.tgz. Do `sh install/doinst.sh`

10) unpack d8/binutils.tgz

There is one more step: edit **limes/globals.make** file in your Limes tree, and replace “ifeq (0,1)” with “ifeq (1,1)”. If you have chosen another directory for GCC2.6.3 instead of “/oldgcc”, change that in the file, too.

The procedure is now over. It will not affect the rest of your system, except that you will now have 11MB of disk space less.

**If you do not have access to a Linux distribution that contains GCC v.2.6.3 files, you can download it from the following URL:** `http://galeb.etf.bg.ac.yu/~dav0r/limes/oldgcc.html` **(or you will find a pointer to a site that contains it). It is in fact somewhat reduced, and requires only 7MB of space (the archive itself is 2.4MB long).**

## 2.2 Compiling and running a simulation

Two examples will be described, both for the same parallel application. One example is running the application with the ideal memory system, and the other is running it with a realistic memory system. Both procedures are extremely simple.

First, select an application to run. Let it be **FFT**. You will find it in **limes/applications/fft**. Just cd there. You will find a **makefile**, which controls the compilation process. Everything in Limes is done through the make files. Now open the makefile with your editor.

### 2.2.1 Simulation with an ideal memory system

Inspect the makefile and take a look at two variables, named **SIM** and **SIMHOME**. If they are not commented out, put a hash (#) before them. Leaving the simulator unspecified will make Limes to select an ideal memory system as the default simulator. This systems assumes that all the reads and writes complete in a single cycle.

Exit the editor and type

```
make
```

If everything is installed correctly, an executable will be produced, named **FFT**. It consists of the parallel application, the simulation kernel, and the ideal memory simulator, all in one file.

Now run FFT telling it that it has to perform FFT over 1024 complex points and that only one processor is simulated. You do that by specifying

```
FFT -m10 -p1
```

(m is a switch for problem size, p specifies the number of processors. This has nothing to do with Limes, it is the calling syntax of the FFT application.)

Let it execute and watch the results. The FFT will tell you how much time it spent in processing the data.

Now tell FFT that you want it to run on four processors. Do it with

```
FFT -m10 -p4
```

The FFT will think that it really executes on four processors. Watch the result when it finishes (it will happen very quickly) and you’ll see that the simulated time spent in processing is nearly four times less than it was in the case of a single processor. The total simulated time will not be four times less, since the same job of data initialization had to be done by the master processor in both cases.

Try to run FFT with other problem sizes and number of processors. The default maximum number of processors is 16. If you want more, you should edit a certain header file an do the make again; for details refer to section 4.

### 2.2.2 Simulation with a realistic memory system

Edit the same makefile as you did before and uncomment the variables **SIM** and **SIMHOME**. They should be set to **ber** and **snoopy**, respectively. This information tells Limes that the simulator that has to be linked with the application is in the **limes/simulators/snoopy** directory, and that the simulator's name is **ber**, which stands for the Berkeley protocol.

Save, exit, and do

```
    make
```

again. You will get the "FFT" executable, but this time it will contain the simulator that simulates complete caches, bus and the memory module that N Intel P6+ CPUs with implemented Berkeley cache consistency protocol would have.

Run it as you did before. You will see different, more real times after the FFT finishes, and you will also see that the simulation runs slower (but still not too slow). The reason is that for every single memory operation FFT threads make, the simulator simulates data being looked at in caches, showing on the bus, going to main memory, being invalidated by other processors and so on, much like a real hardware would do. You will see that the simulator does not have to include so many details while it still can retain reasonable accuracy. But it is always a trade-off between accuracy and {performance and complexity}.

If you want to see how it would run if the cache had 256K instead of the default 8K, run it like

```
    FFT -m12 -p4 -- -dcache_size=256
```

### 2.2.3 Simulation without simulating references

This option is similar to the first one, with an exception that you cannot produce traces but you will get a simulation that runs several times faster, and is designated for those who investigate parallel algorithms. All the reads and writes also complete in one cycle, and the correct order is preserved (a read or a write is deferred until its time stamp reaches global simulation time). The difference is only that requests do not pass through the simulator (but an ideal simulator does nothing with them anyway), and, of course, in the execution speed. Another difference is that this scheme uses, by default, *abstract* implementation of synchronization primitives; more about it later.

Edit the makefile as you did before and comment out the variables **SIM** and **SIMHOME** again. Now uncomment the line **ALGEVALONLY=1** (or add it if it does not exist). Do a **make totalclean**, and then **make**. Now run FFT again and notice the difference in the wall-clock time elapsed during the simulation. When you are finished, comment out ALGEVALONLY again, and do **make totalclean** again. You can try the same for the **LU** application and see how less scalable it is.

# 3. Interface to the simulation process

This section will describe the user-to-Limes interface that will enable you to program and compile any parallel application that follows certain standards. More advanced topics, like the organization of the simulator and the rules that should be obeyed when writing a custom memory simulator, are left for further sections.

## 3.1 Applications

Limes was written primarily with the purpose to allow the SPLASH-2 applications to run on multiple simulated processors. These applications should be able to compile without change (in fact, the compiler might complain for some of them, like “Warning: PI redefined”; this is not due to Limes but to the difference in header files for GCC and the SGI-IRIX CC, or whichever). The SPLASH-2 applications use a simple model for expressing parallelism, which will be described below. This is the model you should follow if you want to write your own applications.

### 3.1.1 Programming model

Programming model supported by Limes is the *lightweight threads* model. In such a model, a program consists of several concurrently executing processes which share the address space. This means that they all have the same virtual addresses for common data and that all the writes to a global variable from one thread are visible to all the others. The only thing that is private for a thread is its stack. (In fact, a thread *could* read and write other thread’s stack, but it never does so, of course.) More formally, page tables describing the translation of virtual to physical addresses are the same for all the threads, with no copy-on-write bits set.

In Linux terminology, this is called the **CLONE** model; in SGI-IRIX terminology, it is called the **SPROC** model. They are both opposed to standard UNIX **FORK** model, where each child processes receives a private copy of its parent’s data.

Consider the following piece of code:

```
int global;
void proc () {
    int local;
    local = 3;
    global = 4;
    }
```

If a thread executed this function, his write to *local* would not be visible by others, but his write to *global* would be seen by everybody else.

Having in mind this simple model that threads use, the only question of relevance is how do threads get created, and what primitives they use for synchronization. Communication between threads can either be done explicitly, with sync primitives, or implicitly, through shared data.

### 3.1.2 Macros

There are several macros you should use in a parallel application. They are called **ANL** macros, by the name of the place of origin (Argonne National Lab). Using them will make your application portable to other simulators and, in fact, real multiprocessor machines.

Here is the minimal list of those supported by Limes:

### 3.1.2.1 Macros for thread control

- **CREATE(FunctionName)**: this macro will create another thread, which will begin its execution in parallel with the parent who created him, starting with the *FunctionName* function. When it reaches the end of this function, the thread will cease to exist.
  *FunctionName* is a pointer to a function that uses no arguments and returns no value.
  Example: `for (i=1; i<NumberOfProcessors; ++i) CREATE(SlaveFunction)`
  creates NumberOfProcessors -1 threads (slaves). Thread #0 is the master.
- **WAIT_FOR_END(NumProcs)**: this macro is usually called by the parent at the end of the parallel processing phase. It waits for all the child threads to finish. NumProcs is the number of threads to be awaited (currently ignored - parent waits for all the children).
- **CLOCK(variable)** will put the thread's current execution time (in cycles) to *variable*. Use it to measure the duration of certain phases in your program.

### 3.1.2.2 Macros for shared memory allocation

- **G_MALLOC(size)** allocates *size* bytes in global, shared memory. On an SMP (shared memory multiprocessor) G_MALLOC would be equivalent to standard UNIX *malloc()*. In fact, they are by default equivalent in Limes as well. But you should use G_MALLOC for each dynamic block of data that should be shared, for the writer of the memory simulator may impose specific allocation strategies for data that is known to be shared. For the data that is known to be local, use simple *malloc*.
  G_MALLOC returns void * pointer to the newly allocated block.
- **G_FREE(ptr)** has the same semantics as standard *free( )*.

### 3.1.2.3 Synchronization macros

ANL defines a set of macros that should be used for synchronization. Limes supports all those used by the SPLASH-2 and probably some others.

#### 3.1.2.3.1 Locks

- **LOCKDEC(lockname)** declares a lock. It is typically placed inside a `struct`, but it does not have to be.
- **LOCKINIT(lockname)** initializes the lock to its default value – *free*. Typically done by the master process before the parallel processing begins.
- **LOCK(lockname)** awaits until the lock is free, then sets him to *busy* and returns. Used for entering into critical sections, of course.
- **UNLOCK(lockname)** sets the lock to *free*. Used for leaving critical sections.
- **A*LOCK*** macros are used for dealing with *arrays of locks*. They have two arguments – array name and array size or element number: ALOCKDEC(alockname, size), ALOCKINIT(alockname, size), ALOCK(alockname, element), AUNLOCK(alockname, element).

Here is an example of the program with four threads, each of which increments a variable. Incrementing the contents of a memory location is a non-atomic operation and must be protected by a lock:

```
int counter;
LOCKDEC(counterlock)                /* declaration of the lock */
void SlaveStart();
main() {
   int i;
   counter = 0;
   LOCKINIT(counterlock);           /* lock initialization */
   for (i=1; i<4; ++i)
      CREATE(SlaveStart)            /* create three threads */
   SlaveStart();                    /* and you be the fourth */
   WAIT_FOR_END(3)                  /* make sure everyone has finished */
   printf("Counter=%d\n", counter); /* should be 4 */
   }
void SlaveStart() {
   LOCK(counterlock)                /* enter the critical section */
   ++counter;
   UNLOCK(counterlock)              /* leave the critical section */
}
```

#### 3.1.2.3.2 Barriers

Barriers are special ANL constructs that enable processes to synchronize. The barrier macros are similar to those for locks.

- **BARDEC(barriername)** declares a barrier.
- **BARINIT(barriername)** initializes its elements.
- **BARRIER(barriername, N)** will make the thread to wait at the barrier together with the other threads that are waiting at the same barrier. The moment the Nth thread enters the barrier, all the N threads will be released.

Actually, WAIT_FOR_END macro called by the parent after return from the SlaveFunction is similar to placing a barrier at the end of SlaveFunction.

### 3.1.2.3 Environment macros

ANL proposes several macros to be used for environment setting, like MAIN_ENV, MAIN_INITENV, MAIN_END and EXTERN_ENV. They are not needed in Limes, and are set to null.

### 3.1.2.4 Limes-specific macros

There are several other macros you may find usable when you develop your own applications or simulators.

- **AUG_OFF, AUG_ON** macros turn off/on the instrumentation in the compilation phase. Everything between these two will execute totally without control of the kernel. These macros can be even put outside any procedure (useful if you want whole procedures uninstrumented).
- **DEBUG_EVENTS_ON , OFF** macros turn on/off printing of events generated by the processors (actually, their threads).
- **DEBUG_MEMSIM_ON, OFF** do the same for dumping the internal state of the memory simulator upon each call to it.
- **DEBUG_SOURCE_ON, OFF** will print for each thread every line of C source whose assembler equivalents are beginning to execute.

### 3.1.2.5 Notes on macros

There are two more things that should be noted about macros. First is, that for many of them you do not need to put semicolon (**;**) behind them. Actually, the compiler might complain if you do in certain situations (like “if (something) MACRO(args); else {…}”)

The second is that the macros are preprocessed by the compiler, not by the **m4** preprocessors as some other simulators do. ANL actually assumes m4 for preprocessing, and therefore suggests all the source files to have extension **.C** or **.H** instead of **.c** and **.h**. Limes does not use m4 in order to make the compilation process faster and simpler, so if you add a SPLASH-2 application into limes/applications directory and see a dozen of .C and .H files that you need to rename, just execute `sh ../C2c.sh` and the script will rename them for you.

## 3.1.3 Level of instrumentation

Before you compile an application, you have to make the decision on what kind of events that a thread can generate should be instrumented by Limes. There are three possible levels:

- **Level 0**: only synchronization and user defined machine instructions (yes, Limes allows you to have them) will be instrumented. This is particularly useful if you investigate algorithms only and use locks for protecting critical sections. In that case all you care about is the communication-to-computation ratio, and the execution slowdown will be minimal.
- **Level 1** will instrument all that it does in level 0 plus shared reads and writes. This is the default level.

- **Level 2** will instrument everything, including stack frame references, like access to local data and pushes and pops. Beware, they are very frequent on an i486! Use this if you want to simulate the system to a full detail – for example, if you simulate caches etc. On the other hand, by omitting stack references from your simulation (i.e. if you use level 1), you might mimic the behavior of a RISC processor with a lot of general purpose registers without sacrificing accuracy, since the stack rarely exceeds few hundred bytes.

Instruction fetches are not simulated. In fact, you can safely omit the stack references as well, since they have little impact on accuracy if you simulate systems with larger caches (on the order of 10% or even much less).

The instrumentation level (also called the "augmentation level") is specified in the makefile that will be described now.

### 3.1.4 Makefile

You could have noticed that the entire simulation, consisting of the application, the kernel, and the simulator, is linked into a single executable. While this process is fairly complex, Limes provides a clean and simple interface to control the compilation, ensuring further that only modules that have changed are recompiled. A **makefile** is defined *per application*, and an executable is created by running "make".

Your minimal makefile should contain only three lines:

```
APPCS = <list of C files that form your application>
TARGET = <name of the executable>
include $(LIMESDIR)/all.make
```

By running make, you will get ${TARGET} executable for the application linked with the *ideal* memory simulator. Consider the minimal makefile for the *simple* application:

```
APPCS = simple.c
TARGET = SIMPLE
include $(LIMESDIR)/all.make
```

There are also certain flags you can specify in your makefile and certain make commands that enable you to delete intermediate files. They will be listed now, but it is useful to know that it usually suffices to use some of the given makefiles with APPCS redefined.

Note: if your workload is a parallel application written in C++, the `include` line should read `include $(LIMESDIR)/all.`**`C++`**`.make`. Your C++ files must have **.cc** extension in order to be compiled with this makefile.

#### 3.1.4.1 Makefile variables

Variables:

- **APPCS**: defines the list of C modules your application comprises.
- **APPCCFLAGS** defines flags to be passed to the compiler when compiling the application. You will almost without exception put **-O2** here (for maximum optimization). Another very useful flag is **-g**. As you know, it will tell the compiler to produce debug information. This way you can without problem debug your parallel application in **GDB** (the GNU Debugger). In addition, Limes will be aware of the fact that this information exists, and will show the source line a thread currently executes, in case of a crash or at your request. It will also be exploited if DEBUG_SOURCE_ON or DEBUG_EVENTS_ON macros are placed in the application source.
- **APPLDFLAGS** specifies application flags that should be passed to the linker, like **-lm** for including the math lib.
- **AUGFLAGS** enables you to specify the level of augmentation – they will be passed to the augmentation tool. Use **-0** for level 0, **-1** for level 1, **-2** for level 2. If you omit them, level 1 is assumed.
- **SYNC** variable specifies the synchronization policy. It is by default SYNC=realistic, but you can set it to SYNC=abstract or to SYNC=your_implementation_of_synchronization_primitives. This will be explained more elaborately later.
- **TARGET** specifies the name of the executable

- **SIM, SIMHOME** specify the name and the location of the memory simulator, relative to the **limes/simulators** directory. If you omit them, the ideal memory is assumed. If you specify SIM but omit SIMHOME, directory limes/simulators/$(SIM) is assumed. You will typically use SIMHOME if there are several similar simulators in one directory. For example, to link the application with the Berkeley simulator, use SIM=ber, SIMHOME=snoopy. For Dragon, use SIM=dra, SIMHOME=snoopy. (The $(SIM) will be used for running $(SIM).make makefile; see section on writing the memory simulators for details).
- **ALGEVALONLY** enables you to avoid simulating references but still to preserve correct order of reads and writes. So if you put ALGEVALONLY=1 in the application makefile, you will get a simulation that runs several times faster, and is therefore suitable for mere evaluation of parallel algorithms, where hardware is not of your concern (PRAM – perfect RAM – model is assumed).

#### 3.1.4.2 Makefile commands

When you run `make` for the first time, Limes basically does four things:

1) First, it compiles your application. It consists of three sub-phases:
   - asks the compiler to translate your application into assembly files
   - calls the augmentation tool to instrument the assembly files
   - compiles instrumented assembly files and stores the object files into an archive, named $(TARGET).a
2) Calls the kernel makefile that will produce kernel.a archive with kernel object files in it.
3) Calls the simulator's makefile that produces $(SIM).a archive with simulator object files
4) Links all the archives into $(TARGET) executable.

All the .a files are placed in **limes/lib** directory, a sort of temporary directory used for caching. Clearing it sometimes can do only good.

Sometimes it may be useful to know how to delete all these intermediate files. Limes supports the following make commands:

- **make dep**: you should issue this if you want to edit kernel and/or simulator source files. Actually, it is advisable to do it first for each new application you want to link into a simulation.
- **make clean**: cleans all the application and the simulator files
- **make simclean**: cleans the simulator files only. For example, you will have to do make simclean if you want to link the application with the Dragon simulator just after you had finished linking a version of it with the Berkeley simulator. It is not needed when switching between simulators with different SIMHOME variables.
- **make appclean**: cleans all the application's intermediate files (object and assembly)
- **make appsclean**: removes .s files of the application (unaugmented assembly)
- **make appSclean**: removes .S files of the applcation (augmented assembly). Note: you could always type "rm *.S", for example, to remove them. It's easier. But it's also easier to accidentally put a space between the asterisk and the dot, and all your work will go to hell. This is why these silly make commands were introduced.
- **make kerclean**: cleans the kernel intermediate files.
- **make syncclean**: cleans the object files of the synchronization implementation.
- **make libclean** clears the limes/lib directory. This refresh does no harm.
- **make totalclean**: cleans up everything. You should do it every time you suspect something is wrong, and it usually happens if you untar a previous version of your source over the lastly compiled one. Also, if you change SYNC or ALGEVALONLY makefile variables, you should do this, or at least "make appclean" and "make kerclean".

Of course, the advantages of makefiles are fully exploited. If you compile a simulation, and then change something in your application source, only the changed source files of the application will be recompiled, not the other ones nor the kernel or the simulator.

#### 3.1.4.3 A “complex” makefile example

Here is the most complex makefile you can find (for the OCEAN application):

```
APPCS         = main.c jacobcalc.c jacobcalc2.c laplacalc.c linkup.c multi.c \
slave1.c slave2.c subblock.c
APPCCFLAGS    = -O2
AUGFLAGS      = -2
TARGET        = OCEAN
APPLDFLAGS    = -lm
SIM           = ber
SIMHOME       = snoopy
include $(LIMESDIR)/all.make
```

## 3.2 Simulators

Limes package includes several simulators. In architecture studies, they may be used to produce real traces, for example. But they can also be used as a model for writing other simulators. This is especially true if one wants to develop a snoopy cache coherence simulator – it might require only a couple of source files to be altered.

Internal organization of the simulators and the rules to be followed when writing them are described in section 5 of this document.

### 3.2.1 The ideal simulator

This is the default simulator (if no other is specified). It simulates a perfect memory system where each read and write completes in one cycle; and so does the UNLOCK (lock release) operation.

The LOCK operation (or “lock acquire”), however, is a different case: a requested lock may be busy. In that case, the requesting process busy-waits until the lock becomes free, then it immediately locks it again and returns.

The Limes kernel can be told to optimize the behavior of locking (and it is optimized indeed by default): if, at certain point of time, Limes detects that all the requested locks were busy, and there were no other events, it will move the global execution time to the earliest moment in future where “something can happen” and thus avoid to repeatedly check for busy locks that can’t be freed before a certain event happens. If the instrumentation level is 0, this can greatly increase the simulation speed.

The “codename” for this simulator is “**ideal**”. It resides in **limes/simulators/ideal** directory.

### 3.2.2 The four snoopy protocols

The package includes simulators for four snoopy cache coherence protocols that were used in performance evaluation for [ToM92]. These are:

- **Berkeley**: a write-invalidate snoopy protocol. See [Katz85] for description.
- **Dragon**: a write-update snoopy protocol. See [McCr84].
- **WIN** (Word Invalidate) - a snoopy protocol based on the method of partial word invalidation. See [ToM92] for description.
- **WTI** (Write-Through Invalidate): a write-invalidate snoopy protocol. Exhibits the worst performance of all.

They all reside in **limes/simulators/snoopy** directory, and their respective codenames are **ber**, **dra**, **win** and **wti**. You can try some of the SPLASHes with these simulators and observe their relative performance. You can see their default cache sizes when you run the application, but it can be changed (see the top of **limes/simulators/snoopy/cache_memory.h** file).

### 3.2.3 The "color" simulator

This simulator is basically the same as the "ideal" simulator; the only difference is that it is extended to show each data access on the terminal screen, using a different color for each processor, representing each read with a dot and each write with an underscore. Application's heap is mapped onto 23x80 characters on the screen.

This "simulator" is useful for visualizing data access patterns: it might be of help when considering possible improvements for existing coherence protocols. You can see what it looks like by putting SIM=color in the application's makefile and recompiling it.

## 3.3 Compilation

To compile the simulation, just change the directory to the one of the desired application and do the "make". For details refer to subsection 3.1.4.

## 3.4 Running a simulation

To run the simulation, just start the produced executable. Many of the SPLASH-2 files have similar syntax; you can learn about it by issuing "**-h**" switch. Usually **-p<N>** stands for number of processors. Any arguments you specify pertain to the application.

While the simulation is executing, you can press **CTRL-** to see how it is unfolding. It will dump the state of each process on the terminal, and the current source line if "-g" was specified in APPCCFLAGS.
NOTE: if you have piped the output of the simulator, do not press CTRL-\\; it will kill the process on the other side of the pipe (as in "FFT -p4 | tee outfile). Use "kill -12" for state dump in such cases.

Finally, it is useful to know that there are certain command line options that you can pass to the simulation kernel. To do that, after all the arguments for the application put a double dash (**--**) and specify a kernel option. To find out which options are supported, say "-- -h".

Currently there are at least five options you can give:

- **-t**: **produces a trace file**. See Appendix B for details.
- **-s**: **dumps threads' statistics** after the simulation ends: the number of reads and writes (shared and private), locks, barriers, total time, and the percentage of time spent in synchronization. Gathering statistics introduces a slight overhead (simulations last some 5% longer with it than without it).
- **-e**: forces Limes output to stderr instead of stdout, which is the default.
- **-i initialization_file**: this enables you to specify an initialization file other than **limes.ini**. An INI file (as recognized by Limes) is a list of lines in form **ITEM=VALUE**. Typically, cache parameters and things like that are placed in such a file, so that you don't have to recompile each time you want to change a parameter. For example, **limes.ini** (or the file specified with **-i**) may contain the following lines:
  ```
  cache_size=16
  cache_way=8
  ```
  If you start a snoopy simulator (say, Berkeley) with the limes.ini in the same directory where the simulation is, it will read these values and initialize the cache to have 16KB and to be 8-way associative. If there is no INI file, it will use the default of 8KB and 2-way associative. You will learn later how to read the INI files, it's very easy and indispensable for comprehensive simulations.
- **-dITEM=VALUE** allows you to define initialization parameters without creating INI files, or to override them. An example: FFT -m12 -p4 -- -dcache_size=16 -dcache_way=8. As in the INI file, ITEM is always a string without blanks and VALUE is an integer. ITEMs are case sensitive.

