- *-contig* - Request a contiguous set of COW nodes
- *-np 8* - Request 8 COW nodes
- *-time 10m* - Request use of our COW partition for up to 10 minutes

Here we're asking to run script "lam_srvr_script" on a COW server which is connected to a contiguous set of 8 COW nodes, and we're specifying that our job must complete within 10 minutes.

lam_srvr_script

```
#! /bin/csh -f
crsh -lam all lam_node_script
```

- *crsh* - run a remote shell on our COW nodes
- *-lam* - set up to use active messages
- *all* - run *lam_node_script* on each node in our COW partition
- *lam_node_script* - shell script to be run by the remote shell

Here we ask the COW server to run the script "lam_node_script" on all the nodes in our partition. We specify that we want the network setup for the "lam" protocol (active messages).

lam_node_script

```
#! /bin/csh -f
wwt2 -smp 2 -n 16 -ns 1 -ncpu 1 -hwstache mm
```

- wwt2 - binary to run on the COW nodes (the simulator)
- *-smp 2* - number of host CPU's to use on each host node
- *-n 16* - number of target CPU's to model
- *-ns 1* - number of target nodes to model on each host CPU
- *-ncpu 1* - number of target CPU's to model on each target node
- *-hwstache* - simulate a stache protocol running in hardware
- *mm* - the name of our target binary

This says to simulate 16 target machines, one on each of two host CPU's on 8 COW nodes. The other arguments to the simulator are the same as in the previous examples.

2. Use *csub* to submit a batch job on the COW.

[5] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, page ?, January 1994.

[6] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[7] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[8] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[9] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[10] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[11] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[12] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[13] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[14] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

Appendix A: Execution on the Wisconsin COW

The Wisconsin COW is a cluster of dual CPU sparcstations connected by a very low latency system area network (the *myrinet*). When running on the COW, you can choose to use it as a cluster of SMPs (using both CPUs) or a network of single CPU workstations (using only one CPU per sparcstation). The COW is controlled by the Distributed Job Manager (DJM), so to run jobs on it, you must issue DJM commands[1]. To run on the COW, the machine you launch the job from needs to be on the approved list. In addition the scripts described below are located in **`/p/cow/bin`**.

Note that these instructions apply only to the Wisconsin COW, which has a