# 4. The simulator

This section will describe in much detail how the simulator kernel actually works. Reading this section can be helpful if you intend to develop your own memory simulators, or to extend the possibilities of the kernel. (There is always a lot of things you can add in tools like these.)

## 4.1 Execution-driven simulation

### 4.1.1 The concept

Limes uses the approach of the so-called *Execution-driven simulation*. In this approach, parallel application executes on the host processor, executing its native machine code, and special call-outs are inserted into the original code to instrument the events that the application generates. Events are scheduled as requests to the memory simulator at proper times. (We'll see later what does "proper" mean in this case). The name of this approach depicts the fact that the events that drive the memory simulator are generated by executing the application.

There are other approaches. One of them is *Trace-driven simulation*. In this approach, the simulator is driven by the traces generated by some other tools. However, while it can be suitable for simulation of single-processor machines, due to the non-deterministic nature of parallel processing this method does not guarantee accuracy: parallel traces generated for one type of the memory system may never happen on another. On the other hand, this method is fast; further, in some cases such inaccuracy can perhaps be neglected. The bad side is that good traces are many megabytes long. You can use Limes to produce traces, too.

Another method is to interpret the application's machine code at run time instead of executing it. This method can have its good sides as well: for example, if you want to simulate memory to the maximum detail, a thread in execution-driven simulator will barely manage to perform one or two non-memory operation before it returns to the kernel. The overhead for saving registers and context switching may be close to the one for interpreting machine instructions. Still, if you do not need such detailed simulation, you will waste a lot of time interpreting machine instructions that make no global effect. Besides, even if it is suitable for a RISC, there is no acceptable way to do it for a CISC like i486 is.

The following subsections will further describe the execution-driven simulation method.

### 4.1.2 Event handling

The machine code of the original application contains ordinary machine instructions intertwined with certain machine instructions whose execution can have a global effect. An obvious example is the UNLOCK statement: it may release a thread that have been busy-waiting for the lock. Of course, shared writes also count: while not all the data in the formally shared data segment is truly shared, portions of it certainly are. Further, if you want to simulate small caches for example, *every* memory reference can have global effect: a private read may cause one cache to replace a cache line, thus holding the bus and stalling other processors that need access to it. (With large caches, private references have little effect). A counter-example would be reading or writing a CPU register: it cannot affect any other processor. (We are talking about general-purpose or floating-point registers, of course.)

It is easy to conclude that a program can be let to execute without interference the sequence (in a *dynamic* sense) of machine instructions that produce no global consequences, until a machine instruction is encountered whose execution can affect other processes. At that point, the process will enter the simulation kernel mode, *before* the operation is performed. It will leave the kernel again some time later *after* the operation is performed by the simulator. If we call that operation an *event*, we can say that the description of the event will be passed to the simulator by the kernel, and call the whole activity *event handling*.

One important issue deserves attention here: knowing that the simulators executes applications compiled for an i486 CPU or higher, the question is whether the simulation results are bound to the behavior of the Intel family of processors, or it can be used for simulating some future interconnected RISC processors too? It is hardly to give the exact answer, but here is an example:

One real SPLASH-2 application (FFT) was executed on a MIPS machine (DEC station) and one on a PC-486. In both cases, the same application parameters were used: 65536 complex doubles (a realistic problem size that is suitable even for target systems with 1MB caches), single processor, result testing etc. The first simulation was conducted using the **TangoLite** simulation tool, and the second using **Limes**.

- the MIPS generated **18.393 millions of shared reads** and **12.908 millions of shared writes** during the execution of the application
- the i486 generated **18.552 millions of shared reads** and **12.782 millions of shared writes** during the execution of the application

These results indicate that, for a real application, a RISC such as MIPS generates approximately the same number of shared reads and writes as the i486/P5/P6 does. CISCs certainly generate more private references, but these references are generally of little importance to multiprocessor studies; shared references is what determines the behavior and performance of multiprocessor systems.

(As for the time spent for the simulations, Limes shows comparable performance: the simulation with TangoLite, which simulated a pretty complex memory system, lasted 4.57 hours on the (old) DEC station, and the simulation with Limes, simulating caches with the Berkeley protocol, lasted 8 minutes on the 486/133 PC. Simulation speed nearly doubles on a P5/133.)

### 4.1.3 Measuring the time

The time is, of course, the key point in simulation. It is necessary to determine how much time a thread had spent executing non-globally-resulting instructions, and how much time did the event take to be processed.

The basic question is the one of **time units**. The only reasonable choice is **the number of processor cycles**. But how do we compute them? The answer is simple: assume that each instruction lasts only one cycle (by default), and for those that have global effect, add the time (again in cycles) needed for the operation to be performed.

The justification for this approach can be found in the fact that the CPU, host or target, will have pipelined instruction processing in any case.

Actually, this is a rather sensitive issue, or isn't, depending on the level of desired accuracy. In compile time, each instruction is associated with a number that represents the number of cycles it takes to execute before the execution of the next instruction begins. For most instructions, this number is **one**. It means that such instructions are effectively executed one per cycle. If all the instructions lasted only one cycle, that would mean that the target processor implemented an ideal pipeline. (Of course, execution times depend on the program flow, but on the average even fixed numbers can successfully reflect the real behavior.)

By changing these values, you can model a faster or a slower processor. Certain experiments indicate that assigning a one-cycle delay for each instruction, even for the complex floating-point ones, results in simulating processors that are some 30% faster than the Pentium Pro. By increasing these numbers, a P5 or i486 could be modeled. However, the author's opinion is that the next generation processors should be simulated, for they will be connected together in a multiprocessor system, not the old i486s. If you *decrease* these assigned numbers, you can model a processor with a superpipeline, capable of executing multiple instructions in parallel. Check **limes/aug/augment.c** for current values. It is nonetheless possible that some FPU instructions are defined to delay the pipeline for more than one cycle – like multiply/divide (FPU and non-FPU).

The model further assumes that all the processors are driven by the same clock. For an SMP (shared-memory multiprocessors) this can be OK, for a DSM (distributed shared memory systems) perhaps not quite, but it complies with the fact that all the processors will be driven by the same clock *period*, if not the same clock *signal*. Since it is the smallest unit in the system, incongruity between this signals can not introduce any significant distortion.

The memory simulator writer can go one step beyond, and assume that all the modules in the simulated system are driven by the same clock signal, for the sake of the simplicity. But the simplicity need not be hurt if we assume that the bus, for example, is driven by the clock that drives the processor, but divided with two. So the only assumption that exists about the clocks in the system is that they are all synchronized, and that clocks that drive modules other than processors are multiples of the processor clock.

The second question is how do we count these clock ticks? As of memory operations, it's up to the simulator: it can either return some average times or simulate the system modules in a VHDL manner. The instructions, on the other hand, can be counted at compile time, or at run time, via a *trap* mechanism, for example. The first method is the one selected and will be described later in this section.

### 4.1.4 Multithreading

It has been said earlier that the simulation kernel behaves to a certain extent like the operating system kernel. The whole simulation executes in the context of a single UNIX process - the N threads, the kernel, and the memory simulator. A thread executes ordinary instructions in an equivalent of **user** mode, and when an instruction with global effect is encountered, it enters the simulation **kernel** mode. In the kernel mode scheduling and request preparing operations are performed, and so are the simulator operations.

Therefore there is a state associate with each thread; threads that are *ready* are scheduled for execution etc. The kernel maintains a separate stack for each thread, and performs **context switching** when necessary. This comprises of saving and restoring of general purpose registers, the floating point unit state, and the stack pointer.

Current version of Limes saves the complete machine state upon a context switch. It might be optimized in the future to restore and save only those registers that are used in user mode. Such analysis can be performed at compile time, though it's not very likely that it would yield any significant speed-up while it's very likely that it would introduce a lot of complications. Intel x86 processors have pushal instruction that pushes all the general-purpose registers on the stack at once.

Following the operating system kernel analogy, the requests generated by threads can be viewed as system calls. These "system calls" can, in fact, be made at a source language level, not necessarily only the machine level. One example is via the use of certain macros. Another example is calling certain standard library functions. These will be covered in detail later.

### 4.1.5 Scheduling

There are two types of scheduling operations in the kernel. The first is scheduling a thread's request for simulation, and another is scheduling for execution a thread whose request has just been processed.

Scheduling for simulation means preparing certain set of requests that happen at the same point of time and calling the simulator. The simulator processes these requests in a manner that will be described in section 5, and returns, indicating the status of each request, which gives the kernel an information which threads, if any, can be scheduled for execution. (The simulator is called repeatedly, introducing new requests if necessary, until at least one of the request completes, i.e. at least one of the requesting threads becomes ready for execution.)

Scheduling a thread for execution is already described: his context is restored (registers and stack) and the control jumps to the point where it left off. Upon entering the kernel mode again, the thread's context is saved.

The issue in this subsection is the scheduling *order*. For the proper order to be kept, the kernel must find the thread (or threads) with the earliest execution time and schedule them. (If there are ready threads among those with the earliest time, the kernel schedules one of them for execution; if there are not, *all* their requests are scheduled for simulation).

This logic can be approved by observing the fact that none of the global operations is performed by thread itself; it enters the kernel mode before the operation is performed, and leaves it afterwards. The kernel records the time at which the operation should happen, and performs it in cooperation with the simulator.

## 4.2 The compilation process

The simulation kernel and the simulator are statically linked with the application. The process can be pictured this way:

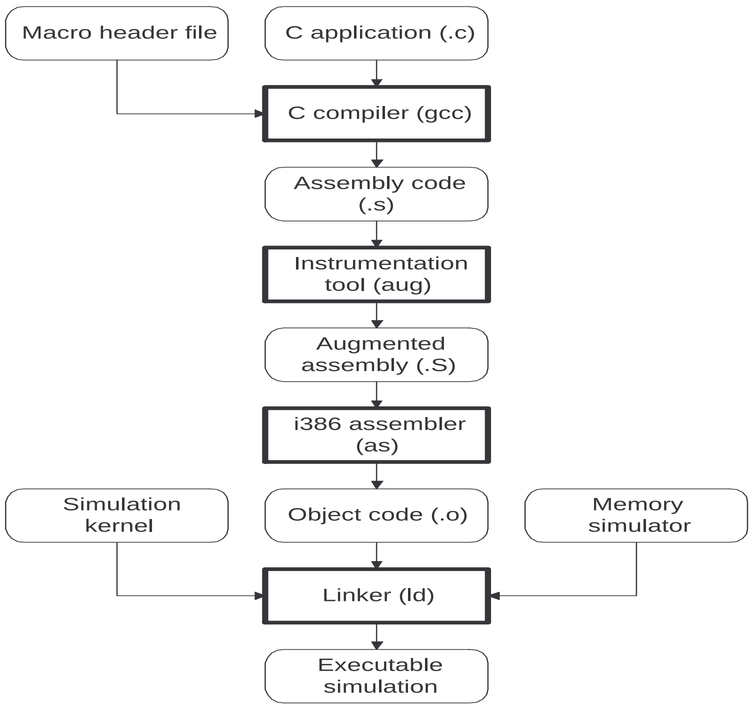


**Figure 3:** The compilation process

As described above, the compilation is controlled via a set of make files, with the ones on the application side being the simplest. Please refer to section 3.1.4 for details on make files and make variables.

The purpose of the macro header file is to make the abstraction of the programming interface. Refer to 3.1.2 for description of macros.

## 4.3 i386/387+ assembly code instrumentation

Much of the information presented in earlier sections relates to almost any execution-driven simulator. The main difference between the existing simulators and this one lies in exactly this phase; the available simulators are written mainly for RISC machines whose assembly code is straightforward for analysis and instrumentation. Complex i386 instruction set required somewhat more complicated and sophisticated approach, and it will be described in this section. Nevertheless, one might find surprising that the program that performs this instrumentation is relatively small, though tricky.

The tool used for instrumentation, **aug**, is frequently referred to here as the *augmentation* tool, since it produces assembly code that may even be up to ten times larger than the original (in the worst case). Fortunately, this is not an obstacle, because instruction fetches are not simulated.

### 4.3.1 What makes it possible

The i486+ CPU has many instructions, frequently of a very non-uniform nature. It would be extremely hard to create instrumentation tool that could handle every situation that can appear, due to the fact that there are so many special cases.

But the analysis of the assembly code produced by the GCC compiler revealed that many instructions and their ugly combinations are never generated by the compiler. There are two reasons for that:

First, there are instructions that are used for system operations only; they can not appear in a user application. Instructions that manipulate with segment registers are a good example.

Second, there are instructions that are perfectly legal in user mode, but the compiler never uses them. Such instructions are created for assembly programmers, not to be used by the compiler, and these are mostly special-cases instructions. This complies with the general philosophy of advantages of RISCs over CISCs, from the view of compiler manufacturers. The overall result is that the compiler never generates instructions as `pushal`, `enter`, and similar, and the trouble-making string instructions are generated in only one of the many possible combinations, and in fact this is the only special case that the kernel needs to take care about.

[For those that are curious, the only combination that appears is REP MOVS with direction flag=0. Actually, with -O2 a SCAS or CMPS can appear too, but only as a replacement for strlen() and strcmp() functions, which are covered by the standard-library functions replacements (see below).]

Floating-point instructions, on the other hand, are easy to instrument, as will be shown.

### 4.3.2 Kernel entry points

Before we proceed, we will once more stress the fact that the whole idea of code instrumentation is to leave uninteresting instructions as they are, and to replace those that produce certain global effects with call-outs to the kernel, where the parameters of these call-outs should be parameters of the instruction. It is thereat desirable to reduce the number of special cases to the minimum. The routines called by the inserted machine instructions are entry points to the kernel. (Routines that implement synchronization are also kernel entry points; more about them later.)

### 4.3.3 RISC vs CISC instrumentation

The main trouble in instrumentation of CISC code seemingly lies in its number of addressing modes. Consider this piece of a simple RISC code, for example:

```
mov r0, r1                   # r1 <- r0
mov $3, r2                   # r2 <- 3
store r4, (r3)               # Memory(r3) <- r4
```

The first two instructions are not of interest, for they do not operate on memory. The third stores the contents of the register r4 into the memory location whose address is equal to the contents of the r3 register. Thus it would be enough to replace the *store* instruction with a call to the kernel indicating that it is a WRITE, that the address is (r3), and the value is (r4):

```
mov r0, r1                   # unchanged
mov $3, r2                   # unchanged
<call the kernel with type=WRITE, address=r3, value=r4>
```

With a RISC, situation is different. Consider these i386 variants of *store*:

```
movl %ecx, _var              # var <- %ecx
movl $3,(%eax)               # Memory(%eax) <- 3
movl %eax, 4(%edx,%eax,2)    # Memory(%edx + 2*%eax + 4) <- %eax
```

With some arithmetic instructions, the situation looks even worse:

```
cmpl $3,(%eax)
incl (%ebx)
subl %ebx,-8(%ebp,eax)
```

or even

```
pushl 4(%eax)
```

and so on. (Please note that the syntax used is *op source, destination*. This is the syntax used by the GCC).

In all these cases, if instructions were replaced with a call, arguments to the kernel entry point should be pushed on the stack: the type of the operation (READ or WRITE), the size (1-byte, 2-word, 4-long – determined from the suffix), the address, and the value if it is a WRITE. If it is a READ, the value should be returned. This would require the code of the instruction to be sent too, and the kernel should update the thread's registers as well, according to the instruction code. It would become very complex and move dangerously close to instruction interpretation, which could slow the simulation down for thousands of times.

### 4.3.4 i386/387+ instrumentation basics

There are two aspects that help us to effectively overcome the problem of analyzing instruction code and updating the registers. Both will be described soon.

Bear in mind that instruction arguments that reference to memory have certain form: it is either

```
[offset](base register[,offset register[,size]])
```

as in

```
-4(%eax)
```

or

```
[offset]_variable
```

(in a.out format)

A relatively simple scanner generated by FLEX is capable to successfully determine if any of up to two arguments refers to memory. Instructions that do not deal with memory are not augmented (just calculated in the basic block length, see below).

#### 4.3.4.1 Values for READs and WRITEs

The idea is to let the CPU do the operation for us: for a write type, let the original write instruction execute, and then we will collect the result and make the call. For a read, we will make the call and the kernel + simulator combination will decide what the result should be, then we'll place the result to the address that should be read by the CPU, and let it do the original read.

In the first approximation, it would look like this: a write instruction

```
<write addr, value>
```

would be replaced with

```
<write addr, value>
{call the kernel with type=WRITE, addr; it will read value from addr}
```

and a read

```
<read addr>
```

would be replaced with

```
{call the kernel with type=READ, addr; it will write value to addr}
<read addr>
```

In both cases, original write or read instruction would not be included in thread's time measuring; it will be calculated in the time needed for the operation to be performed (that is, at least one cycle).

The read is OK, but the write still misses something: if *addr* is shared, it will write *value* to it before it is time. Therefore, if *addr* is suspected to be shared,

```
<write addr, value>
```

will be replaced with

```
{save the contents of addr to a location known to the kernel}
<write addr, value>
{call the kernel with type=WRITE, addr; it will read value from addr}
```

When the thread enters kernel mode, it will check whether *addr* is shared, and, if so, will restore its old contents (after reading the new one, of course).

What gives the instrumentation tool a clue whether *addr* is shared or not? The fact that if **`%esp`** or **`%ebp`** registers (stack, base pointer) are mentioned in the address, it means that the address refers to local data. Of course, an asm programmer might use these registers to form the address of the global data, but GCC would never. In addition, a stack address might also be formed by the compiler without `%esp` and `%ebp` (for example, when a pointer to a stack variable is sent as an argument to another function, references to it may be done without ESP or EBP), but these cases are rare and are not a mistake but only unnecessary.

To conclude, the main idea is **not to *replace* the instructions that need instrumentation with kernel call-outs, but to insert these call-outs *before* or *after* the instruction, depending on whether the instruction is of type READ or WRITE.**

#### 4.3.4.2 Addresses for READs and WRITEs

We have seen that with the values we deal implicitly – we let the CPU calculate them and we just collect them (write) or give them to the CPU (read).

Another question is: how do we supply the kernel with the address of the memory location? Fortunately, there is a simple answer for that, too: i386 honors with **LEA** instruction, which loads effective address into a register. Since the analysis is static (at compile time), it suffices to LEA the argument that we know pertains to the memory into a register, and to push that register on stack.

Here is an example: for

```
subl %eax, -4(%ebp, %ebx)
```

we just produce

```
...
leal -4(%ebp, %ebx), %edi
pushl %edi
...
```

and the address is on the stack, ready for the kernel to collect it.

### 4.3.5 Instruction types

The instrumentation tool has built-in “knowledge” whether the instruction is of type READ or WRITE or both. We’ll cover them briefly now.

#### 4.3.5.1 MOV instruction

A `mov` instruction is either a read or a write, depending on the type of source and destination operands. If the source operand refers to memory, it’s a read, and if the destination operand refers to memory, it’s a write. The kernel is supplied with the so determined operation type, and the address.

#### 4.3.5.2 Plain READ and WRITE types

Some instructions are of type read, regardless of which of their operands refers to memory. An example is `cmp`. A few math instructions are also *read*s – like `fadd`. The call to the kernel occurs before them.

Some other instructions are write only - `set`, `fstp` etc. The kernel is called after they are executed. Before them the augmentation tool inserts old-value preserving sequence, if the address can be shared.

#### 4.3.5.3 Arithmetic instructions

Particularly interesting are the instructions that can be both read and write.

Consider first the `add` instruction, for example: if its source operand is a memory address, then it is a read:

```
addl (%eax),%ebx        # %ebx <- %ebx + Memory(%eax)
```

will be replaced with

```
{call the kernel with type=READ, addr='(%eax)'}
addl (%eax),%ebx
```

The kernel will prepare the proper data in location pointed to by the EAX register before the actual `add` begins. The CPU will then read its contents and add it to EBX.

But if a destination operand refers to memory, then it’s both READ and WRITE (in this order). For example, the instruction

```
    addl $3,(%eax)         # Memory(%eax) <- Memory(%eax) + 3
```

which adds 3 to the location pointed to by EAX, will be replaced with

```
    {call the kernel with type=READ, addr='(%eax)'}
    addl $3,(%eax)
    {call the kernel with type=WRITE, addr='(%eax)'}
```

Someone might object that it is a lot of overhead for a simple `add` instruction. It is true, but this allows the simulator to be special-case-free. Besides, when the context switching becomes optimized, only the registers used in such instructions will be restored before returning to user mode to execute this single instruction, and saved before entering the kernel mode again. Moreover, statistics shows that these combinations are not very frequent.

### 4.3.6 Stack references

For detailed simulations, stack operations may be included as well; this consists of `push` and `pop`. The first is a write, second is a read. The value does not matter (for the simulation), the address is the value of `%esp` at the moment of the operation.

A special case is pushing or popping a memory location; an example is `pushl (%eax)`. That will be instrumented as a read (to read the contents of `(%eax)`) and a write (on the stack).

Local data on the stack is referenced like any other, except that the addresses are `%ebp` relative. If instrumenting stack references is not a goal, the instrumentation tool makes no replacement for such instructions.

### 4.3.7 Special case: MOVS

As stated before, the only exception from these rules is the `movs` instruction. The compiler generates it for copying one long structure to another, and it is always in the form

```
set %ecx; set %edi; set %esi; cld; rep movsl
```

`rep movs` is replaced with a kernel call with ECX, EDI and ESI as parameters (count, destination, source), and the element size (determined from the suffix). Forward direction is always assumed.

When the instrumentation tool encounters `rep movs*`, it replaces it with a call to the kernel entry point that deals specifically with string operations.

### 4.3.8 User-defined instructions

The user can produce instructions that do not exist on the host processor. They are inserted with C macros into original assembly. The GNU C compiler allows that with **asm("…")** directive.

Here is the example for LOCK macro:

```
#define LOCK(L)   asm("lock %0" : "=m" (L) );
```

So if the user issues LOCK(Global->Data.lock) in her C source , the compiler will produce instructions that compute the address for Global->Data.lock and will finally insert `lock <computed address>` into the assembly code, probably in the form like `lock 16(%ecx)`. The instrumentation tool should eliminate such instructions and replace them with appropriate kernel calls.

The instrumentation tool is, of course, opened for user additions.

### 4.3.9 Time counting

Time is counted through accumulating times needed for execution of basic blocks. At the end of each basic block there is a couple of machine instructions that do this accumulation. The estimated execution time for each basic block is computed during the augmentation phase. In most cases, since ordinary instructions by default delay the pipeline for only one cycle, the time for a basic block will be equal to the number of instructions in it.

The definition of a basic block is only extended so that the basic block can end with an instruction that requires instrumentation.

### 4.3.10 Saving and restoring machine status

Before a call to the kernel is made, all the registers and flags are pushed on the stack (with `pushal` and `pushfl` instructions). Upon return, they are restored. Within the kernel, FPU state is stored and restored with `fnsave` and `frstore` instructions. This behavior can be optimized by the register usage analysis, which will probably be done in the future.

### 4.3.11 Debug information

One more simple facility the augmentation tool has can be invaluable for debugging. If the code is compiled with the **-g** switch, the compiler produces debug information that the augmentation tool can understand. For every set of machine instructions that apply to a line in the C code, **aug** inserts a call to the kernel to record the line number. If the application crashes, the kernel will print the source line the offending thread was executing when the crash occurred.

Finally it is worthy to know that all the augmentation does not prevent the application to be debugged by **GDB**, the GNU debugger, if the debug info is produced, of course.

### 4.3.12 A basic block instrumentation example

The following basic block (in GNU assembler syntax):

```
L1:    movl $3, %eax                 # EAX <- 3
       addl %eax, %ebx               # EBX <- EBX + EAX
       cmpl %ebx, -4(%edx,%ecx,2)    # compare EBX with Memory(EDX+2*ECX-4)
```

will be instrumented as follows:

```
L1:    movl $3, %eax                 # no change
       addl %eax, %ebx               # no change

       pushfl; pushal                # save flags and regs
       addl $2, _sys_thread_time     # update time
       leal -4(%edx,%ecx,2), %edi    # get memory addr
       pushl %edi                    # send it to the kernel
       pushl $4                      # size: 4 bytes (cmpl)
       pushl REQUEST_READ            # type: READ (equ $1)
       call _kernel_memoryop         # call the kernel
       addl $12, %esp                # restore stack pointer
       popal; popfl                  # restore CPU status

       cmpl %ebx, -4(%edx,%ecx,2)    # the original instr.
```

## 4.4 Simulations without simulating references

This strategy is mentioned several times in this document as "algorithm evaluation" only. It is a way to obtain fast simulations without simulating memory references, while preserving their order. That means that a thread never enters "waiting for the memory simulator to tell me that my request is satisfied, which I already know" state, it's always READY, but just waits to be rescheduled when its time comes. For example, a thread can try to commit a READ at cycle 100. It will enter the kernel mode (from the user mode), and the kernel will reschedule some other, earlier thread, if such exists – say that it does, and that it entered kernel mode when it tried to commit a WRITE to the same location while his time was 90. That second thread will be allowed to return to user mode, but his WRITE will be performed immediately before it returns to user mode. Assume now that it enters the kernel mode with his private timer equal to 110. Our thread, that was READY but waiting to continue, will be the earliest with his time of 100, and will return to user mode, where the READ instruction will be the first it will execute. That way it will get the correct value, the one written by the second thread at time 90 (assume that the first thread had left the kernel mode in cycle 80 before he entered the kernel again in cycle 100).

The only difference between this model and the "ideal" memory system is that, in these cases, requests are not sliced into chunks whose size is determined by the architecture of the simulated system. Regardless of whether the READ takes one, two, four, eight or ten bytes, it will be completed immediately, unlike with the ideal memory system, which will by default slice requests longer than four bytes into four-word chunks, so an eight-byte read, for example, takes two cycles. This (ALGEVALONLY=1) model also increases the simulation speed up to four times.

## 4.5 Synchronization primitives

Limes implements ANL synchronization primitives described in 3.1.2.3. However, since synchronization is frequently the target of many researches, an option is left to the user to easily implement his own synchronization primitives, or to extend the existing ones.

Limes supports two synchronization strategies: *realistic* and *abstract*. They can be specified in the application makefile with SYNC=realistic or SYNC=abstract. By default is SYNC=realistic. These implementations are placed in directories **limes/sync/realistic/** and **limes/sync/abstract/**. If you want to implement your own, see the README file in **limes/sync/** directory.

### 4.5.1 "Realistic" synchronization

With this approach, each LOCK and UNLOCK request is passed to the memory simulator. When a macro like **LOCK(lockvar)** is encountered in the application source file, it will be replaced with a kernel call that prepares the request for the memory simulator, stating that it is of type REQUEST_LOCK and what the address of "lockvar" is. The same applies to UNLOCK.

Barriers are implemented through locks: each BARRIER macro is transformed into a series of reads, writes, locks and unlocks that all go to the memory simulator.

This implementation is "realistic" in a sense that no action is performed without knowledge of the memory simulator, unlike the "abstract" model described below. All the synchronization primitives pay their due in terms of simulated cycles. This model is the default since (in the current version of Limes) the number of threads equals the number of processors. Therefore there would be no point in "blocking" a thread (putting it to sleep) when it asks for a lock that is busy, since no one else could take its place on its processor. This model is suitable for realistic simulations of SMP systems, particularly knowing that the SPLASH-2 application use communication primitives extensively and ignoring the synchronization would distort the overall performance picture of a simulated system significantly.

### 4.5.2 "Abstract" synchronization

Still, for the sake of completeness, an "abstract" model is also included, so the user can pick it instead. It is useful as an example how one could implement his own synchronization primitives, and is also useful for algorithm evaluation where hardware is not simulated.

The "abstract" implementation uses the following scheme: if a thread requests a LOCK and it is free, it continues. If the lock is busy, the thread puts itself to sleep by marking its state BLOCKED, and adding his ID in the queue associated with the lock. The kernel will ignore him from that point on, and the memory simulator won't see any requests from thread's processor, until another thread commits an UNLOCK (lock release) for the same lock. It will take the first thread from the queue and set its state to READY, and the kernel will reschedule the newly awaken thread, who will know, upon return from sleep, that the lock is all his. This resembles the `sleep()` and `wakeup()` primitives that exist on UNIX, and in fact their names are `sys_thread_sleep()` (puts the thread to sleep and does a reschedule) and `sys_thread_wakeup(thread_ID)` (sets the thread state to READY and updates its time to "now", but does not reschedule it immediately).

Barriers are implemented in a similar manner, and it all looks somewhat simpler than its "realistic" counterpart. By analyzing the source for these two implementations, you can easily write your own locks, barriers and other. For example, you could write sync primitives to implement C-Linda with Limes.

## 4.6 The standard library

Most C programs call standard library functions. So do the parallel applications. However, it is a very rare occasion for a stdlib function to be called in the code that all the threads execute in parallel. Such code consists mainly of operation over vectors, matrices and numbers. For instance, there is no point of doing `printf` within your parallel processing and then to measure the processing time, for you can never know how long will it take for `printf` to complete.

Still, for the sake of accuracy and correctness, and to allow stdlib functions to be called within the parallel processing phase, these functions should be instrumented as well.

The approach used is the following: certain most frequently used standard library functions are compiled and instrumented just like ordinary application files. To avoid collision with the functions that are called by the kernel and the simulator themselves, instrumented functions are prefixed with **sys_**, and a set of macros is defined for the applications that will redirect the calls to the instrumented functions, instead to call the originals.

For example, one of the modules in the Limes-supplied standard library implements string-to-number conversions. There is a function `sys_atoi()` with the same arguments as the original `atoi()` function, and the application sees the macro `#define atoi(S) sys_atoi(S)`, thereby calling instrumented `sys_atoi()` instead of the original `atoi()`, without the need to modify the application source. `sys_atoi()` is instrumented for time counting and memory referencing at the desired level, which retains accuracy. If the application ever calls `atoi()`, the instrumented module where `sys_atoi()` is implemented will be linked together with the application.

However, not all the standard library functions are implemented. The choice was based on several criteria:

First and most important, many functions never get called in the parallel computation phase. `printf()` is an example.

Second, it should be noted that there exist, from the point of instrumentation, two basic types of standard library functions: ones that never operate on global data and the others which possibly do. The criteria for distinction is very simple: those functions that require a **pointer** as an argument might read or write in the global memory zone. Those that do not, certainly operate only with stack. Functions working with pointers must be instrumented, for they could read or write a global memory without the control of the kernel, which would be incorrect (but probably wouldn't change anything, since such functions are called mainly by the master thread, before any other child is created or active).

Third, for some calls the time cannot be determined: some functions call the UNIX kernel and this is the point where every tracing and instrumentation stops anyway. And many of these do the I/O (`printf`, `scanf`…, though `scanf()` replacement *is* implemented).

Fourth, stack references produced by the functions that are rarely or never called in the parallel phase could be neglected. As a matter of fact, with multiprocessor studies that involve large caches, omitting stack references entirely frequently introduces distortion of the order of 2% or less! The reason is that, on a cache hit, the instruction completes in one cycle – as would be the case if the instruction were not instrumented at all, except for the fact that simulating a cache read does take time.

Fifth, many standard library functions on Linux are not reentrant. This means that if a thread would enter `fprintf`, for example, and stopped in the middle of it, and if another thread entered the same function, `fprintf` would get confused and corrupt its data.

Sixth, some functions like those in the math library actually reduce to only few instructions and always operate on stack. It is different with a RISC, but the i486+ has a powerful floating-point unit, and most basic math operations are performed on the lowest level, using FPU machine instructions that are instrumented anyway.

Having all this in mind, the selected approach was to introduce a replacement for every common standard library function whose argument can be a pointer. These include only few: `*scanf()`, `str*()`, `mem*()`, `sprintf()` and `ato*()`. Further Limes releases may include more, and missing ones can always be added by the user who needs them. For the exact list of instrumented functions, please refer to **limes/sys_stdlib/** directory.

Note that this surely does not mean that the uncovered standard functions cannot be called. It just means that they will be counted as a single instruction instead of counting the time they really take.

As we can see, the same strategy of instrumenting only what can actually happen is used for both stdlib functions and machine instructions.

For other stdlib calls, it would be fair to account for their average times in their basic blocks, though not necessary, due to their low calling frequency. This will be done in the instrumentation phase (in future releases).

Finally, an example: SPLASH-2 applications in their parallel phase mostly call only `pow()` and `sqrt()` of all the standard library functions; in regular stdlib, these functions are in fact just single FPU instructions.

## 4.7 The kernel source files

This final subsection will present the hierarchy of kernel modules, in case that you need to make modifications in them. The code is profusely commented, so it should not be a problem to locate and change what is needed, particularly knowing that the whole kernel is relatively small.

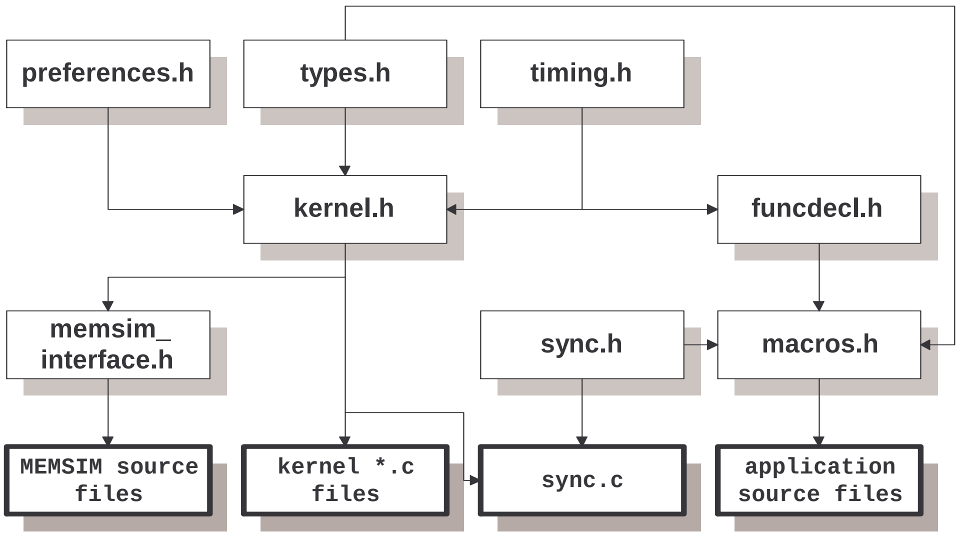


**Figure 4:** Compilation dependencies of the kernel modules

Follows a brief description of each file (description of kernel functions can be found in the Readme file):

- **preferences.h** - defines maximum number of processors (16 by default), lock behavior, default data types and widths (32bit is the default), and some macros for enabling/disabling debugging and statistics collecting.
- **types.h** - defines possible request types (read/write/lock/unlock…) and user-defined instructions. If you want to add your own types, add them here.
- **timing.h** - holds the definition of the timer variable, and some macros for turning the timing on/off.
- **kernel.h** - the main kernel header file: defines memory request structure, thread states, thread structure, etc.
- **kernel.c** - everything relevant is here: startup and cleanup, thread creation and cessation, scheduling and context switching, and some auxiliary functions like `sys_INI_read()` (described later), `sys_thread_sleep()`, `sys_thread_wakeup(thread_ID)` and others.
- **funcdecl.h** - declaration of certain kernel functions, mostly called by the application, but there are useful ones that can be called by the memory simulator as well.

- **interface.c** - the set of kernel entry points, called by the application. These entry points are reserved mainly for memory references, entry points for synchronization are defined in the particular implementation of synchronization primitives.
- **sys_stdlib/*.c,*.h** - redefinition of certain standard library functions and macros to make the application call them instead of the real Linux standard library functions.
- **debug.c** - functions needed for debugging, crash handling etc.
- **memsim_interface.h** - a set of macros that describe kernel-to-memsim interface
- **sync.h** - the definition of ANL synchronization macros
- **sync.c** - implementation of ANL sync macros (LOCK, BARRIER…). These two files are actually placed in limes/sync/ directory and are not really a part of the kernel. Remember, no matter how many files there are in your own implementation of synchronization primitives (should you decide to write one) you must provide the **sync.h** file which is implicitly included by a parallel aplication.
- **macros.h** - the main file included by the application. Contains definitions of other ANL macros.

# 5. Writing a memory simulator

The purpose of this section is to describe the rules for writing a simulator module that have to be followed if the module is to be connected with the parallel application and the Limes kernel, which concerns those who wish to perform architecture evaluation.

It will also present certain hopefully helpful advices for doing it, and expose the details of the snoopy simulators example that comes with Limes. It is highly recommended that the reader be acquainted with the material introduced in previous sections.

The complete simulating conventions and philosophy is presented in the subsections below. In most cases, however, it will be sufficient to add and replace certain modules of the included snoopy simulator with the ones that describe the desired system.

## 5.1 The role of the simulator module

In shortest, the simulator module should imitate, to some extent, the hardware that lies beneath the simulated processors. Actually, the simulator simulates certain parts of the processors as well. Logically, it "begins" at the point where a processor generates memory read/write signals and *virtual* memory addresses, and may include everything from that point on: TLBs, caches, buses, memory modules, networks that connect multiprocessor clusters…

But the level of hardware emulated depends on your need for accuracy. For example, there is no point in investing 30% more efforts in programming the simulator to achieve 1% higher accuracy. The simulator writer, which presumably possesses maximum apprehension of the system it wants to model, will know what details of the hardware are negligible from the overall performance view.

As explained in previous sections, the kernel takes requests from the executing threads, prepares them in a form suitable for the simulator, and initiates it when it's time. Much like the kernel abstracts the memory system from the application, it abstracts the application from the simulator. The simulator sees only an ever ongoing stream of processor requests, without knowledge to what parts of the application they belong. (But of course, since everything here is a program, not the real world, the user can introduce primitives that the application can use to communicate with the simulator directly.)

These requests are in the form of basic READ, WRITE, LOCK and UNLOCK, with appropriate parameters. The user can extend this minimal set of requests with his own, like CACHE_FLUSH for example, modeling in that manner extensions of the instruction set of simulated processors. The simulator must follow certain protocol for indicating the state of requests being processed (actually, it has to tell the kernel whether a request is completed or not), and everything else is up to it.

### 5.1.1 Request scheduling and responding; the FSM approach

The kernel schedules requests; the simulator responds to them, and internally undertakes actions that ensure more or less accurate timing of simulated request processing.

It was explained how the kernel maintains correct global order of events. The simulator does not need to worry about it. It will receive the requests in the same order the real multiprocessor hardware would.

For the sake of convenience, we will repeat the strategy again: every time the kernel decides that something has to be scheduled, a thread or a set of requests, it first finds one or more threads whose current point of time is the minimum of all threads' times. If there is a ready thread among these, it will be scheduled for execution. Some time later, it will enter the kernel again. The procedure will repeat.

When the kernel ascertains that there are no ready threads between those with the minimal time, it will prepare a set of their requests for the simulation. The requests are on **per-processor** basis, not per-thread basis (future Limes extensions will allow migration of threads etc.). At that point, the simulator is called. It receives the list of processor requests **that occur at that point of time**, and the value of the variable that holds the current time (and something else, we'll see later.)

What does the simulator do now? It reads the requests, and begins their simulation. The simulator is allowed to simulate the processing that occurs **only within the current cycle**. Afterwards, it has to return to the kernel. And it has to indicate what happened to those requests, so the kernel can know whether the operation for a requesting thread is completed or not, i.e. whether a thread can be scheduled for execution or not. For example, if you simulate caches, and a read request happens to be a cache hit, it will be performed in the current cycle. If it is a miss, it will not, but the simulator will return before the whole missed read completes.

If you want, you can simulate for more cycles, and then return the kernel the information how many cycles have passed. But it seems that it's much harder to conceive such a simulator than one that operates on a cycle-by-cycle basis.

So there are basically two possible answers to a processor request: it is either **SATISFIED**, in **this** cycle, or it is not, in which case the requesting processor **STALLS**. (With locks, you can have one more response: **LOCK BUSY**. It means that the lock was successfully read, but was found to be busy. This is the default lock semantics, that corresponds to `test&set` or `test&test&set` instructions. Your locks may behave differently.)

The concept of *stalling* a processor until its request finishes allows for accurate, detailed simulation. You might decide that you do not need such level of detail. In that case, you can "immediately" respond to every request and all the requests will complete in a single cycle; SPLASH-2 characterization, for example, was conducted using this approach, while the caches were nevertheless simulated (see the original paper). Otherwise, you will choose an approach that somewhat resembles the VHDL concept, but with far greater flexibility. As we will see, that way you build your simulators to behave like **finite-state-machines**.

Let's introduce few examples that will be referred to in further explanations. We'll assume that we want to accurately simulate processor cache.

The first example is a read that incurs a *read hit*: it can be performed immediately, within the current cycle (of course, that means that the cache is on-chip, with zero wait-states.)

The second example is a read that is a *read miss*. The processor must fetch the missed line from memory or other caches (depending on the protocol), possibly writing back the line beforehand. In addition, it must contend for the bus. Therefore, the number of cycles required for the read to complete is not deterministic.

Let the third example be a write that is a *write miss*. The processor can immediately continue to process subsequent instructions (assume that the write buffer is empty), which in our parlance means that the thread can resume execution, but the processor continues to process the WRITE in parallel, which will eventually be completed after, again, indeterminable number of cycles. It may also have to contend for the bus, to write the line back, to fetch the new line, to actually write, and possibly update or invalidate other processors.

In examples 1 and 3, the simulator will return "*satisfied*" in the same cycle in which the requests logically occur; in example 2 it will return "*processor is stalled* " after the first call.

What happens if a request does not get immediately (in the current simulated cycle) resolved – i.e. if the simulator says that the requesting processor must stall? The kernel will call the simulator again in the next cycle, and again, and again, until the request completes.

You will observe that accurate simulation could be accomplished if the simulator was called after *every* cycle in the simulation, regardless of whether there exist any requests in that cycle or not. The simulator would be built as a finite-state machine, driven by the clock, and it would undertake actions depending on internal states of each of its sub-modules, possibly none if all the previous requests were satisfied, and no new ones appeared. It would also assume that each thread executes one instruction at a time.

This observation is correct. However, it would slow the simulation down significantly, and make the simulator useless for algorithm studies. Therefore, another, equally correct approach is chosen: **the kernel calls the simulator in subsequent simulated cycles as long as there is at least one new request or at least one stalled processor**. The moment all the requests complete *from the threads' view*, the kernel schedules one of them for execution, then the next one, and so on until the ready-thread list is exhausted and the earliest event that should happen is to perform memory operations. It may happen several, or thousands cycles after the previous call to the simulator. Hence, the kernel will call the simulator not only with the list of new requests, but also with the value of current time and the time the simulator was last called in the past. The simulator will then internally simulate every cycle from the last time it was called, to the current time. In that manner, the goal of simulating *every* cycle in the system will be achieved, while the correctness will be preserved. But this allows for two significant speed-ups: first, the threads will be allowed to execute non-global instructions continuously, without expensive entering and leaving the kernel mode, and second, the simulator will be able to optimize its behavior. This means that while looping from the past time to the current time and internally calling itself, it will be able to determine whether each of its sub-modules is in its initial state. If they all are, the simulator will know that nothing can possibly happen before current time – for this is the time at which new requests appeared – and will be able to safely skip all these idle cycles up to the current time and to start processing requests occurred at current time, thereby saving a lot of unnecessary checks and increasing the simulation speed.

Don't get anxious about this seemingly complex process: not only that it is in fact simple, but the simulator module that handles all this, which is developed for the snoopy simulators, can be used without change for every memory system you intend to simulate; you will need to concentrate only on finite-state-machine representation of your simulator. This story is, in fact, only a justification of the simulation logic.

Let us show what would happen for each of our three examples:

Example 1 (read hit): assume that the request occurs at time t=1000. The simulator reads the request, and gives its description as the input to the cache module. The cache module checks its line tags and states, and determines that the line is hit. It updates the LRU information for the set, and indicates that the read is over. The simulator collects the cache module information, responds that the request is satisfied, and returns to the kernel. All in one cycle.

Example 2 (read miss): the same sequence of operations occur as in the example 1, until the point where the cache module determines that the line is not in the cache. It puts itself in the state "requesting a bus", and returns. The simulator indicates that the processor is stalled, and returns. The kernel calls him immediately again, saying that the time-now is t=1001. It calls the cache module, which, assume, has got the bus. It puts itself in state "fetching the line from memory" (which will probably consist of sub-states, depending on the simulated memory latency), outputs the address on the simulated bus, etc. The simulator returns, indicating that the processor is still stalled. The kernel calls him again at t=1002 (doing nothing else in between), and so on. Perhaps at t = 1015 the cache module will say that the read is over, the simulator will indicate that the request is satisfied, and return. The kernel will now mark the thread ready, with its current time t=1016, and schedule him for execution.

Example 3 (write miss): assume that the write happens at t=1000. The cache module responds that the write request is acknowledged, and that the processor may go on. Internally, it will decide that the line is not there, and put itself to state "waiting for the bus", but it does not have to stall the processor since it does not need any further cooperation from it in order to accomplish the write. The simulator will indicate that the request is satisfied, though formally isn't, and return. The thread will be scheduled for execution with time t=1001. Assume that it tries to commit a read 49 instructions later: that read will carry a time stamp t=1050. Now the kernel will call the simulator, indicating that the time from which the simulator should proceed is 1001, and the current time, at which the request occurred, is 1050. So the simulator will loop from 1001 to 1050. It will call the cache module; the module has been granted the bus. The cache module initiates the line fetch. The simulator calls him in the next cycle (1002), and so on. The write will get completely finished at, say, t=1012. At that point, just like every time before, the simulator will check the state of all its modules. It will see that they are all idle (in initial state). Knowing that the new requests, whichever they are, occur at the time named "current time", i.e. t=1050, it will skip all the cycles from 1013 to 1049, and, at t=1050, initiate the read as in examples 1 and 2. When it returns, the global simulation time will be 1051, regardless of whether the new read request was a hit or a miss.

Finally, let us mention one more benefit of this approach, other than accuracy: you will see that it suffices to draw the structure of the simulated hardware on the paper, and to write code for each module in a straightforward manner: you don't have to think hard to conceive the organization of your simulator, just to describe each module as a black box which has its inputs and its outputs.

### 5.1.2 Timing and statistics

We have seen that the total simulation time depends not only on plain number of executed instructions in the program, but also on the amount of cycles needed for each global operation to be performed. The concept of a *stalled processor* is what yields different execution times for the same applications and different simulated memory systems. Otherwise, if all your instructions complete in one cycle, your application will show the same execution times for all such memory systems.

Frequently, this is enough for architecture evaluation: a parallel application usually reads the current simulation time before it begins the parallel processing phase, and also afterwards, displaying the total time spent. Running it over different simulated systems, you will be able to conclude which one suits *that* application better.

On the other hand, you may be interested only in application characterization: all your memory operations will complete in one cycle, but you will still simulate cache hits and misses, although in a far more relaxed manner. This approach is also far less accurate.

In both cases, you can collect various statistics about internal events in your modules: number of cache misses, invalidations etc.

### 5.1.3 Global memory maintenance

There is one more important issue in the simulation, and that is the one of the global memory maintenance. We have seen that the kernel ensures that no write actually happens before it's time; and the value that the CPU will read will be in real memory immediately before the thread resumes execution. That way the consistent picture of global memory is maintained, since when a thread resumes execution, it is the first possible event in global order.

But there is the question *who* supplies the CPU with the value to be read, and where the value that is the result of a write goes. Is it the kernel or the simulator?

The answer is that it can be either one, but is by default the kernel. This negotiation occurs at the simulator initialization phase. If the simulator writer decides that the kernel should do that task, he has relieved himself of a very painful task, without losing much accuracy. All he has to do is to make his simulator tell the kernel *when* the request completed, or, more precisely, how long did it take to complete, using the mechanism of stalling. When the simulator indicates that the request is completed, the kernel will do the following:

- If the request is of type WRITE, the kernel will do the write of the original value in the global memory as soon as the request is satisfied. (Recall that the CPU-performed write, though committed by the thread, was undone immediately, while the new value was stored in a buffer.)
- If the request is of type READ, the kernel does not have to do anything. The thread will be scheduled for execution, and the value it will read is the value of the previous WRITE (possibly by another simulated processor) to the same location. If it wants to change the contents of that location, it will not be able to do it before returning to kernel mode. Therefore, if another thread who wanted to read the same location is scheduled next, it will read the same value that the physically previous but logically concurrent thread has read.

But your memory simulator still has to take care itself about the locks: it cannot assume that a lock request was satisfied without reading its value beforehand; it also must write zero to the lock location if the request is of type UNLOCK. Otherwise, if more than one thread tried to read the same lock at the same time, and if your simulator said “ok” to all of them, they would all enter the same critical sections, which would be disastrous. Therefore, your modules will read and write locks themselves, using two simple macro directives.

The described approach has a drawback in the sense that it can be too consistent: if your simulator has bugs, or your proposed coherence protocol is not coherent in all cases, the threads will still work. Such a bug will be reflected in a disagreement from the results that would be obtained if your simulator was correct. Still, this disagreement will probably be very small.

Nevertheless, if you want, you can write your simulator so that it has complete responsibility over the values the threads receive for their reads and the write values they give. In this case, on a read, you will not only respond that the read is satisfied, but also you will have to return the value for that read. But it may require extreme effort to ensure correct work; you will have to simulate cache memory storage as well, so be prepare to have a lot of physical RAM on your host machine.

The snoopy simulators you received with Limes were at first written in that manner. They were crippled later to do only what is necessary when the kernel is assumed to do the memory maintenance. (Check if **snoopy_full.tgz** archive exists in your Limes tree.) The discrepancies between results for simulations with the old and the new version are about 1%, which in no case justifies the extra effort involved. Still, the choice is yours.

## 5.2 Kernel-to-simulator interface

This section will describe the interface between the simulator and the kernel. The simulator itself can be coded in C or C++, though the C++ is the only real choice for complex simulators. The snoopy example is written in C++. Studying this example before writing own simulator is recommended. There is a lot of README files in the **limes/simulators/** directory, and the code is profusely commented so you can easily figure out what it works.

### 5.2.1 Structures and data types

The simulator does not have to worry about thread's internal structures, states, etc. Only few structures are of its concern, so that it can be maximally independent of the kernel and its future revisions and improvements.

The header files that contain these definitions are `types.h`, `kernel.h` and `memsim_interface.h`.

#### 5.2.1.1 Data types

This is an excerpt from `preferences.h`:

```
typedef unsigned char       bits8;
typedef unsigned short int  bits16;
typedef unsigned int        bits32;
typedef unsigned long long  bits64;
typedef bits32   bitsMAX;   /* for 32-bit bus. Use bits64 for 64-bit bus. */
typedef bits32   bitsLOCK;
typedef bits32   bitsADDR;
#define DATA_WIDTH sizeof(bitsMAX)
```

Your simulator should always refer to `bitsMAX, bitsADDR, DATA_WIDTH, bitsLOCK` , for the sake of compatibility and readability.

#### 5.2.1.2 Processor requests

This is defined in `types.h`:

```
enum Memory_Request_Type {  /* possible requests from the memory simulator */
    REQUEST_NONE    = -1,   /* no request */
    REQUEST_WRITE   = 0,    /* memory write request */
    REQUEST_READ    = 1,    /* memory read request */
    REQUEST_LOCK    = 2,    /* request for a lock (test & set) */
    REQUEST_UNLOCK  = 3     /* unlock request (simple clear) */
    /* add your own here, if you have them. */
    };
```

When the simulator checks for request type, it checks whether it is `REQUEST_WRITE`, or `REQUEST_READ` etc.

### 5.2.1.3 Response types

The following two structures are defined in `kernel.h`:

```
enum PE_response {          /* MemSim's PE[ID] response to a request */
    PROCESSOR_NORESPONSE,   /* no response due to no request */
    PROCESSOR_STALLED,      /* processor stalled due to a read miss etc. */
    PROCESSOR_SATISFIED,    /* memory operation completed */
    PROCESSOR_LOCK_BUSY     /* memory reference complete but the lock is busy */
    };
```

The simulator returns for each requesting or stalled processor one of these values. ("PE" stands for "Processing Element".)

### 5.2.1.4 Memory request structure

```
struct Memory_Request {     /* structure describing the request */
    enum Memory_Request_Type type;  /* READ/WRITE/LOCK/UNLOCK/... */
    bits32 virtual_addr;    /* set by the kernel */
    bits32 addr;            /* computed by the memsim, if TLBs are simulated */
    int    pid;             /* process ID (== sys_thread_ID) -- for TLBs */
    int    size;            /* size of one data chunk - 1..MAX_DATA_WIDTH */
    int    isshared;        /* 0 if it is not */
    bitsMAX data;           /* data buffer, for reads and writes */
    int requested_now;      /* this one is set by the scheduler when appropriate
    enum PE_response satisfied; /* and this one is set by the mem. simulator */
    };
```

This is the basic structure for communication between the kernel and the simulator. The simulator can read each request partially, or can read the entire structure, using macros presented below. It also uses macros to respond to each request.

## 5.2.2 Macros

Here are the inevitable macros that are used for communication. They all refer to a *processor*, not a thread.

- **CPU_COUNT** returns number of "live" processors, i.e. processors where a thread executes (or was executing). The simulator will usually loop from 0 to this value -1 and call its modules.
- **MAX_CPU_COUNT** is the maximum number of processors in the system. 16 by default.
- **REQUEST_EXISTS(I)** returns *true* if a request exists on processor **I**. All the following macros use **I** as the processor index. Remember that **I** does not have to be equal to the logical number of the executing thread, if migration is supported.
- **REQUEST_RESPOND(I, RESPONSE)**: RESPONSE is one of the types from `PE_response`.
- **REQUEST_FILL(I, REQUEST_PTR)** reads the entire request structure in the structure whose pointer is REQUEST_PTR. It is better to refer to these structures by address than by value – nothing can change this structure if the processor stalls, so it's safe to use pointer to it.
- **REQUEST_TYPE(I)** returns one of the Memory_Request_Types.
- **REQUEST_VIRTUAL_ADDR(I)** returns the virtual address of the request. Probably all your virtual addresses will be equal to their physical addresses.
- **REQUEST_SIZE(I)** returns integer, from the range 1..DATA_WIDTH. DATA_WIDTH is the default data width on the simulated system (4 bytes by default). If a request asks for wider data, it will be split into DATA_WIDTH sized chunks. (For example, a FPU request which operates on an 8-byte double, will be seen by the simulator as two consecutive 4-bytes long requests. Your simulator will not know that these two `long` reads are in fact one `double` read.)
- **REQUEST_DATA(I)** reads the data the thread wants to WRITE or sets the value to be read. Use it if your simulator maintains memory instead of the kernel.
- **REQUEST_THREAD_ID(I)** returns the logical number (sort of process id) of the thread executing on processor I. Use it for virtual to physical address translation, if you need it.

Here are some auxiliary macros:

- **MEMSIM_READ_LOCK(lock_addr)** returns the lock value. 0 is *free*, 1 is *busy*.

- **MEMSIM_WRITE_LOCK(lock_addr, value)** writes it. For UNLOCK, you will do MEMSIM_WRITE_LOCK(lock_addr, 0). For LOCK, you will first check if it's free, and if it is, write 1 to it.
- **COUT** should be used instead of regular `cout` to redirect the output to the same stream the other simulation output goes. If you prefere `printf`, use **PRINTF**.

The following are functions, not macros, but nevertheless usable:

- **int sys_INI_read(char *item, int default_value)** is a function that your simulator can use to read certain parameters of your choice at startup. It searches the INI file or command line for a string equal to "item", followed by "=", followed by an integer. If there is no INI file, or no such item is defined, sys_INI_read() returns "default_value". For example, the cache memory class, before allocating space for the tags and line states, calls this function as `size_in_KB = sys_INI_read("cache_size", 8)`; If the user places "cache_size=16", for example, in "limes.ini", or invokes the simulation with -- -dcache_size=16, sys_INI_read() will return 16. If not, it will return 8. So the complete cache geometry can be determined at startup. The cache memory class will be described later.
- **void * sys_malloc(size_t ammount)** allocates "ammount" bytes on system heap. This heap is separated from the application heap, to avoid interference (though it does not really have to be). For C++, it is neater to redefine **operator new** instead; in fact, it is done so in the file "memsim.cc" (described later), which can be used with any simulator. So all your "new"s will allocate space on the system heap, but you can also call `sys_malloc()` explicitly if you want to.

### 5.2.3 Mandatory memsim functions

Your memory simulator must provide the simulation kernel with a few functions, some of which may be empty. The file "memsim.cc" in the snoopy directory, or "memsim.c" in the "ideal" simulator directory, is an example of how these functions should be implemented. Note: if your memsim is written in C++, do not forget to give the linkage specification for these functions as `extern "C"`.

The functions are listed here for completeness; it is very likely that you won't have to change them, and that you will use the whole "memsim.cc" file as it is.

- **enum Memsim_Memory_Strategy memsim_init()**: should initialize your simulator (this is called by the kernel at initialization time), and return either `Kernel_Maintains_Memory_Consistency` or `Memsim_Maintains_Memory_Consistency`. The meaning should be obvious.
- **void memory_simulate(Thread_Time time_before, Thread_Time time_now)**: the main function, called by the kernel millions of times, at simulation time. Time_before is the last time this function was called + 1, and time_now is current time. So, for this call, the simulation should be performed for cycles from time_before to time_now.
- **void memsim_cleanup()**: called at the end of the simulation. It is just for the memsim to dump some statistics on the output, you don't really have to deallocate objects, since the program will exit soon after it anyway.
- **void memsim_dump_state()**: dumps the state of your simulator on the output. Can be empty.
- **void memsim_debug_toggle(int on)**: called when DEBUG_MEMSIM_ON and DEBUG_MEMSIM_OFF directives are encountered in the application; "on" is 0 or 1. Can be empty.
- **void memsim_fork(int parent_ID, int child_ID)**: called every time a new process is CREATEd, in case that you need that information. Can be empty.
- **Thread_Time initial_thread_time(Thread_Time parent_time, int thread_number)**: by default, a child thread time equals the parent time at moment the child is created. This allows you to modify this default. Cannot be empty, but usually consists of only one line: `return parent_time;`.

### 5.2.4 Developing a DETACHED memory simulator

It is wise to develop and debug the simulator driving it with simple traces that you can put in a text file, and then, when it's finished, to be able to link it with the application and the kernel without changing the source. Limes includes simple (but sufficient) support for such developing.

To do so, write your simulator (huh!) and create a makefile, similar to those the applications use. Run it and it will produce an executable, whose first argument (or stdin) is a test-trace file. The support is in

**limes/simulators**, and the makefile example and the test sample are at **limes/simulators/snoopy** and **limes/simulators/snoopy/tests**. Please refer to them for more information, and consult the README files, which describe how to do it all.

### 5.2.5 The simulator makefile

You should select an alias for your simulator, like **ber** is for Berkeley. It will be specified in **SIM** variable of the application makefile when you want to link your simulator with the application. To create your **$(SIM).make** makefile, simply use the **ber.make** file as a template and replace **CS** with the list of your source files.

## 5.3 The snoopy cache protocols simulator

This section will describe the structure of the four snoopy simulators. The principles presented here will more than likely be valid for other simulators as well. In fact, at least two of its modules are generally usable without change.

It will also describe the approach of splitting the simulators into modules and organizing them as finite-state machines (FSMs).

In addition, it will point out the differences and similarities between the four simulators. If you come to a situation that you need to evaluate several similar protocols, you will try to identify common modules and thus minimize the code.

### 5.3.1 The snoopy protocols

The four protocols comprised by the simulators are WTI (Write Through Invalidate), Berkeley, Dragon, and the Fourth Protocol is WIN (Word Invalidate).

The system modeled with one of these protocol consists of N processors with on-chip (L1) caches, snoopy and processor cache controllers, the bus and the memory module. The bus and the memory are the same for all the systems, while the controllers and cache memory differ. But the cache memory for all the protocols differs only in line states. Everything else – number of sets, lines, line sizes, block replacement policy etc. is the same.

### 5.3.2 Identification of modules in the system

Having described the basic system configuration, we can picture the relations between the modules:

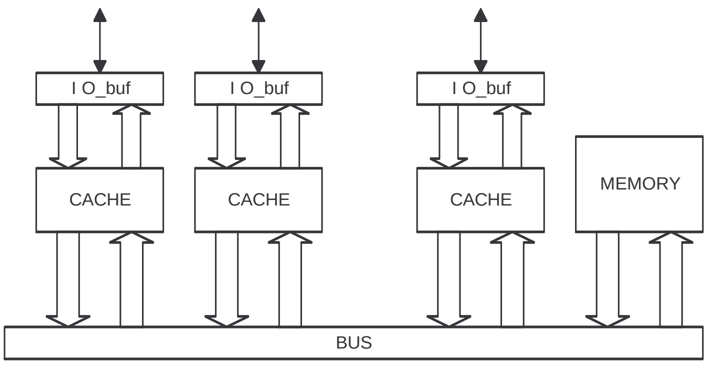


**Figure 5:** Relationship between hardware modules in an SMP system

The basic idea is to model the data flow between modules in such way that every module reads the necessary information from its inputs, performs various operations (possibly changing its state) and leaving computed results on its outputs, without being aware of the other modules in the system. A special module in the simulator (not shown here) is the only module that possesses knowledge about the system topology. It will be called upon each simulated cycle to read the output from all the modules and write it into input ports of neighboring modules.

In short, there are three key points in our simulation model:

- Each module is organized as a **finite-state machine**. A module is called on *per-cycle* basis, and it changes its state depending on what the current state and the current input is, possibly producing output.
- Modules that are isolated from the Limes kernel (in the sense that they do not communicate with it directly) have their **input and output ports**. Upon each cycle, they read the information from their input ports (if there is any) and, according to their internal states, possibly produce information that they leave at their output ports.
- Modules are not aware of the other modules in the system – the whole communication goes through the ports. Only one module in the system knows how the modules are connected, and it follows certain order to read modules' output ports and write the information found therein to their neighbors' input ports.

Here is a brief description of the modules in the picture:

- **IO_buf** module insulates the other modules from the kernel interface. It collects the requests using macros described above and prepares it in the form that the cache module will understand (for example, it can omit the thread_ID and similar information). It also reads the information from the cache module and replies to the Limes kernel.
- **Cache** module reads somewhat transformed processor requests – for example, the addresses it receives are *physical*. If there existed a TLB, it would be placed between the Cache and the IO_buf module. The cache module also reads bus signals (for snooping, sharing detection etc.). Based on these two sets of inputs and its current state, it produces information that goes to the bus and to the IO_buf module.
- **Bus** module keeps the data and the control signals. It does the arbitration; cache modules that want bus access first issue bus_request signal and listen for bus_grant signal.
- **Memory** is the abstract module and in fact not used in the simulator. It would be used if the simulator simulated maintained a separate copy of the global memory that would be a replica of memory in the real system. The knowledge about memory latency (wait states) is built into cache modules, but it could be arranged that the modules and the memory negotiate through special bus lines, like function_completed etc.

## 5.3.3 Topology and data flow

The previous picture described the system topology. It requires some careful planing to connect the modules in such way that all the inputs and outputs can be correctly read and written without the need to implement the complete VHDL approach. One invisible module does this reading and writing.

Data flow should be organized so that signals cannot *loop* among modules within a single cycle.

Based on the fact that the clocks for each module are synchronized, one possible scenario for correct reading and writing signals would be as follows:

1) At the beginning of each cycle, **IO_buf** modules read the requests from the Limes kernel, if they exists. Then they put these requests, perhaps even without any refining, to their output ports. All the N IO_buf modules must be called first to read request from their corresponding processors.
2) Then, all the N **Cache** modules are called. Within each call, a cache module reads its request input port. When it says "I want to read the signals on my request input port", it means "I'll ask the topology module to fill this input port with the signals that appear on the output port of my upper neighbor, whoever that may be. Then I'll read the actual values." (In fact, he does not have to read these input ports if his internal state is such that, for example, the cache is deeply devoted to processing the previous request). The cache module also reads the bus signals, again with help from the Topology

module. Since a synchronous bus is assumed, the signals it will read would in fact be those signals that were on the bus waiting to enter his input port immediately before the rising edge of the clock signal (we'll see why in a moment). Depending on the values of these input signals, the cache module will change its state and produce information that will be put in two of its output ports, one that goes towards the IO_buf module, and the other that goes to the bus.

3) Then the bus module is called. It collects the information from the cache modules and leaves it on its output port. If there were bus requests in this cycle, the bus module will arbitrate and produce bus_grant signal for one of the modules. The description of the bus module deserves a little more attention. If the bus consisted of plain wires, it would be hard to model the caches that communicate through the bus without using the event-driven VHDL approach. For example, cache module #j may produce the information that the cache module #i should read in the current cycle, but since you call the cache modules in order, if i < j the #j-th module will not be called early enough to prepare the information for the #i-th. Therefore the bus is synchronized, or better, the input ports that are connected to the bus are written at the rising edge of the clock. The i-th module can't expect that the j-th module will respond within the same cycle, but will wait for the next. The j-th module will produce information that will be used by the i-th module (for example, whether the line whose address is on the bus is shared) and leave it on the bus. This information will be read by the i-th module at the beginning of the next cycle. This assumption is OK when high-speed systems are simulated, which will always be the case. Moreover, your bus might work with a clock that is a multiple of the processor clock.

4) Finally, the IO_buf modules are called again, only to check if the cache modules responded at the end of this cycle. The response is further transmitted to the Limes kernel.

Therefore, the main simulator loop will first call all the CPU_COUNT (recall that this is the number of "live" processors) IO_buf modules, then the CPU_COUNT cache modules, then the bus, and, at last, all the CPU_COUNT IO_buf modules again (in the first call it is indicated that IO_buf modules should read kernel requests, and in the last call that the modules should read cache responses, if there are any, and reply to the kernel). This loop is performed for every simulated cycle.

As you could see, the approach presented here assumes clock-driven input ports in certain cases; in others it does not. For example, cache inputs that are connected to the IO_buf outputs are not clock driven, and vice versa. This allows read hits to be completed in one cycle (they don't have to be, though): at the beginning of the cycle, the IO_buf reads the request from the Limes kernel; then, cache module takes this request, sees the line is in the cache, and responds positively to the request. At the end of the cycle the IO_buf module reads this response, and in turn responds positively to the Limes kernel.

Further, you can often distinguish between two kinds of signals: ones that have to be written to the module's input ports at every cycle, and the others that don't. For example, the cache may take several cycles to satisfy a read. When it is finished, it will place the results at its to-IO_buf output port, and will also set the special **STROBE** signal at that port to high. The topology module, called by the IO_buf at the end of the cycle, will check the value of this signal, and, if high, will clear it and write the rest of the signals at that port to IO_buf's input port, if it is ready to read it. A module that communicates the readiness of its output signals with the STROBE signal, can check if that bit is cleared in succeeding cycles. If it is, it will know that the neighbor has read the module's output. Hence the usage of STROBE simulates handshaking. (You don't have to use STROBE for a signal that is only 0 or 1 where 0 is the default value. For example, your L2 cache may have "function_complete" output signal that goes to the corresponding "function_complete" input signal in the L1 cache module. So you (the L2) set it to 1 when you are completed in processing L1's request; when the L1 asks the topology to read this signal, the topology will set it to 0 again, and this will be the information for you (the L2) that the L1 has received this signal.)

## 5.3.4 Identification of common modules

You will probably use a distinct C++ class for every group of modules. It is up to your OOP skills to identify common modules in the system for different, yet similar, simulated memory systems, to reduce the amount of necessary code.

For example, it is obvious that IO_buf, memory and even bus module can be common for all the snoopy protocols (actually, signals that appear on the bus are not quite the same, but you can extend a basic bus-signals structure with protocol-specific bus-signals structure or include all the signals that will be needed for all the protocols in one bus-signals structure. The bus module itself is written to be generic (he does not care about the contents of the bus, it used just for arbitration) and you can use it for your simulators.)

But the cache modules are also similar in their cache tags, replacement policy etc. They differ in their processor cache and snoopy cache controller parts. So the cache modules, different for each protocol, can include the same sub-modules for cache lines handling. Here is the picture:

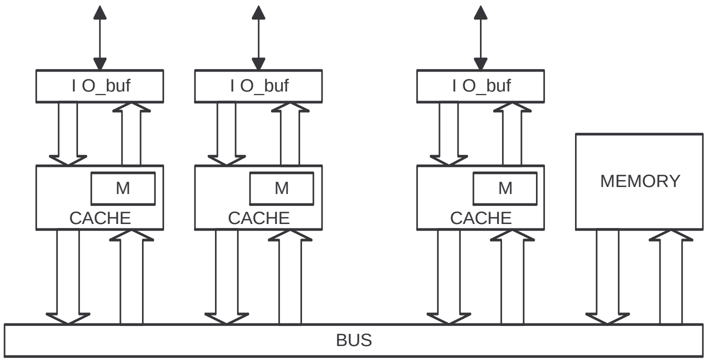


**Figure 6:** Software modules representing their hardware counterparts

The “M” modules are the same for all the snoopies, the rest of the cache module differs. These “M” modules abstract the entire cache tags organization from the simulated controllers.

## 5.3.5 Implementing modules as finite state machines

Some modules can be in different states, some can not. IO_buf is an example: in every cycle, it performs the same sequence of operations.

Bus_module, on the other hand, can have two states: either the bus is granted to someone, or it’s free. In the later case, it has to check if someone is requesting a bus. If the bus is already granted, the module has to check whether the control was relinquished in this cycle, so that the bus can be assigned to someone else in the next.

The most complicated modules may have a number of states. These states do not always have to be explicit, however. For example, a cache module may first be in initial state, and it will check for processor requests. If it determines that the line is not in the cache, it will put itself in the “waiting for the bus” state. This is an example of two explicit states. But a state can also have its substates, as in the example of “waiting for the memory to complete” state. At the beginning, it will set the counter to the number of memory wait states. Upon each cycle, it will simply decrement the counter, and return if it is greater then zero. It will be all that the module will do in the current cycle! Therefore, the state is expressed as explicit state + counter value combination in this case.

The choice of state representation is up to the programmer. However, this section will try to suggest one convenient form of that representation. The following mixture of a flow chart and a state diagram pertains to the WTI cache simulator:

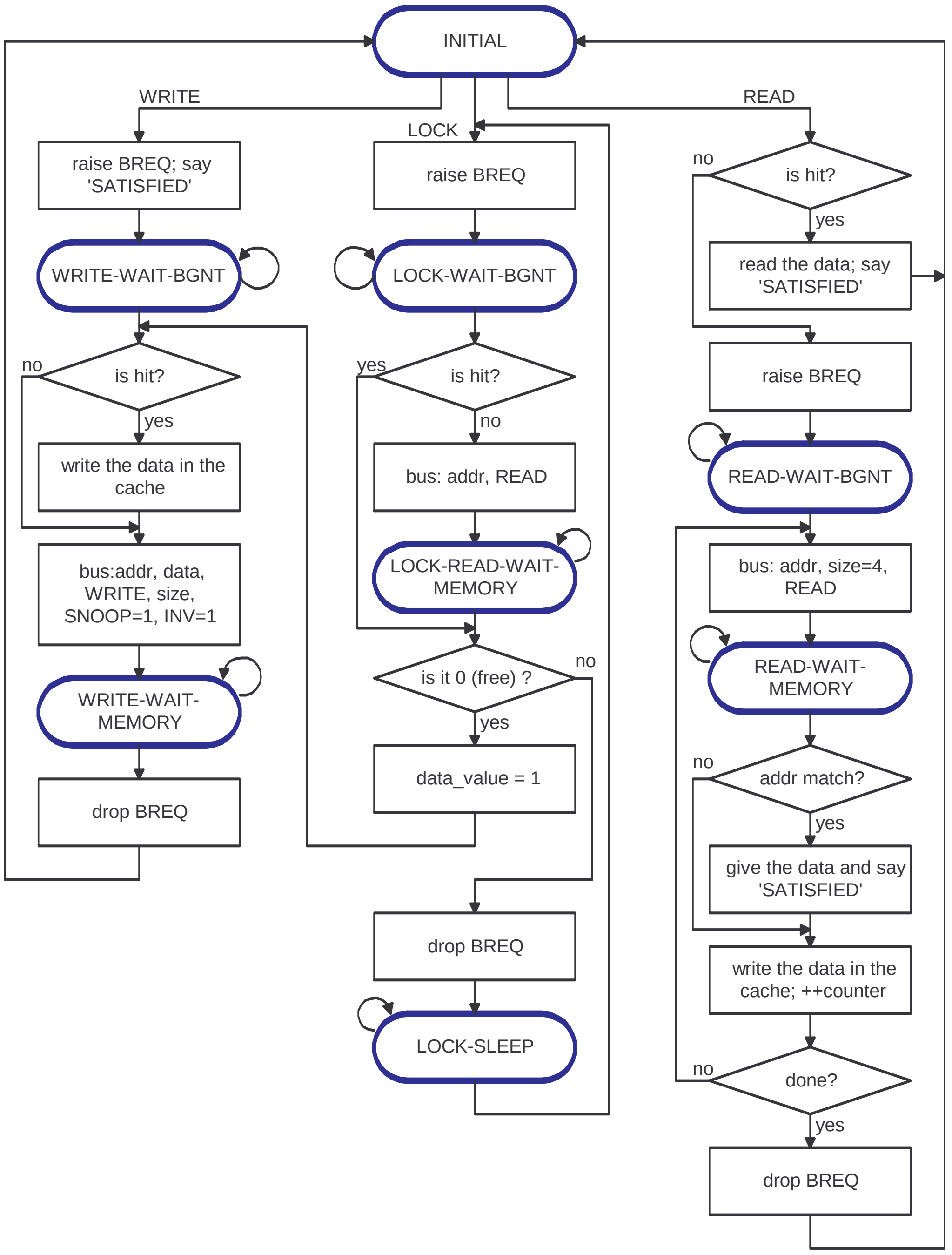


**Figure 7:** State diagram/flow chart for the WTI protocol

This diagram is relatively easy to represent programmatically, and the other snoopy caches aren't much more complicated. The states are represented with thick blue border (or just thick border if you are reading this from paper.) The loops at the waiting states represent waiting for a condition to fulfill; you can determine the nature of the condition by the state name (by the way, BREQ stands for "Bus Request" and BGNT stands for "Bus Grant").

While this diagram is not quite precise, it will hopefully be informative enough. Please compare it with its code equivalent in **limes/snoopy/simulators/wti_cache.h**.

### 5.3.6 The snoopy simulator source

There would be no point in including all the source files in this document. You can better examine them with your text editor. Instead, this sub-section will present the hierarchy of the snoopy simulator source files, so you can follow the correct order in such examination (the code is hopefully well commented). In addition, it will describe the purpose of each module and give some hints to what modules can be used in your simulated system.

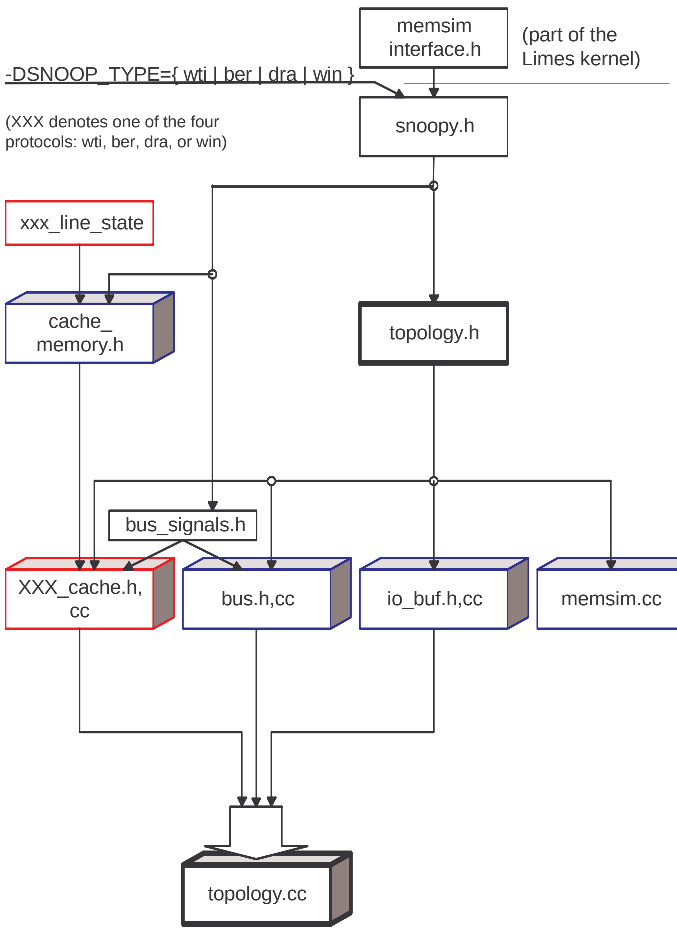


**Figure 8:** Compilation dependencies of the snoopy simulator modules

Red boxes indicate modules that greatly differ from protocol to protocol. Everything else is common for all. Blue boxes are the modules that you can use without change in your system (and therefore ignore much of what was said about the kernel-to-limes interface).

- `memsim_interface.h` was described earlier.
- `snoopy.h` defines which modules apply to which protocols, and certain global constants.
- `xxx_line_state` describes cache line states for each of the protocol. It is defined in xxx_cache.h.
- `cache_memory.h` is a generic module that abstracts protocol-independent information about caches
- `topology.h` is a very important header file that declares functions that each module uses when it needs to perform an environment-dependent operation such as reading its inputs. Again, it contains only *method declarations*.
- `bus_signals.h` defines common signals (addresses, INV line etc.) that exist on the bus.
- `xxx_cache.h,cc` is the module that implements specific protocol in details.
- `bus.h` implements the bus module – arbitration etc.
- `io_buf.h,cc` is the module that communicates with the Limes kernel. Cooked kernel requests are presented to the cache module. This is the module you can use for your simulator.

- `topology.cc` implements servicing functions that all the modules call.
- `memsim.cc` is the main module, called by the kernel at each simulated cycle. You will practically have no need to modify anything in it in order to use it (though you can, of course).

The **snoopy.h** file is included by nearly every module in the system. Besides some global definitions, it embraces the `#include "memsim_interface.h"` in `extern "C" { }` specification. It is suggested that you do the same way (have one common header file that includes Limes header files), and if you have some functions that have to be called from outside of the simulator, do not forget to give their linkage specification as `extern "C"`.

## 5.5 The snoopy simulator details

This section explains in detail how the snoopy simulator is implemented. It is recommended that you examine the source code along with reading this text.

Again, you do not have to follow the model presented here when you write your own simulator; you just have to follow the simple kernel-to-simulator interface. However, the ideas exposed here can probably help you a great deal if you don't know how to organize your simulator.

### 5.5.1 Communicating modules

The following picture shows how the modules communicate. Recall that a module is unaware of any other module; it just reads its input ports and writes something to its output ports. The topology module is in charge for transferring data from one port to another at correct order.

So there are N IO_Buf modules, N WTI_Cache modules, and one Bus module.

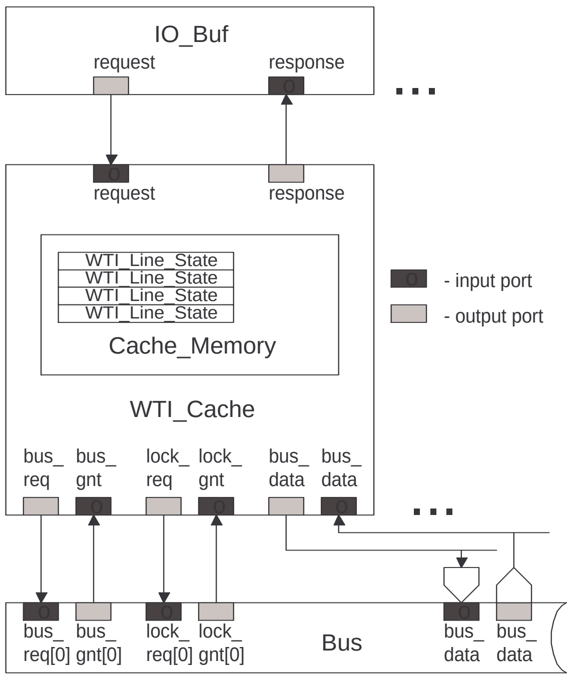


**Figure 9:** Modules and their input/output ports

This example demonstrates the topology for the WTI cache simulator, but it looks the same for any of the other three protocols. Generic module names are defined for each of the four protocols; you will find these names in **snoopy.h**.

### 5.5.2 IO_Buf class

This module is reusable for other simulations. You might want to extend it to support write buffers (and even read buffers), relaxed ordering and things like that.

Output ports:

- **request, STROBE**: "request" contains the original request, as read from the kernel, and the STROBE signal. When the module puts this request into its output port, it sets STROBE to high. When the Topology module reads this port and writes the information into the requesting cache's port, the STROBE is cleared. This gives the module the information that the request has been read.

Input ports:

- **response**: the module asks Topology to read the cache module's response to the pending request at the end of each cycle

Mehods:

- **start_cycle()**: called by the Topology at the beginning of each cycle. It does a **REQUEST_EXISTS(index)** check and, if a request exists, reads it, leaves it on the output port, and sets STROBE high.
- **end_cycle()**: called by the Topology at the end of each cycle. If there is a pending request, it checks if the cache has responded to it. If it has, the module returns the information to the kernel. If not, it tells the kernel that the processor is still to be *stalled*.

### 5.5.3 WTI_Line_State class

Defines the states a cache line in the Write-Through Invalidate protocol can be found in. These values are used by the WTI_Cache class through approriate methods.

States: INVALID,  VALID. Lines for other protocols have more states.

Methods:

- **isvalid()**: returns either True or False. This method is called by the Cache_Memory module to determine whether the line being accessed is valid. For the caches where no INVALID state exists (like the Dragon protocol), this method always returns True.
- **get()**: returns the current state. Indirectly called by the WTI_Cache, through Cache_Memory.
- **set(state)**: sets the new state for the line.

This class is not reusable, of course. But it's a good sample for writing your own state class(es), if you intend to use the Cache_Memory module described below. Here is what this class looks like:

```
// states a line in the WTI protocol can be found in
class WTI_Line_State {
  public:
    enum State { INVALID, VALID } state;
    WTI_Line_State()                {state = INVALID;}
    bool  isvalid()                 {return state != INVALID;}
    State get()                     {return state;}
    void  set(State new_state)      {state = new_state; }
    char  symbol()                  {return state == VALID ? 'V' : 'I';}
    };
```

These are exactly the methods and variables you have to provide:

- **state**: describes the state of the line (WTI is simple, it has only VALID and INVALID)
- **constructor()**: to initialize the line to its default state
- **isvalid()**: returns true if the line is in a valid *state* (the cache will take care of other things, like whether the tag is valid). Some protocols, like Dragon, do not have INVALID state – so this method always returns *true* with them.
- **get()**: returns the state
- **set()**: sets the state
- **symbol()**: returns a character representing the state; it is used for debugging.

You can place additional information and create more methods for your line. These method will not be known to the cache memory module, but you can extend it (inherit from it) to more specific cache memory that suits your particular protocol and deals with this extra information per line. This is how the WIN protocol was implemented; please refer to its source, but please read the description of the cache memory class first.

### 5.5.4 Cache_Memory class

This module encapsulates all the cache properties within itself; It is part of the cache controller module, but it works only with *lines*, without knowing how large a line is, what is the cache capacity, associativity etc. It is reusable for all other systems.

Cache_Memory class is a *generic* class – a *template* whose parameter is a Line_State class, compatible with the one described above. The parameters pertaining to cache geometry can be specified at the initialization phase; these are: cache size (in Kbytes), associativity (2-way or 4-way etc.), and line size.

To make it a part of your cache controller, say something like

```
    Cache_Memory<WTI_Line_State> cache_memory;
```

which will create a specific instance of the Cache_Memory template that deals with WTI_Line_State type of lines. Then you can apply the following methods to each instance:

- **Initialize(int cache_size_in_KB, int line_size_in_bytes, int associativity)**. The parameters are optional; if you do not specify them, the defaults will be used. These parameters can also be other then defaults if you define the following items in your INI file: "**cache_size**", "**line_size**", "**cache_way**". The defaults are 8KB, 32 bytes, 2-way, respectively. If you simulate multilevel caches, here is a good way to initialize the cache memory in L1 and L2 caches: in your L1's cache controller, say

  ```
  cache_memory.Initialize(sys_INI_read("l1_size", 8),
                          sys_INI_read("l1_linesize",32),
                          sys_INI_read("l1_way", 8) );
  ```

  and do similarly for the `cache_memory` variable in your L2 cache controller, but with (presumably) larger default values. This way you don't have to recompile the simulator each time you want to measure the impact of cache size on performance, you can specify the values in an INI file or through command line switches. Call this method first, in your cache controller constructor, before you call any other Cache_Memory method.
- **bool isvalid(address)**: this method is first called when WTI_Cache receives a memory request, or when the snoop controller (part of WTI_Cache) detects SNOOP=high on the bus. The method returns True or False, and the Cache_Memory module marks the line where the address belongs (or should belong) as *current*. Calling other methods implicitly deals with the current line. Even if the line is not in the cache, the module will pick one suitable for replacement, using the LRU algorithm. So, if `isvalid()` returns false, the current line will be the one chosen for replacement, and you can write it back to memory, for example, if it is in one of the *modified* states, and write new data into it, setting it to a new state etc.
- **Line_State getstate()**: returns the state of the *current* line
- **setstate(state)**: sets the state for the *current* line to "state".
- **touch()**: forces the *current* line to be marked as the most recently used
- **untouch()**: the opposite
- **line_fetch_init(&addr) / bool line_fetch_putnext(&addr) / line_write_init(&addr) / bool line_write_getnext(&addr)**: these four functions simulate writing a line to the main memory (or the cache, if a cache supply is performed), or from the memory/cache. First the _init() is called, setting "address" to the address of the first word is the line. The cache controller presumably outputs this address on the bus. When the first ford is written to/from the cache, _*next() method is subsequently called until it returns False. In this way, the information of the number of words in the line is hidden (and irrelevant) from the controller. If this seems unclear, please take a look at the Cache_Memory source and the WTI example of its usage.

- **set_current_frame(int framenumber)**: this method is used for selecting the set of *Current* variables (those that describe the *current* line – position of the line in the cache, its tag, offset of the requested word within the line etc.) that the cache memory will operate with. For example, your processor cache controller part will always use frame#0, and your snoopy cache controller part will always use frame#1. Why is this needed? Suppose the PCC (Processor Cache Controller – part of the cache controller that looks at requests coming from the processor) had called isvalid() and decided to wait for the bus. His *current* vars are set. Now, the SCC (Snoopy Cache Controller – the other part of the controller that monitors the bus) sees an invalidation request, and has to check whether the line exists in the cache. He has to call isvalid() as well. But if there existed only one set of Current variables, this isvalid() call would spoil those Current vars set by the PCC. So the snoopy cache controller always selects frame#1, and the processor cache controller always selects frame#0. This frames do not exist in the real world, they are only used to support encapsulation of the cache memory details from the cache controllers.
- **Introduce()** prints the cache geometry information to standard output. Arrange that your cache controller module calls this method from his own Introduce() method; this is useful if you have many simulations and redirect their output to files that you'll analyze later.
- **dump_state()** prints the information about the valid lines to standard output. This is used for debugging.

The WTI controller module, shown later, will hopefully clarify things further.

## 5.5.5 Bus_Signals class

This simple class defines all the signals that can be on the bus. A cache module has one instance of this class in its input port, and one in its output port. There is a single instance of it in the Bus class. This instance is a copy of the output port instance of the cache controller that is holding the bus.

(Public) elements of this class are those signals that exist on a real bus – address, data, control line, SNOOP line, etc. "bus_data" label on the picture in 5.5.1 implies all these signals.

Analogously to the WTI_Line_State, this simple struct is not reusable, but you can modify it to add the signals that appear on your bus.

## 5.5.6 Bus class

This module comprises of: the bus arbitrator, the bus "wires", and the lock arbitrator.

A module contends for the bus by sending the bus arbitrator BUS_REQ high (actually, setting the BUS_REQ element of its output port to 1), and waits until it receives BUS_GNT from the arbitrator (or, more precisely, until it finds 1 in the BUS_GNT element of its input port). The arbitrator uses the rotating-priority scheme, and it has two states: bus_free and bus_busy.

Again, this is how the concept employed here works: the module that has the bus granted, leaves on its own output port the signal that it wishes to appear on the bus and to be seen by other modules. It leaves it in "*this*" cycle. The bus module is called last within "*this*" cycle, and it asks the Topology to read the data from the output port of the current bus holder, and to copy it into bus module's data placeholder. At the beginning of the *next* cycle, each module will call the Topology to copy the data from the bus to the module's input port.

Input/output ports:

- **bus_contents**: an instance of the Bus_Signals class

Input ports:

- **BUS_REQ[MAX_CPU_COUNT]**: a vector of bus requests coming from the cache modules. At any given time, none, one or more than one of its elements can be nonzero.
- **LOCK_REQ[MAX_CPU_COUNT]**: analogously, a vector of lock requests

Output ports:

- **BUS_GNT[MAX_CPU_COUNT]**: a vector containing the information who owns the bus. At most one of its elements can be nonzero at any given time.
- **LOCK_GNT[MAX_CPU_COUNT**: analogously.

Mehods:

- **cycle()**: this is the only (important) method. It is called by the Topology module near the end of each cycle. It checks the state of the request lines, grants the bus to the chosen requester, and reads the bus-output (by calling Topology) of the module that holds the bus.

The bus module is generic, and thus reusable. It is not interested in the details of Bus_Signals class, so there can be anything.

### 5.5.6.1 Lock arbitration

Something more about the lock arbitration: this system tries to implement efficient locks. A lock, viewed as atomic read+write, must use some sort of hardware binary semaphore: therefore, a processor attempting to acquire a lock could wait to get exclusive access to the bus. Then it could read the lock location, perhaps write to it, and then release the bus. But it would be a waste to keep the bus locked for many cycles during which only busy lock checks are performed. Instead, there is a lock arbitrator, and it serves as a semaphore, while the bus is left for bus transactions alone.

So the scenario for acquiring a lock is as follows:

1) A processor that has a LOCK request sets LOCK_REQ high. It waits to receive LOCK_GNT high from the arbitrator for as many cycles as needed.
2) Then it reads the value from the lock address, as it usually reads it.
3) If the value is 0, it writes 1 in it, as it usually writes it.
4) It drops the LOCK_REQ signal, and the arbitrator drops processor's LOCK_GNT signal in turn.

The sequence is similar for releasing a lock.

This mechanism is consistent, since locks are read and written only through LOCK and UNLOCK (except when they are initialized in the single-process mode), and not through ordinary reads and writes.

This scenario is followed by the Berkeley, Dragon, and the WIN protocol. WTI still uses the bus as a semaphore – WTI is inefficient anyway, and is included here only because it's simple to analyze.

## 5.5.7 WTI_Cache class

This is the heart of the system, of course. Being the most complex module of all, it has input and output ports, and several internal states. Its state diagram / flow chart is shown in the section 5.3.5. Please refer to it again before you continue reading.

Input ports:

- **request**: structure describing the memory request (of type struct Memory_Request) coming from the IO_Buf. In fact, this doesn't have to be IO_Buf; it could also be a TLB, if we inserted such module between the cache module and the IO_Buf module.
- **bus**: an instance of the Bus_Signals class
- **BUS_GNT, LOCK_GNT**: grant values for the bus and the locks

Output ports:

- **response, STROBE**: when a request is completed, the cache sets "response" to PROCESSOR_SATISFIED, and also sets STROBE to high. When the neighboring module reads this response, Topology will clear the STROBE signal, and it will be an acknowledgement for the cache that the response signal has been read.
- **bus**: an instance of the Bus_Signals class. Remember that we have to simulate hardware only functionally, so we can afford to have two groups of signals – one that goes to the bus and one that comes from it. We write values into this output instance, and read values from the input instance.

- **BUS_REQ, LOCK_REQ**: set to high when the cache wants to perform a bus operation, or when it wants to do a "lock acquire" or a "lock release" operation.

### 5.5.7.1 Line states and transitions in WTI (Write-Through Invalidate)

The WTI cache implements a very simple protocol:

- each line in the cache can be either INVALID or VALID
- on a READ, if the line is VALID (read hit), the request is immediately completed
- on a READ, if the line is INVALID (read miss), the line is fetched from the memory and declared VALID. The request is completed after the line is fetched, or, better, after the requested word is read (while the controller continues to read remaining words in the line)
- on a WRITE, the cache sends the address and the data on the bus to be written to memory regardless of the line state (that's why it is "Write-Through") and sets the signals INV and SNOOPY to high. Every other cache that has the copy of the line invalidates it (that goes for "Invalidate"). If the originating cache has this line valid in its own cache memory, it updates it.

The cache controller consists of the processor cache controller and the snoopy cache controller. The processor cache controller takes requests from the processor, and eventually satisfies them. It reads the cache memory, contends for the bus etc. The snoopy cache controller makes his life bitter. It monitors the bus, and if it sees the SNOOP line high and the INV line high, it takes the address from the bus and checks if there is a valid line with the tag that matches the address. If there is, it invalidates the line.

The model assumes duplicate cache tags: one set of tags for the cache controller, one for the snoopy controller. When the snoopy cache operates on cache memory, the processor cache controller must wait.

It is suitable to integrate these two modules in a single class. We could have these modules separated, but it would give the same results (in terms of simulation times) and the whole design would become unnecessarily complex.

The cache module is implemented as a finite state machine, as shown in the diagram in section 5.3.5 (Figure 7). The diagram represents operation of the "Processor Cache Controller" unit. If the WTI cache had a more complex "Snoopy Cache Controller" unit (as the Berkeley has), there would be another diagram for operation of this part. Similarly, if the controller consisted of a Prefetch unit as well, this unit would have its own diagram. These units would be called in certain order, all within one cycle, and communicate through some common variables. For example, the SCC unit and the PCC unit in a cache controller operate in parallel, since they have duplicate cache tags. This parallel operation is simulated by calling the SCC unit and the PCC unit (internally, by the cache controller class) both in one simulated cycle. But the SCC unit has precedence – for example, if it invalidates the data it writes something in the cache memory, and the PCC part can not access it within that cycle. So if the SCC does something with the cache, it will set some flag that indicates that the PCC part is not to be called in the current cycle. Please take a look at the Berkeley implementation if this seems unclear.

The WTI finite state machine has the following states:

- INITIAL - the controller is idle
- READ_WAIT_BGNT - it has received a read request, it was a miss, and is waiting for the bus now
- READ_WAIT_MEMORY - waiting for the memory to output the requested word on the bus
- WRITE_WAIT_BGNT - the controller has received a write request. It must get the bus first
- WRITE_WAIT_MEMORY - waiting for the memory to accept the write
- LOCK_WAIT_BGNT - waiting for the permission to read a lock
- LOCK_READ_WAIT_MEMORY - reading the value from memory (on a read miss)
- LOCK_SLEEP - if the lock is busy, the controller waits few cycles before reattempting to acquire the lock. This is in fact unnecessary, since the Limes kernel is also perfectly capable to re-issue LOCK<addr> if the lock was busy.

When the WTI wants to write/read data to the memory, it waits the memory module to complete for the predefined, fixed number of cycles, regardless of whether it is reading successive addresses or not. Other protocols (Berkeley etc.) use more realistic schemes, suitable for contemporary memory modules.

### 5.5.7.2 WTI_Cache source

```
#include "snoopy.h"
#include TOPOLOGY_HDR
#include "cache_memory.h"
#include "bus_signals.h"

// states a line in the WTI protocol can be found in
class WTI_Line_State {
  public:
    enum State { INVALID, VALID } state;
    WTI_Line_State()                {state = INVALID;}
    bool  isvalid()                 {return state == VALID;}
    State get()                     {return state;}
    void  set(State new_state)      {state = new_state; }
    char  symbol()                  {return state == VALID ? 'V' : 'I';}
    };

// the WTI cache controler module declaration
class WTI_Cache {
    int index;                          // index of the belonging processor
    // instantiate cache memory with WTI_Line_State as its Line_State
    Cache_Memory<WTI_Line_State> cache_memory;
    // port definitions
    struct In {                         // Input port:
        struct Memory_Request request;  // from the CPU
        bool BUS_GNT;                   // bus grant (from the bus module)
        bool LOCK_GNT;                 // ignored
        Bus_Signals bus;                // bus contents, to be read from the bus
        } in;
    struct Out {                        // Output port
        struct Response {               // to the CPU
            enum PE_response signal;    // signal to the IO_Buf:
                                        // PROCESSOR_SATISFIED, _STALLED, etc.
            bool STROBE;                // set to high when 'signal' is formed
            } response;
        bool BUS_REQ;                   // bus request: sent to the bus module
        bool LOCK_REQ;                  // ignored
        Bus_Signals bus;                // bus contents, to go on the bus
        } out;
    // states this finite state machine called WTI can be found in
    enum State {
        INITIAL,                        // INITIAL -- not doing anything
        READ_WAIT_BGNT,                 // waiting for bus to do a READ
        READ_WAIT_MEMORY,               // waiting for the memory to complete
        WRITE_WAIT_BGNT,                // waiting for bus to do a WRITE
        WRITE_WAIT_MEMORY,              // waiting for the memory to complete
        LOCK_WAIT_BGNT,                 // waiting for bus to do a LOCL
        LOCK_READ_WAIT_MEMORY,          // waiting for bus do read lock value
        LOCK_SLEEP                      // sleeping, simulates spin delay
        } state;
    // some internal vars
    bool snoop_hit;                     // true for a snoop hit
    int wait_state_counter;
    const READ_WAIT_STATES              = 2;
    const WRITE_WAIT_STATES             = 2;
    const BUSY_LOCK_SLEEP_CYCLE_COUNT   = 1;
    int busy_lock_sleep_cycle_count;
    // statistics
    struct Stat {
        int req;                        // requests
        int inv;                        // invalidations
        Stat()  { req=inv=0;}
        void dump(int index);
        } stat;
  public:                               // declaration of methods
    WTI_Cache();                        // ctor
    void assign_index(int ind) {        // index is assigned by Topology
        index = ind; }
    void Introduce();                   // called at startup
    void cycle();                       // called at each simulated cycle
    bool isidle() {                     // false if not in INITIAL state
        return (state == INITIAL); }
    void dump_state();                  // for debugging
    void dump_stats() {                 // dump statistics at end
        stat.dump(index); }
    TOPOLOGY_FRIENDS                    // Topology class is WTI's friend
    };

#ifdef PRAGMA_IMPLEMENTATION_WTI_CACHE
// the WTI cache controller module implementation

WTI_Cache::WTI_Cache() {                        // ctor
    index = 0;                                  // default, but will be changed
    cache_memory.Initialize();                  // initialize cache memory
    state = INITIAL;                            // go to the INITIAL state
    out.response.signal = PROCESSOR_NORESPONSE; // clear input and output ports
    out.response.STROBE = 0;
    in.BUS_GNT = 0;
    out.BUS_REQ = 0;
    in.bus.reset();
    out.bus.reset();
    };

void WTI_Cache::Introduce() {
    COUT << "Protocol: WTI (v1.0). ";
    COUT << "\n";
    COUT << "READ_WS=" << READ_WAIT_STATES << ", ";
    COUT << "WRITE_WS=" << WRITE_WAIT_STATES << ", ";
    COUT << "LOCK_WS=" << BUSY_LOCK_SLEEP_CYCLE_COUNT << "\n";
    cache_memory.Introduce();
    }

void WTI_Cache::Stat::dump(int index) {
    if (req) COUT << "WTI"<<index<<": " <<
        "req="<<req << ",inv="<<inv << "\n";
    }

// the main method: Topology calls it in each simulated cycle. If it was not
// in initial state, it will resume from the point where it left off in the
// previous cycle.
// This method first reads the bus. Then it checks if there is SNOOP signal
// high on it, and, if so, checks if it has the line to invalidate. If true,
// does so and skips this cycle. This reflects the fact that there are two
// controllers in this one, the Processor Cache Controller (PCC) and the
// Snoop Cache Controller (SCC), each of which has its own copy of cache
// tags, but the cache memory itself can be accessed only by one controller
// at a time, and the SCC part has precedence.

// A few words about this protocol: WTI means 'write-through, invalidate'.
// That means that each write is write-through, i.e. straight to the memory
// (and also to the cache if we have the line in it), while instructing
// all the other caches to invalidate their own copy of the line, if they
// have one. On a read miss, the whole line will be fetched from the memory.

void WTI_Cache::cycle() {

    Topology::cache_read_bus(index);        // read the bus contents first

    // check if there is SNOOP signal on the bus, but issued by a cache
    // other than myself:

    if (in.bus.SNOOP && !in.BUS_GNT) {    // !in.BUS_GNT: 'I don't have the bus'
        // there is: select frame #1 of cache_memory's C (current) vars,
        // which keep the current position of the line
        cache_memory.set_current_frame(1);
        // check if the address is 'valid' - the line exists, and is not INVALID
        snoop_hit = cache_memory.isvalid(in.bus.addr);
        if (snoop_hit) {
            // if so, force this line INVALID:
            cache_memory.setstate(WTI_Line_State::INVALID);
            cache_memory.untouch();         // force invalid line MRU
            ++stat.inv;          // statistics: increment the invalidation count
            }
        // now restore Current frame #0 for use with the PCC part
        cache_memory.set_current_frame(0);
        if (snoop_hit) return; // if this was invalidate HIT, skip this cycle
        // (actually, it should skip this cycle anyway; try it to see how
        // it affects the performance.)
        }

    // OK, there was no snoop or there was a snoop miss, so we continue.
    // We might or might not be in INITIAL state. If we are, we'll check
    // if there's a request for us from our processor. If we are not, that
    // means we continue from where we stopped in the previous cycle, and
    // we'll use 'goto' to resume.

    switch (state) {
        case INITIAL                : goto STATE_INITIAL;
        case READ_WAIT_BGNT         : goto STATE_READ_WAIT_BGNT;
        case READ_WAIT_MEMORY       : goto STATE_READ_WAIT_MEMORY;
        case WRITE_WAIT_BGNT        : goto STATE_WRITE_WAIT_BGNT;
        case WRITE_WAIT_MEMORY      : goto STATE_WRITE_WAIT_MEMORY;
        case LOCK_WAIT_BGNT         : goto STATE_LOCK_WAIT_BGNT;
        case LOCK_READ_WAIT_MEMORY  : goto STATE_LOCK_READ_WAIT_MEMORY;
        case LOCK_SLEEP             : goto STATE_LOCK_SLEEP;
        default                     : break;
        }

STATE_INITIAL:  // check if there is a request
    if (! Topology::cache_read_request(index)) return;  // not as such
    ++stat.req;                                         // statistics
    // much of the following should be self-explanatory:
    switch (in.request.type) {
        case REQUEST_READ :
            // first, check if there is a line in the cache
            if (cache_memory.isvalid(in.request.addr))  {
                // yes. There is nothing else to be done, so tell the processor
                // that its request was satisfied
                out.response.signal = PROCESSOR_SATISFIED;
                out.response.STROBE = 1;
                state = INITIAL;    // and set state to INITIAL (was already)
                return;             // nothing else to be done in this cycle
                }
            // no, the line is not in the cache, or is invalid
            out.BUS_REQ = 1;        // tell the bus that we need it and set
            state = READ_WAIT_BGNT; // 'state' so that in the next cycle we'll
            return;                 // continue from STATE_READ_WAIT_BGNT
        case REQUEST_UNLOCK:        // will fall down to the 'WRITE' case
            MEMSIM_WRITE_LOCK( in.request.addr, 0); // do the actual write
                                                    // to the memory
        case REQUEST_WRITE :
            // a write is always satisfied immediately
            out.response.signal = PROCESSOR_SATISFIED;
            out.response.STROBE = 1;
            out.BUS_REQ = 1;        // we need the bus to write the word through
            state = WRITE_WAIT_BGNT;
            return;
        case REQUEST_LOCK :
            out.BUS_REQ = 1;        // the bus is used as a binary semafore
            state = LOCK_WAIT_BGNT; // for accessing lock variables
            return;
        default: break;
        }
    return;


STATE_READ_WAIT_BGNT:   // here we come if we are in state READ_WAIT_BGNT
    // don't forget that all the inputs from the bus have been read already
    if (!in.BUS_GNT) return;    // return if the grant signal has not come yet
    // initialize cache memory line for 'fetching' from the memory: this
    // method operates on the current line; by calling isvalid(), we also
    // instructed the cache memory module to select an appropriate line for
    // replacement, which did so by inspecting in.request.addr's tag and
    // checking LRU info in the belonging set. So we initialize the cache
    // memory for this operation, and it will return the addr of the first
    // word in the line. Therefore, we don't have to know anything at all
    // about the cache geometry.
    cache_memory.line_fetch_init(&out.bus.addr);
    out.bus.memcmd = REQUEST_READ;  // a 'READ' cmd will be seen on the bus
    out.bus.size   = DATA_WIDTH;    // this isn't really necessary, but...
    out.bus.SNOOP  = 0;             // no snooping and no INV; we just want
    out.bus.INV    = 0;             // to fetch the line from the memory
    // now prepare to wait READ_WAIT_STATES cycles for the memory to finish.
    wait_state_counter = READ_WAIT_STATES;
    state = READ_WAIT_MEMORY;
    return;

STATE_READ_WAIT_MEMORY:
    if (wait_state_counter--) return;
    // ok, the slow memory has completed. Note that other protocols use
    // more sophisticated scheme, where the # of wait states for subsequent
    // words may differ from the number of wait states for the first word

    // check if the current address matched the one from the request
    if (out.bus.addr <= in.request.addr &&
        in.request.addr+in.request.size <= out.bus.addr + DATA_WIDTH) {
        // yes, send this word that has just been read to the processor
        out.response.signal = PROCESSOR_SATISFIED;
        out.response.STROBE = 1;
        }
    // cache memory hides the line size from us: so, we'll repeatedly call
    // .line_fetch_putnext() method, which will return the address of the
    // next word, and false if there are no more words in the line
    if (cache_memory.line_fetch_putnext(&out.bus.addr)) {
        wait_state_counter = READ_WAIT_STATES;  // more words to come
        return;
        }
    // counting done, whole line fetched
    // set new state to VALID
    cache_memory.setstate(WTI_Line_State::VALID);
    // drop the bus request. The bus module will in turn drop our
    // BUS_GNT input signal immediately
    out.BUS_REQ = 0;
    state = INITIAL;    // return to the initial state.
    return;

STATE_WRITE_WAIT_BGNT:
    if (!in.BUS_GNT) return;
    // a real cache would now do a "write" in its memory
    // send the bus the information for both the main memory and other
    // cache controllers:
    out.bus.memcmd   = REQUEST_WRITE;
    out.bus.addr     = in.request.addr;
    out.bus.size     = in.request.size;
    out.bus.SNOOP    = 1;                       // SNOOP = high
    out.bus.INV      = 1;                       // INVALIDATE = high
    wait_state_counter = WRITE_WAIT_STATES;     // now wait for the memory
    state = WRITE_WAIT_MEMORY;                  // to complete
    return;

STATE_WRITE_WAIT_MEMORY:
    out.bus.SNOOP    = 0;   // do not activate snoop controllers
    if (wait_state_counter--) return;
    out.BUS_REQ = 0;
    state = INITIAL;
    return;

    // This cache controller is implemented so that it knows about locks,
    // and performs them as TEST&TEST&SET instead of just TEST&SET.
    // TEST&SET could be accomplished by gaining exclusive access to the
    // bus, reading the location, writing non-zero to it, freeing the bus,
    // and returning the old value of the given location. TEST&TEST&SET
    // omits the third step (writing) if it founds a non-zero value in
    // the location, thereby saving possibly a lot of invalidations.
    // (Berkeley and Dragon use special arbitrator for locks so that the
    // bus can be used for ordinary read and writes unobstructedly.
    // If the controller finds a non-zero value on the lock address, it
    // could either return PROCESSOR_LOCK_BUSY response, and let the simulator
    // spin by calling the cache again and again, or it can spin itself
    // thereby making the simulation a bit faster. On the system where
    // each process resides on a different processor, there is no difference.
    // between the two approaches, so the other one is used, though it does
    // not reflect the reality. But the results will be the same.

STATE_LOCK_WAIT_BGNT:
    if (!in.BUS_GNT) return;
    // check if it is in the cache
    if (cache_memory.isvalid(in.request.addr))
        goto check_lock_value;
    // ok, there is not, read it from the memory
    out.bus.memcmd   = REQUEST_READ;
    out.bus.addr     = in.request.addr;
    out.bus.size     = sizeof(bitsLOCK);
    out.bus.SNOOP    = 0;
    out.bus.INV      = 0;
    wait_state_counter = READ_WAIT_STATES;
    state = LOCK_READ_WAIT_MEMORY;
    return;

STATE_LOCK_READ_WAIT_MEMORY:
    if (wait_state_counter--) return;

check_lock_value:
    if (MEMSIM_READ_LOCK(in.request.addr) == 0) {   // lock free
        out.response.signal = PROCESSOR_SATISFIED;
        out.response.STROBE = 1;
        // Do the actual "write 1" to the lock location in
        // real (application's) memory
        MEMSIM_WRITE_LOCK( in.request.addr, 1);
        // go to the regular WRITE to do the rest of the job
        goto STATE_WRITE_WAIT_BGNT;     // [we do have GNT already]
        }
    // the lock is busy; there is nothing to be written. drop the bus and sleep
    out.BUS_REQ = 0;
    busy_lock_sleep_cycle_count = BUSY_LOCK_SLEEP_CYCLE_COUNT;
    state = LOCK_SLEEP;
    return;

STATE_LOCK_SLEEP:
    if (busy_lock_sleep_cycle_count--) return;
    out.BUS_REQ = 1;
    state = LOCK_WAIT_BGNT;         // start again
    return;
    }
```

Your cache controller may not be as simple as the WTI controller, not as much in the sense of a code line count, as in the sense of a trickiness of its behavior. Again, Berkeley is a good example: for instance, if its cache controller wants to write the data to a line, and if it finds that the line is not exclusively owned, it must send an invalidation signal. Before it sends it, it must wait to get the bus first. But after the bus becomes owned by him, it must check whether the line is still in the same state it was before asking for the bus – because another processor could have invalidated the line meanwhile. So writing the cache controller code (and designing the controller hardware!) somewhat resembles parallel programming, in the situation where the designer attempts to put as little data in critical section as possible in order to achieve more concurrency, but pays the price of having to worry about every possible scenario whose number increases as the concurrency grows. In this case, the bus serves as the hardware binary semaphore, and critical operations, such as writing to shared data, are enclosed in bus acquire and bus release operations. A write would be more simple if it acquired the bus before even checking whether the line is exclusively owned, but such a hardware would be too slow.

Of course, if you estimate that some effects are too unlikely to happen in your simulations (*and* if you do not do data movement) you can ignore them, if their mishandling can't introduce a distortion larger than few percents. But it seems hard to predict how often something would or would not happen, and it's better to go straight.

## 5.5.8 Topology class

The Topology class is the heart of the system. It organizes the communication, reading output ports of some modules and writing their contents to the input ports of the others.

It is also responsible for polling the modules in correct order to do their share within a simulated cycle.

To repeat, this is the only class that possesses the knowledge of the whole simulated system.

### 5.5.8.1 Compilation issues

Every module (that communicates through ports) must be aware of the Topology class; and the Topology class must be aware of every communicating module.

Here is how this circle is broken: the header file where the Topology class is defined uses only module class names; it does not refer to any specific instance of a module. This header file is included into the header file where the module class is defined, and the Topology class is declared its **friend**, for it will peek and poke module's private variables. The Topology class implementation includes all the headers where classes of communicating modules are defined, for it will be dealing with specific instances. Refer to the picture in section 5.3.6, and you will see this compilation dependency yourself.

Take another look at the source of the WTI cache, for example:

```
#include "snoopy.h"
#include TOPOLOGY_HDR      // TOPOLOGY_HDR symbol is defined as "topology.h"
#include "cache_memory.h"
#include "bus_signals.h"

class WTI_Cache {
    ...
    struct In {
        ...
        Bus_Signals bus;
        } in;
    ...
    TOPOLOGY_FRIENDS       // this macro expands to "friend class Topology;"
    };
...
void WTI_Cache::cycle() {
    // read the bus contents first
    Topology::cache_read_bus(index); // call Topology to read the bus
    ...
```

The method `Topology::cache_read_bus(index)` that the cache calls will read the contents on the bus and write it into WTI_cache's in.bus variable.

All the variables (declared pointers to instances of the modules, or pointers to them) and methods in topology class are declared **static**. The topology could also be implemented with regular stand-alone functions, but grouping them into a class is neater.

### 5.5.8.2 Topology declaration

This is the definition of the Topology class, included by the Bus, IO_Buf, and WTI_Cache (**topology.h)**:

```
#include "snoopy.h"           // definition of the aliases that follow
class CACHE_CLASS;            // class WTI_Cache;
class BUS_CLASS;              // class Bus;
class IO_BUF_CLASS;           // class IO_Buf

class Base_Topology {
  protected:
    static IO_BUF_CLASS             * io_buf[MAX_CPU_COUNT];
    static CACHE_CLASS              * cache[MAX_CPU_COUNT];
    static BUS_CLASS                * bus;
  public:
    // services for the communicating modules
    static bool io_buf_input(int index);
    static bool cache_read_request(int index);
    static void cache_read_bus(int index);
    static bool bus_read_REQs();
    static bool bus_read_LOCK_REQs();
    static void bus_read_contents(int master_index);

    // system services,called by the Memsim class. They exist for all simulators
    static void system_init();
    static void system_cleanup(Thread_Time last_time);
    static bool system_isidle();
    static void system_dump_state();
    static void system_collect_requests();
    static void system_cycle();
    };
```

The purpose of these methods should be obvious: `cache_read_request(index)` reads the request from the output port of the index-th instance of the IO_Buf class and writes in the input port of the index-th instance of the WTI_Cache class. "cache" is the class that needs service, "index" is the index of its instance, "read_request" is the name of the service.

System services are a bit more interesting: apart from the obvious `system_init()` and `system_cleanup()`, `system_isidle()` checks if all the modules are in their idle state. This information is used by the Memsim class (described below) to optimize the simulation (as discussed in 5.1.1) and to save time to avoid unnecessary polling of idle modules at cycles where nothing can happen. `system_collect_requests()` forces IO_Bufs to ask the kernel for pending requests, and it is called by the Memsim at the times he knows there must be some. `System_cycle()`, on the other hand, is called for every simulated cycle.

The information about the system topology does not go below the Topology class – to the communicating modules, which makes them simple and "context-free", nor it goes above this class, which makes the Memsim class generally usable for all kinds of simulators.

### 5.5.8.2 Topology implementation

Only here are instances of the modules defined. Only the implementation of the topology class knows how many modules exist in the system and how they are connected. It is responsible for creating and destroying them. Have a look at the source, which is pretty simple indeed:

```
#include <stdio.h>
#include "topology.h"
```

```
#include CACHE_HDR      // #include "wti_cache.h"
#include BUS_HDR        // #include "bus.h"
#include IO_BUF_HDR     // #include "io_buf.h"

// we have to instantiate objects through pointers
IO_BUF_CLASS             * Base_Topology::io_buf[MAX_CPU_COUNT];
CACHE_CLASS              * Base_Topology::cache[MAX_CPU_COUNT];
BUS_CLASS                * Base_Topology::bus;

// these routines are called from the memsim
void Base_Topology::system_init() {
    bus = new BUS_CLASS;
    for(int i=0; i<MAX_CPU_COUNT; ++i)  {
        io_buf[i] = new IO_Buf;
        cache[i]  = new CACHE_CLASS;
        io_buf[i]->assign_index(i);     // each module in the array is assigned
        cache[i]->assign_index(i);      // an index
        }
    cache[0]->Introduce();              // say who you are (to be printed out)
    }
// system_cleanup is mainly used for dumping statistics:
void Base_Topology::system_cleanup(Thread_Time last_time) {
    PRINTF("TOTAL simulation time: %d cycles.\n", last_time);
    for(int i=0; i<MAX_CPU_COUNT; ++i)
        cache[i]->dump_stats();
    bus->dump_stats(last_time);
    }
// returns true if all the modules in the system are idle:
bool Base_Topology::system_isidle() {
    for (int i=0; i<CPU_COUNT; ++i) {
        if (!io_buf[i]->isidle()) return false;
        if (!cache[i]->isidle()) return false;
        }
    return true;
    }
void Base_Topology::system_dump_state() {
    for(int i=0; i<CPU_COUNT; ++i)
        cache[i]->dump_state();
    bus->dump_state();
    }
// Called at times when a set of request *could* exist:
void Base_Topology::system_collect_requests() {
    for(int i=0; i<CPU_COUNT; ++i)
        io_buf[i]->start_cycle();
    }
// The main function: polls all the modules to do their share in this cycle:
void Base_Topology::system_cycle() {
    for(int i=0; i<CPU_COUNT; ++i)
        cache[i]->cycle();
    bus->cycle();
    for(i=0; i<CPU_COUNT; ++i)
        io_buf[i]->end_cycle();
    }
// Topology for your own simulator will need all the methods listed
// above; only implementations of these method will differ.

// These methods are called from various modules; their names and
// implementations depend on the structure of your system:

// ******************* write in IO_buf's input port *******************

// each IO_Buf calles this method (passing its index) upon the end
// of each simulated cycle. This method checks if the cache controller
// placed immediately below the io_buf module in question has set its
// STROBE bit high. If so, the controller has an information to communicate
// to the io_buf module. The topology will read this information from the
// controller's output port, write it into the io_buf module's input port,
// and clear the STROBE bit.

bool Base_Topology::io_buf_input(int index) {
    if (cache[index]->out.response.STROBE) {
        io_buf[index]->in.response.signal =
            cache[index]->out.response.signal;
        cache[index]->out.response.STROBE = 0;  // clear
        return true;    // tell the io_buf that a response appeared
        }
    else
        return false;
    }

// ******************** cache modules services ********************

// The similar procedure is followed for reading io_buf's output
// (request description) and writing it into cache controller's
// input port. The STROBE bit is also used. However, the controller
// module will not call this method at every cycle, but only when
// it's idle. If it is already processing previous request (such as
// read that was a miss), it will not call this method and the STROBE
// bit will not be cleared. Only when the controller gets idle again
// (returns to its INITIAL state) it will call this method. By inspecting
// its STROBE bit, the IO_Buf can know that the controller has read the
// request.

bool Base_Topology::cache_read_request(int index)   {
    if (io_buf[index]->out.request.STROBE) {
        cache[index]->in.request =
            * io_buf[index]->out.request.request_ptr;
        // this is where virtual address translates to physical.
        // In this case, just physcial = virtual.
        cache[index]->in.request.addr =
            (cache[index]->in.request.virtual_addr);
        io_buf[index]->out.request.STROBE = 0;          // clear
        return true;    // yes, a request exists
        }
    else
        return false;   // no request this time
    }

// ***************** BUS-related Topology functions *****************

// The bus module calls these to read bus and lock requests
bool Base_Topology::bus_read_REQs() {
    bool any=0;
    for (int i=0; i<CPU_COUNT; ++i)
        any = any | (bus->BUS_REQ[i] = cache[i]->out.BUS_REQ);
    return any;
    }

bool Base_Topology::bus_read_LOCK_REQs() {
    bool any=0;
    for (int i=0; i<CPU_COUNT; ++i)
        any = any | (bus->LOCK_REQ[i] = cache[i]->out.LOCK_REQ);
    return any;
    }

// this method is called by the bus when it wants to read the contents
// of the controller's output port that should appear on the bus,
// for the module that currently holds the bus

void Base_Topology::bus_read_contents(int master_index) {
    bus->bus_contents = cache[master_index]->out.bus;
    // if someone has SUPPLY high, his data goes to the bus
    bool supply_line = 0;
    bool sh_line = 0;
    for(int i=0; i<CPU_COUNT; ++i) {
        if (cache[i]->out.bus.SUPPLY) {
            supply_line = 1;
            }
        sh_line |= cache[i]->out.bus.SH_LINE;
        }
    bus->bus_contents.SUPPLY  = supply_line;
    bus->bus_contents.SH_LINE = sh_line;
    }

// this method is called by each cache controller module when it wants
// to read what is on the bus. Of course, each cache controller calls
// this method in each simulated cycle.

void Base_Topology::cache_read_bus(int index)   {
    cache[index]->in.BUS_GNT = bus->BUS_GNT[index];
    cache[index]->in.LOCK_GNT = bus->LOCK_GNT[index];
    cache[index]->in.bus     = bus->bus_contents;
    }
```

### 5.5.9 Memsim class

Here is another set of functions that did not have to be grouped in a class at all. These functions are called by the Limes kernel, and the whole class (including the functions that represent C to C++ interface) need not be changed for any simulator. The only class Memsim knows directly about is the topology class.

So here it is (vital part):

```
class Memsim {
    Thread_Time last_simulation_time;
  public:
    void init() { Topology::system_init(); }
    void end()  { Topology::system_cleanup(last_simulation_time); }
    void simulate(Thread_Time time_before, Thread_Time time_now);
    void dump_state()
        { Topology::system_dump_state(); }
    } memsim;

/*
 * The main routine. Called by the kernel every time a new request or a set
 * of requests appears, and in successive cycles where all the processors
 * are stalled.
 */
void Memsim::simulate(Thread_Time time_before, Thread_Time time_now) {

    for(Thread_Time time=time_before; time<=time_now; ++time) {
        int i;

        // optimization: if the whole system is in initial time,
        // shift current time to present
        if (time < time_now && Topology::system_isidle()) time = time_now;

        // if time now is time_now, wherein there are some requests, for
        // that is the reason we have been called in the first place,
        // collect them. The only situation where before != now is when
        // in before-1 everyone was satisfied, and we had to finish a
        // write or something. From before to now-1 there are no requests.
        if (time == time_now)
            Topology::system_collect_requests();
        Topology::system_cycle();
        }
    last_simulation_time = time_now;
    }
```

And that's it. The kernel calls Memsim, Memsim calls Topology, Topology calls each of its modules. And all that for every single read/write/lock/unlock the application issues! That is the cost of an accurate simulation.

## 5.6 Hints and tips

If you are an experienced memory simulator writer, you can skip this section. Otherwise, you can read a few hints that can make your life easier:

- Develop and test your simulator as *detached*, at first. Debug it on simple traces and link it with the kernel only when it has overcome its child diseases. Test it thoroughly; it's hard to determine the nature of the error when it's linked to a real application that generates million of events.
- Use as many Limes-included files as you can. These may be **ber.make, makefile, memsim.cc, io_buf.h,cc**. Rather modify them than to create your own from scratch.
- Include `dump_state()` member function in all your classes. It will help in debugging greatly.
- Use `PRINTF` and `COUT` instead of `printf()` and `cout`. You will be able to handle your output more gracefully.
- For each line of state dump, try to satisfy the form *Module_keyword<module index><other info>*. Doing so will help you to quickly extract the desired information using pipes and the `grep` utility.
- If you test the memory system at which the coherence units can be found in different states, try to create test traces that will cover all state transitions.
- Be well acquainted with `make`, `gdb`, and possibly `xgdb`.

## 5.7 Step by step: How to do it

Let us now describe a series of steps that are to be taken if you want to write your own memory simulator. This example will assume that you want to simulate a bus-based, multilevel-on-chip-cache multiprocessor system, but will not go into details of its implementation. (Future versions of this document may include a more elaborate, "live" example).

**Step 1: create the directory**

Create a directory in **limes/simulators/** called, for example, **mysim**.

**Step 2: copy the necessary files**

Copy the following files from the snoopy simulators: memsim.cc, io_buf.*, cache_memory.h, snoopy.h, bus.*, bus_signals.h, topology.*, ber.make, makefile. Rename ber.make to mysim.make. Edit the makefile and put SIM=mysim, SIMHOME=mysim in it. The makefile will be used for developing the simulator as DETACHED. Later, you will edit the first line of mysim.make to list the .cc files that comprise your simulator.

**Step 3: modify the parent header file**

In the snoopy example, snoopy.h is the parent file: all the other modules include it. It includes memsim_interface.h, and defines macros and constants that are common for all the modules. You can delete the unnecessary part from it and include your own macros. If you rename it to something else, make sure the other  files that #include it see a new name.

**Step 4: design your simulator**

This is the creative part: represent each module in your system as a box with input and output ports. In our example, you will have the following modules: L1_cache, L2_cache, and the bus. Define all the input and output ports, the information that the modules will communicate to each other, and the signals on the bus.

**Step 5: define the topology functions**

Pick appropriate names for the topology services that your L1 and L2 modules will require.

**Step 6: define the Bus_Signals class**

Edit bus_signals.h and define the signals that can appear on the bus in the system you simulate.

**Step 7: implement the L1 cache controller**

Create the L1 cache controller in a way similar to the WTI cache controller module is created. Define the L1_Line_State class for the L1 cache and create an instance of Cache_Memory<L1_Line_State> in this controller. The L1 cache controller will perhaps be written in such manner that all the writes in it are of write-through type, so that it is relieved of maintaining consistency (the L2 will take care about it). Create an input port named, for example, "function complete" which will be high when the L2 cache has finished processing a request from the L1 cache.

**Step 8: implement the L2 cache controller**

Actually, the L2 cache will resemble one of the snoopy controllers presented above more than the L1 cache: it will use the bus to communicate with other L2 cache controllers. You can model this controller to have more than zero wait states for a read/write hit. Use sys_INI_read rather than to hardcode such constants in it. The same applies for cache memory parameters. It is very likely that the L2 cache will have different line states and different cache geometry than the L1 cache.

**Step 9: implement the topology functions**

Modify topology.h to have a pointer for each L1 and L2 instance. Modify topology.cc to implement member functions of this class that provide services to the requesting modules.

**Step 10: compile the simulator as DETACHED and test it**

To this, just do a "make" from the simulator's directory. (Do a "make dep" before it.) You will get an a.out file that you can run and specify the test trace file as its argument. For a description of the test trace file and detached simulators, take a look at README files in the limes/simulator/ and limes/simulator/snoopy/test/ directories. You will also find a sample trace file there.

The a.out executable contains your simulator and a main() function which reads the trace and feeds your simulator with it.

Test thoroughly; create a test trace for each state transition, to cover all the cases. Use the **gdb** tool. Follow the tips given in the previous section.

When you are ascertained that your simulator is flawless, you can proceed to the final step.

**Step 11: link your simulator to a live application**

Pick an application (the "simple" application can be a good start, the next best is "fft"), edit its makefile, change SIM and SIMHOME to "mysim", do "make simclean" and then "make". Now you can run and test it, you can vary the simulation parameters and collect the resulting output files.

If you find a bug in the simulator (so that it causes the application to generate a segmentation fault or a similar dreadful thing), compile the application with "-g" switch and debug it. Use DEBUG_EVENTS and DEBUG_SOURCE macros, they can help greatly. Still, the best bet is to carefully design and test all the cases in the detached phase; with tens of thousands of references a second it's hard to find an error!

# 6. Conclusions and future work

Future versions of the tool may include simulators for other systems; hopefully one or more DSM examples will be included along with the snoopy protocols.

Initially, the idea was to improve the simulation speed at parts that are not memory-simulator dependent; however, it probably cannot be much faster than it is now (maybe up to 30% for the PRAM model) and, for detailed hardware simulation, the memory simulator will consume more than 90% of the time anyway. So the future efforts will be directed to creating reusable additions to the whole package while retaining the simplicity of the system.

Limes kernel will be modified to allow for process migration, more processes than processors etc. The simulators that follow the interface described above will not have to be changed.

Timing for standard library functions will be improved. Perhaps the time for the functions that call the UNIX kernel will be accounted by measuring processor ticks and converting it into simulated cycles.

You can reference the older version of LIMES simulator (LIMES v1.0) in your work as:

- "Limes: A Multiprocessor Simulation Environment for PC Platforms," IEEE TCCA Newsletter, March 1997.

If you use the new version (LIMES v1.1), you can reference it as :

- Ikodinovic, D. Magdic, A. Milenkovic, V. Milutinovic, *Limes: A Multiprocessor Simulation Environment for PC Platforms*, Third International Conference on Parallel Processing and Applied Mathematics (PPAM), Kazimierz Dolny, Poland, September 14-17, 1999

Last but not least, if you encounter any problems, bugs, or if you have any suggestions, please mail them to limes@rti7020.etf.bg.ac.yu.

# Appendix A: State transitions

The following state diagrams and state explanations for the Berkeley and the Dragon protocol are excerpts from Prof. Milo Tomasevic's Ph.D. "The Word Invalidation Protocol".

## A.1 The Berkeley Protocol

In this protocol, a block can be found in one of the four states:

|  |  |
|---|---|
| ■ *INV* | (Invalid) |
| ■ *UNMOD-SHD* | (UnOwned) |
| ■ *MOD-SHD* | (Owned Non-Exclusively) |
| ■ *MOD-EXC* | (Owned Exclusively) |

**Read Miss**

When a "read block" request is issued, the cache memory that owns the block in MOD-EXC or MOD-SHD state responds, and then changes its local state to MOD-SHD. If there is no such cache memory, the main memory sends the block. The cache memory which issued the request receives the blocks and sets its state to UNMOD-SHD.

**Write Hit**

If the block is in the MOD-EXC state, the write performs locally, without going out to the bus. If the block is in one of the shared states (UNMOD-SHD or MOD-SHD), before the write performs and the state changes to MOD-ESC, it is necessary to send the invalidation signal to the bus. Other cache memories that hold the block invalidate it by setting it to the INV state.

**Write Miss**

This situation is a combination of a read miss and a write hit, in a way. To perform the write of the word, it

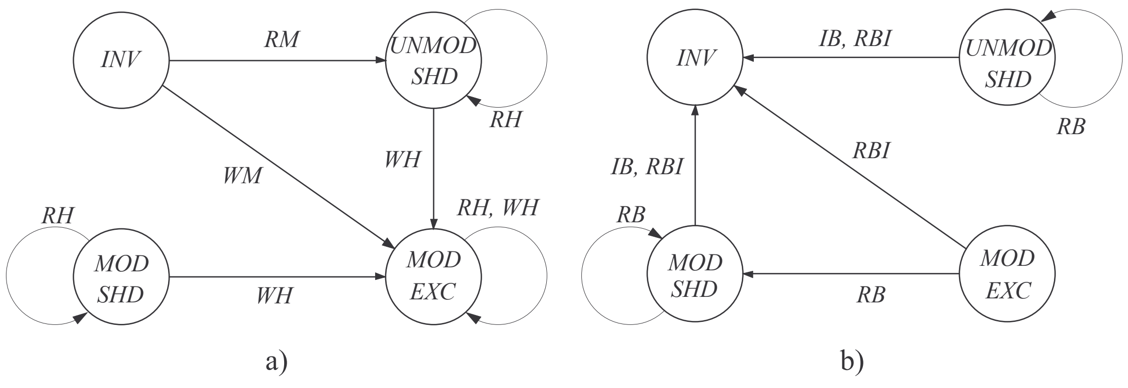


**Figure 10:** State transitions for the Berkeley protocol
a) due to processor operations b) due to bus operations

|  |  |  |  |
|---|---|---|---|
| *RH* — | read hit | *RB* — | read block |
| *RM* — | read miss | *RBI* — | read block and invalidate |
| *WH* — | write hit |  |  |
| *WM* — | write miss | *IB* — | invalidate block |

is necessary to read the whole block beforehand. The block owner responds to the request. After the write, the state of the block changes to MOD-EXC. All the other copies (in other cache memories), if exist, are set to the INV state.

**Replacement**

In the case that the block has been chosen for deposition from the cache memory, and its state is either MOD-EXC or MOD-SHD, its contents must be written to the main memory before the disposition takes place.

## A.2 The Dragon Protocol

As this protocol does not use invalidation, the usual INV state is not necessary, so another state is introduced which represents an unmodified, exclusive state of the block. Therefore, the block in Dragon protocol can be in one of the four states as well:

- *UNMOD-EXC* (Unmodified Exclusive)
- *UNMOD-SHD* (Unmodified Shared)
- *MOD-SHD* (Modified Shared)
- *MOD-EXC* (Modified Exclusive)

The protocol also assumes the existence of the special bus line (the SH line, active low in tables 1.1 and 1.2) which enables the dynamic sharing detection.

**Read Miss**

If the cache memory that owns the block exists, it provides the block, activates the SH line and sets the local state to *MOD-SHD*. Otherwise, the block comes from the main memory. If, while reading the block, cache memories that contain the block in *UNMOD-EXC* or *UNMOD-SHD* state exist, they activate the SH line and set their state (meaning, the state of the block in question) to *UNMOD-SHD*. The cache memory that issued the request accepts the block and sets its state to *UNMOD-SHD* (if the SH line is active), or *UNMOD-EXC* (if the SH line is inactive).

**Write Hit**

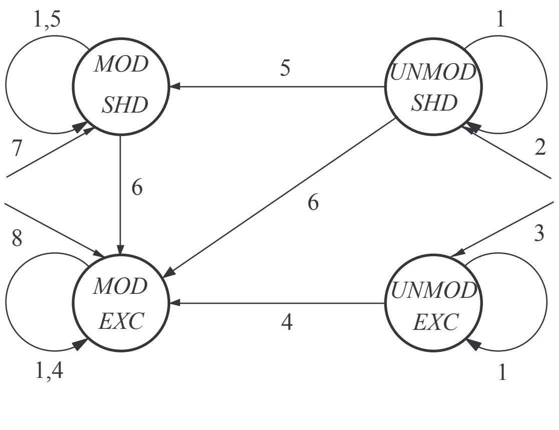
If the copy of the block is exclusive (state *UNMOD-EXC* or *MOD-EXC*), the write performs locally, without going out to the bus, and the state is set to *MOD-EXC*. If the state is either *UNMOD-SHD* or *MOD-SHD*, the word that needs to be written is sent out on the bus, so that all the cache memories that own the block accept the word and update the block, while activating the SH line. Based on the state of the line, the cache memory – initiator recognizes whether there are more copies of the block, and sets the new state to *MOD-SHD* (if the SH line is active), or *MOD-EXC* (if the SH line is inactive).

**Write Miss**

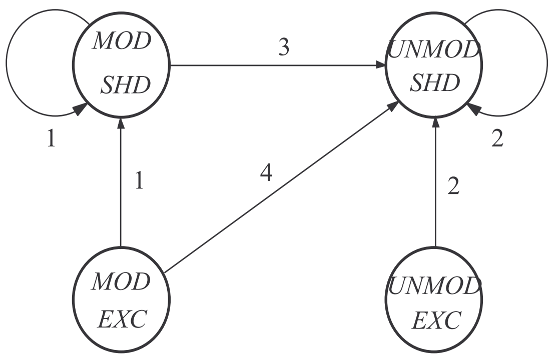
The reaction to this situation begins as in the case of a read miss. If there exists a cache memory – owner, it yields the block, activates the SH line, changes the local state to *UNMOD-SHD*, while the other cache memories, if they contain the block, only activate the SH-line. The cache memory that issued the request takes the block, and, if the SH line is active, proceeds as in the case of a local write, and the local state is set to *MOD-SHD*. If the SH line is inactive, the write performs locally, and the new state is *MOD-EXC*.

**Replacement**

As with the Berkeley protocol, for the blocks in *MOD-EXC* or *MOD-SHD* state, it is necessary that, before the replacement, the block be written back to the main memory.



a)



b)

**Figure 11:** State transition diagram for the Dragon protocol
a) due to processor operations
b) due to bus operations

| number | PCC action | SH# | bus operation |
|---|---|---|---|
| 1 | RH | — | |
| 2 | RM | 0 | read block |
| 3 | RM | 1 | read block |
| 4 | WH | — | — |
| 5 | | 0 | write word |
| 6 | WH | 1 | write word |
| 7 | WM | 0 | read block and write word |
| 8 | WM | 1 | read block |

**Table 1.1:** Actions of the PCC (Processor cache controller) part of the Dragon protocol

| number | bus operation | SH# | SCC action |
|---|---|---|---|
| 1 | read block | 0 | send the block to the bus |
| 2 | read block | 0 | — |
| 3 | write word | 0 | update word |
| 4 | read block and write word | 0 | update word |

**Table 1.2:** SCC (Snoopy Cache Controller) actions for the Dragon protocol

# Appendix B: File formats

## B.1 Trace file

By issuing "**-- -t**" option when running an executable simulation, you can tell the simulation kernel to produce a trace stream on the standard output (or standard error). Memory references are print out as they happen, and are therefore dependent on the actual simulated system. For example, you can link an application with the Berkeley simulator, and instruct the kernel to produce traces. These traces correspond (to a certain extent) to the events that N Intel processors with Berkeley protocol implemented would generate.

This "trace stream" is in plain ASCII; you can redirect it to a file (or pipe) and refine it further, or modify the (small) routine `kernel.c:produce_trace()` to arrange it in a format you find more suitable.

The default format is as follows: the trace consists of a series of lines, each corresponding to a certain point in time when at least one processor generated an event, in chronological order:

```
T=<time> {P<n>:<Operation><Area> <Address>,<size>}...
```

For example:

```
...
T=10240   P0: Rd 220402,4     P3: Wd 324816,4  P5: Rs 3290482312,4
T=10243   P0: Rd 220406,4     P4: Ld 402442,4
...
```

The meaning of each identifier is as follows:

- *time*: the point of time (in cycles) when request(s) were generated
- *n*: ordinal number of the processor that generated the event (starting from 0)
- *Operation*: either **R**, **W**, **L**, or **U**, which stands for READ, WRITE, LOCK or UNLOCK, respectively
- *Area*: either **c**, **d**, or **s**. "c" stands for code segment, "d" for data segment, "s" for stack segment. All the references to "c" and "d" segments are inherently shared, but code segment can be read only. "s" segment is private for each thread. All the locks and unlocks occur in the shared (data) area.
- *Address*: virtual address generated. See Appendix C for more information on virtual address space layout.
- *Size*: size of the data read/written. 1 for byte, 2 for word, 4 for long (by far the most frequent case), 8 for double (if you declare bus width to be 64bits; by default it's 32 only, and doubles are split to two 4-byte chunks ).

Still, regardless of whether you want to perform a trace analysis, or to simulate your own protocol, you'll be better off if you attach your routines directly to the kernel, thus avoiding long trace files or slow piping/redirection.

# Appendix C: Threads' address space

This section will shortly describe what you need to know about the virtual address space layout of the simulated concurrent threads. Actually, this pertains not only to Limes, but probably to threads on Linux in general. For (much) more information than you will find here, please consult the "Linux Kernel Hacker's Guide".

Basically, what the section states is that you don't have to perform virtual-to-physical address translation, and that you can call **ISSHARED(address)** macro at any point in your simulator to determine whether an address is shared or not (that is, whether it belongs to the data segment, or not). If you want to know why, do continue reading.

## C.1 Virtual address space layout for a single process

Virtual address space for a process on Linux spans the area from 0 to 4GB. Actually, the last 1GB is not visible by the process in user mode (there are no page table entries for the range 3GB-4GB), so we have the range from 0 to 3GB.

Only parts of this enormous area are really used, of course. The code segment starts at address 0 (though the linker can be instructed to change this, as well as few other things, but these features are not used). The data segment begins immediately after the code segment, rounded to the next page (pages are 4KB long), and grows upwards, upon each malloc() (more precisely, upon each brk()). It consists of initialized data, uninitialized data (all zeroes), and the heap.

The stack segment begins at 3GB, and grows downwards. It's obvious that the bottom of the stack and the top of the heap will never meet in reality.

Portions around the middle of this space are also used, but they are not of interest here. For example, UNIX shared segments (attached with shmat()) are placed here, and this is also the place for some system and standard library stuff. Since our parallel applications do not do system things, and most of the standard library functions they ever call are covered and instrumented, this middle-area is not of our concern.

Figure 12 shows an example of how it could look like in a typical case. All the heavy-weight processes have similar memory layout, and virtual-to-physical translation tables are established for each such process.

## C.2 Virtual address space layout for multiple threads

When multiple threads of execution are created (for simulation purposes by Limes, in reality by any of the newer Linux releases), they are defined as regular processes that share address space with their parent. More formally, page table entries for mapping virtual addresses to physical have the same values for all the threads. Figure 13 depicts this situation for an example of four threads.

All the threads share the code segment (of course) and the data segment, and each gets its reasonably wide share (say, 256K) of the upper area for the stack. The tops of their stacks are different. Stacks are private only semantically: a thread *could* read and write another threads' stack, but never does so.

This brings us to the point: why do you not have to translate virtual addresses to physical in simulation? The answer is – because no two different virtual addresses point to the same physical address, and no two equal virtual addresses point to different physical addresses. So you can assume that these threads of the parallel application are the only set of processes that are executing on your simulated machine. And you can assume that the code segment was really loaded into memory at address zero, and so on.

There are certain leaks in this assumption: first, you do not have 3GB of physical memory to assume that the stack is physically really placed at 3GB minus something; and second, in reality, the lowest part of the physical memory would be assigned to system tables etc. But since all the allocations occur at page boundaries, it's only a matter of distribution of block tags in your associative memory in caches, which is unpredictable anyway, and has little impact on general behavior for 2-way caches or caches with higher associativity, or larger caches. Still, it'd be easy to create a translation function, should you need one.

Finally, a note on the **ISSHARED(addr)** macro: it returns true if (and only if) the address given belongs to the **data** segment. Formally, code segment is shared as well; but it can be read only, and is therefore not important from the aspect of coherence maintenance. (Note that reads from the code segment can occur, though Limes does not capture instruction fetches; this is because the application may read constants embedded in the code segment, mainly string constants, for various `printf()`s that never occur in the parallel computation phase. If you really need to know whether an address belongs to the code segment, say `if (addr < DATA_SEGMENT_START)`.)

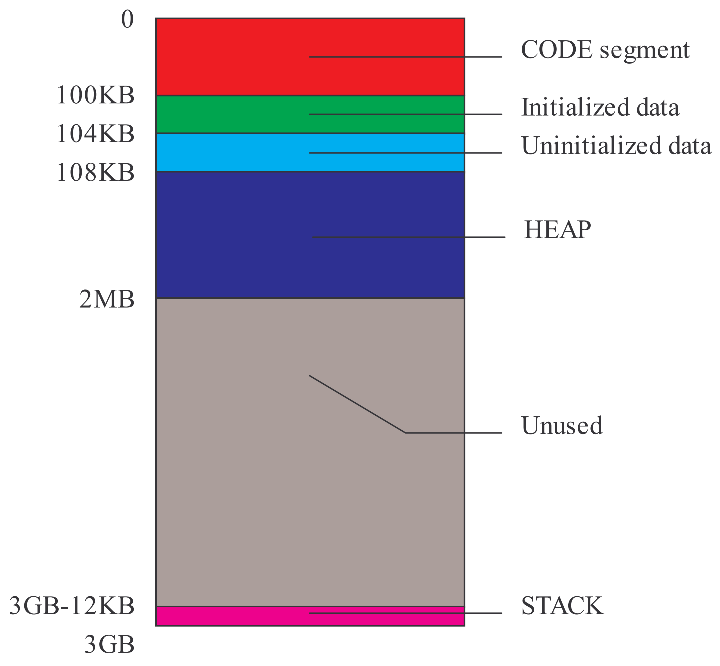


**Figure 12:** Virtual address space layout for a single process (an example)

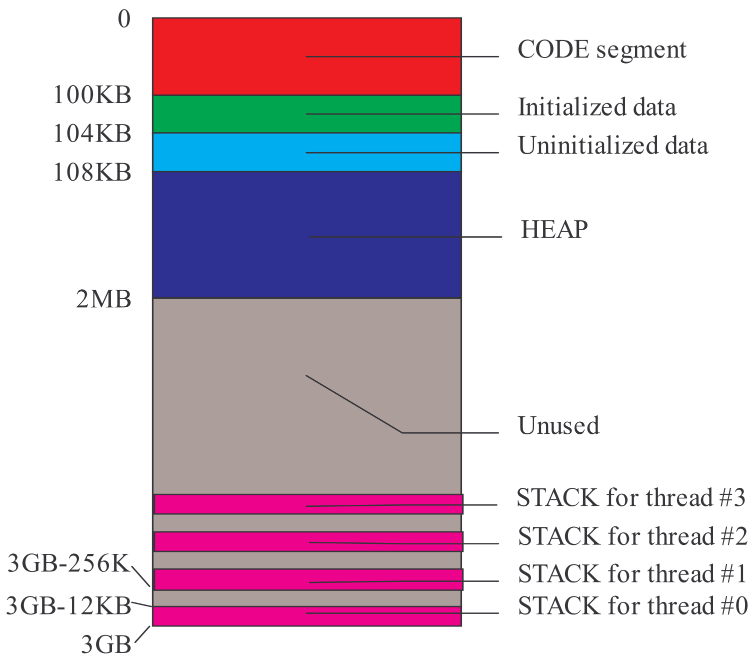


**Figure 13:** Virtual address space layout for multiple threads (an example)

# Appendix D: Java-like multithreading

Here will be described a system for building parallel C++ applications in a manner similar to the one employed by the Java language, which is chosen for its simplicity. The source code for this system resides in **limes/applications/javalike** directory, and it also includes a few "popular" examples, like the Producer-Consumer problem, and the Dining Philosophers problem.

The whole "system" is actually rather small, it's only one relatively small file, and is built on top of Limes, without having any knowledge of its internals. Therefore, it could be used with real parallel machines for which the ANL "parallel macros" support exists, or with any other multiprocessor simulator which uses ANL macros for lock acquiring and releasing (and supports C++).

The system implements only two classes – threads and monitors – and programs written to use them are far more readable and understandable than those using "raw" LOCK and UNLOCK directives. These classes, and their usage, will be described below. Previous knowledge of the Java language is not required, though reading the "Threads of Control" section from [CaW96] is recommended.

## D.1 Concepts

The intention behind the Object-Oriented approach to parallel programming equals the intention behind the Object-Oriented approach in "regular", sequential programming: to improve the readability and maintainability of the code, making the development of applications easier. Programming with manual locking and unlocking of critical sections and control of the concurrent threads of execution is tedious and very error-prone. Such programs are hard to debug, etc. Therefore, it is desirable to establish two basic mechanisms for parallel programming in an object-oriented environment, such as C++ or Java: the first is to make the threads appear as "live" objects that can be manipulated, and the second is to provide some implicit synchronization between data that can be shared among multiple threads.

The system described here implements two classes: the Thread class and the Monitor class. In order to create threads as objects, one should extend (derive, inherit from) the Thread class and make instances of such extended class. In order to provide synchronization between data in a passive shared object, such a passive object should extend the Monitor class and designate (some of) its methods as *synchronized*, so the threads that call the methods of an object of such a class can be assured that all the critical operations are performed atomically. Both classes, and their usage, will be described now.

## D.2 The Thread class

To create a class that describes "live", active objects, it is sufficient to make that class have only one method, **run()**, which will contain the body of the thread, and, of course, to declare the class as derived from the Thread class. (Note: in Java, multiple inheritance is not allowed, a class that must be derived both from the Thread class and some other class will be declared as "implementing *Runnable* interface"; in C++, multiple inheritance is OK.)

An object created from such a class is a live object; or rather, it becomes live after its **start()** method is called. So a Thread object (hereby we mean "an object instantiated from a class that is derived from the Thread class) basically has three states: *created* (its data is allocated in memory, but it's still just data), *running* (executing as a separate thread), and *finished* (or *dead* – finished its execution).

Now let's take look at the example of a thread that, after being born, just prints "Hello world" and dies:

```
class HelloWorldThread : public Thread {
   void sayHello() {
      printf("Hello world!\n");
      for (int i=0; i<1000; ++i);   // make a short delay before you die
```

```
        printf("...and goodbye.\n");
        }
    public:
     void run() {
        sayHello();
        }
     };

void main(int argc, char *argv[]) {
   HelloWorldThread *t;
   t = new HelloWorldThread;
   t->start();
   printf("Parent: continuing.\n");
   delete t;
   }
```

Now let's analyze what happens from the beginning of the program (the main() function). First, the program executing the main() function is a thread itself – the master thread, and so far the only one. The first thing it has is declaration of a pointer to an object of HelloWorldThread class, which is derived from the Thread class. The first thing it does is the allocation – creation – of a new HelloWorldThread object, with the `new` operator. Right now, the new thread's data is allocated, but the thread, as a process, still does not exist. However, some of its data are implicitly initialized. If the HelloWorldClass had a constructor, it would also be executed at the creation time.

The next thing the master thread does is bringing the new thread to life. It does so by calling the thread object's `start()` method. At that point, the new thread (as a process) is created on a free processor and begins execution of its `run()` method. From now on, two threads exist at two distinct processors: the parent, which proceeds to its printf() instruction, saying that is continuing whatever it is doing, and its child, an instance of the HelloWolrdThread class, which calls its private `sayHello()` method. These two threads are executing "in parallel", independently. So the parent will print that it's continuing, and the child will say "Hello world".

The next thing parent does is deleting the thread. But it is a synchronized delete, and it will not actually be performed before the child has reached the end of its run() method. In fact, the parent will attempt to do the delete nearly at the same time the thread enters its dummy for() loop. But it will wait patiently until that loop finishes, and the child thread prints the goodbye message and dies. Therefore, as you can see, the `delete t` operation will not be completed before the thread being the subject of deletion has finished. This is a substitute for garbage collection that you have in Java language for free; in C++, the deletion must be performed manually.

The parent could have allocated and activated as many threads of different kinds it had wanted, and child threads could have created another (possibly different) threads themselves, being their parents in turn. The only rule the programmer should obey in creating threads is that **thread objects must not be created as automatic objects (on the stack)**. They must either be declared as global variables, or explicitly created with the **`new`** operator; the later is more in the spirit of OOP. Of course, *pointers* to threads can be declared as stack variables, as shown in the example.

The `run()` method is mandatory, in the Thread class it is declared as a pure virtual function. The `start()` method is also virtual, and classes derived from the Thread class could have their own, provided that they do not forget to call the original `start()` method. Classes extending the Thread class can have their own constructors, but **must not** have destructors; instead, they should do some cleanup, if they have to, before the end of their `run()` method.

The Thread class has a few more simple methods (far less then the Java Thread class, since it assumes that all the threads will execute on distinct processors; those missing relate to thread priority and scheduling on a machine that can have more threads then processors and those relating to management of threads in groups). The following can be called by the parent:

- `isAlive()` returns true if the thread is live and running, false otherwise.
- `waitForEnd()` returns after the child thread finishes, or immediately if it is dead already. You can guess that the `~Thread()` destructor implicitly calls this method.

The following can be called by the thread:

- `isParentWaiting()`: returns true if the parent is waiting for the child to die. As you can see, the parent does not have a means to explicitly kill a thread, as it can in Java; but asynchronous killing is never recommended anyway.

## D.3 The Monitor class

### D.3.1 Introduction

The previous section has dealt with independent, asynchronous threads. However, the key thing in multithreading is synchronization over data that can be used by more than one thread.

Basic synchronization still can be achieved with LOCK and UNLOCK. Yet the drawbacks of such approach are obvious. Therefore, the Java language designers decided to implement the concept of *Monitors*, first introduced by C. A. R. Hoare, back in 1978. As you already know, a monitor is a passive, shared object – shared in the sense that its data can be read and written to by multiple threads – whose methods that perform operations over the monitor's data are made to be atomic, i.e. no more than one thread can execute any of the monitor methods at the same time.

Consider the following class:

```
class SharedCounter {
    int value;
  public:
    SharedCounter() { value = 0; }
    void increment() {
        value = value + 1;
        }
    };
```

If two distinct threads tried to call the `increment()` method of the same instance of SharedCounter class, the resulting value could be either 1 or 2, though it, semantically, should always be 2 – the method is called twice. The first thread executing the code in the `increment()` method may find the "value" being 0, and so may the second, assumed that they both call `increment()`at the same time. The first will then write 1 to it, and so will the second. Obviously, the "value" is a shared variable, and access to it must be protected somehow.

This problem can be overcome with the use of *monitors*. The `increment()`function should be marked as *synchronized*, and while the first thread is executing it, the second will have to wait before even entering it (or any other synchronized method the class may have). So the proper SharedCounter class would look as follows:

```
class SharedCounter : public Monitor {
    int value;
  public:
    SharedCounter() { value = 0; }
    synchronized void increment() {
        value = value + 1;
        }
    };
```

The **Monitor** class, which is part of the system, implements monitors in Java-like manner. The class will be more described later in more details, but basically this is what it looks like using it. Actually, in the current implementation of the system described here, the programmer must manually place two calls at the beginning and the end of each method marked as **`synchronized`**. But it requires no thinking and planning as would be the case in manual protecting of critical sections with LOCK and UNLOCK, and this placing could be done with a preprocessor. In this simple example it may appear as a nearly equal hassle, but in any example other than this dummy one fiddling with LOCK and UNLOCK would make the code unreadable, particularly having in mind dealing with condition variables described later. In fact, all that LOCK and UNLOCK hassle is included in the Monitor class, and hidden from the programmer that uses it. Anyway, these two calls embracing the body of a **`synchronized`** method are shown here:

```
class SharedCounter : public Monitor {
    int value;
  public:
    SharedCounter() { value = 0; }
    synchronized void increment() {
        acquire();                  // acquire the monitor
        value = value + 1;
        release();                  // release the monitor
        }
    };
```

The "keyword" **synchronized** is a null macro, and is only intended to provide information in the specification of member functions of a class that is a Monitor class. Classic monitors have all their member functions implicitly synchronized, Java monitors do not, and neither do the ones described here. The reason is that not all the methods should always require exclusive ownership over the monitor – for example, some may only read monitor's data written at initialization time, and in such cases there's no need to block other threads that want to perform some reading and writing that can't interfere with such asynchronous read-only-written-once-data methods.

In short, each synchronized method first attains exclusive ownership over the monitor. If the monitor is already in use by some other thread, the thread calling a synchronized method (for the same monitor object, of course) will block. If not, the thread entering a synchronized method will mark the monitor object as "in use". Synchronization occurs only for one instance of monitor objects (for non-static member functions). If there existed two instances of the SharedCounterClass, say a and b, a thread executing a.increment() would not prevent another thread from executing b.increment() concurrently.

Java monitors are reentrant, and so are these. That means that a thread executing one synchronized method of a monitor *can* execute another synchronized method of the same object without having to leave the first synchronized method beforehand. There is an example in one of the javalike/ sub-directories that illustrates that.

Note that the term "a monitor object" actually means "an object which is an instance of a class that is derived from the Monitor class".

If a thread attempts to enter a synchronized routine and finds the monitor to be in use, it will block – waiting for the monitor to be released by the current owner. If more threads want to use the monitor at the same time, they will all block. Generally, monitors do not specify which one of them will be the first to unblock when the monitor is released, but this implementation ensures that it be the one who first attempted to gain the ownership.

## D.3.2 Conditions

While in simple examples declaring a method as "synchronized" suffices, in real life it does not. Consider the Producer-Consumer problem. A class describing the monitor object that would be used as the buffer for the two concurrent threads, Producer and Consumer, would have two synchronized methods: put() and get(); the first called by the Producer, the second by the Consumer. Since they are synchronized, they cannot be executed at the same time. If the Producer is executing the put(), the Consumer will have to wait before entering get(), until the Producer leaves the put() method, and vice versa.

But what happens if the Consumer, which enters the get() method and becomes the exclusive monitor owner, finds that the Producer has not prepared the new value yet? Or, conversely, if the Producer enters the put() and finds that the last value has not been consumed by the Consumer yet?

In such cases, synchronized methods can make use of another two monitor methods: **wait()** and **notify()**. If a synchronized method finds that a condition is not fulfilled, it will do a **wait()**, which temporarily releases the monitor and puts the calling thread to sleep. Such example is the get() method. If it discovers that the new value is not produced yet, it will do a wait(), release the monitor, and the calling thread (the Consumer) will go to sleep. The Producer (which could have possibly been waiting at the put() method) can now, or later, enter the put() method, since the monitor is free, write the new value, and do a **notify()**. Notify() does not relinquish the control over the monitor, it just awakes the *first* sleeping thread, which is waiting for a condition to fulfill, by notifying it that the condition is now fulfilled. When the producer leaves the put(), the awaken thread will regain the monitor ownership; or, rather, it will contend for the monitor ownership with other threads waiting to acquire the monitor (if there are such), and continue.

There is another notification method, **notifyAll()**. It will awake *all* the threads blocked at a **wait()**. However, it does not mean that all of them will acquire the monitor when the notifier leaves the synchronized function; they will only contend for the monitor.

Now let's see what it looks like in the Producer-Consumer example, taken from the mentioned Java tutorial:

```
/*
 * CubbyHole is a monitor object: producer writes to it and consumer reads
 * from it, with get() and put() methods, that require synchronization.
 */

class CubbyHole : public Monitor {
    int contents;    // value written to by the Producer and read by the Consumer
    bool available;  // indicates that the new value is available
  public:
    synchronized int get() {         // called by the Consumer
        acquire();
        while (available == false)
            wait();
        available = false;
        notify();    // notify the Producer that the object has been consumed
        int temp_contents = contents; // this is a workaround for the fact
        release();                    // that release() should be done before
        return temp_contents;         // return; in Java, release() is implicit
        }
    synchronized int put(int value) { // called by the Producer
        acquire();
        while (available == true)
            wait();
        contents = value;
        available = true;
        notify();    // notify the Consumer that the new value is available
        release();
        }
    };

/*
 * Producer and Consumer are threads, getting the address of a CubbyHole
 * object to work with upon initialization.
 */

class Producer : public Thread {
    CubbyHole *cubbyhole;
public:
    Producer(CubbyHole &c) {
        cubbyhole = &c;
        }
    void run() {
        for (int i=0; i < 10; ++i) {
            cubbyhole->put(i);
            printf("Producer put: %d\n", i);
            }
        }
    };

class Consumer : public Thread {
    CubbyHole *cubbyhole;
public:
    Consumer(CubbyHole *c) {
        cubbyhole = c;
        }
    void run() {
        int value;
        for (int i=0; i < 10; ++i) {
           value = cubbyhole->get();
           printf("Consumer got: %d\n", value);
           }
        }
    };

void main(int argc, char *argv[]) {
    CubbyHole *c = new CubbyHole;
    Producer *p1 = new Producer(c);
    Consumer *c1 = new Consumer(c);
    p1->start();
    c1->start();
    delete p1, c1;   // synchronous delete
    delete c;
    }
```

The wait() and notify*() methods operate over an *implicit* condition variable that every monitor has. However, to the author of this system it seemed inappropriate to have only one condition variable, for notify() could wake a thread waiting for a semantically different condition to be fulfilled than the one that is the subject of notification, though such a thread would check the condition of his concern, determine that it is not the one it is waiting for, and go back to sleep, but this is an unnecessary overhead. Therefore, explicit condition variables are supported, and they can be given as arguments to wait() and notify*(). (For example, the CubbyHole class could have `condition notfull, notempty;` declared in its declaration section, and later do `wait(notempty)` and `notify(notfull)`in get(), for example. The later would awake only those threads waiting at the `notfull` condition variable.

### D.3.3 Usage

So if you want your class to be a monitor class, simply derive it from the Monitor class. Mark all the methods that require synchronization as `synchronized`, and surround the body of such methods with `acquire()` and `release()`. Use the `wait()`, `notify()` and `notifyAll()` methods as the logic of your program requires. You can also use explicit `condition` variables if you have to.

As with threads, the same rule about allocation applies here: monitor objects are shared, they contain locks and stuff in it, and they must not be allocated on the stack. Either declare them as global variables, or use the operator **new** to allocate them on the heap. Deleting a monitor object will require that no one is currently using the monitor or waiting for a condition to fulfill.

## D.4 Compilation with Limes

This one is simple: just include the "**threadmon.h**" in your C++ application that uses threads and monitors, and make sure **threadmon.cc** appears in your makefile. It is best to customize one of the makefiles provided with the examples. Your `main()` function has to be declared as `void main(int argc, char *argv[])`.

As stated before, the ThreadMon package described here (that's its draft name, but it appears that the same name is used for some Java package) does not imply anything about the underlying system. All the ANL macros it uses are those for locks, and the CREATE macro. Therefore you can compile it with any hardware simulator you want – Berkeley, Dragon, ideal, your own, whichever. The same applies for the preferred synchronization policy – abstract, realistic, or some other. You can even choose ALGEVALONLY=1 option. Still, bear in mind that C++ standard library functions are not instrumented, so make your thread objects use the standard ones instead – if they need them at all.

Finally, so far it is only a draft version of the package; it lacks exception throwing and handling, and things like that. Still, it may be useful for exercising comfortable and neat parallel programming, in an environment where all the effects of true concurrent execution can be instantly noticed. So if you find a bug, or make improvements to the system, please let the author know.

# References

[Woo+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, Anoop Goopta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995

[Her93] Herrod, S.A., "TangoLite: Introduction and User's Guide", technical report, Stanford University, Stanford, USA, November 1993

[ToM92] Tomasevic, M., Milutinovic, V., "A Simulation Study of Snoopy Cache Coherence Protocols", Proceedings of the Hawaii International Conference on System Sciences, pp. 426-436, Koloa, Hawaii, USA, January 1992

[Katz85] Katz R., Eggers S., Wood D., Perkins C., Sheldon R., "Implementing a Cache Consistency Protocol", *Proceedings of the 12th ISCA*, 1985, pp.276-283.

[McCr84] McCreight E.M., "The Dragon Computer System, an Early Overview", In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, Urbino, Italy, July 1984.

[CaW96] Campione, M., Walrath, K., "The Java Tutorial", online version, ftp://ftp.javasoft.com/docs, July 1996.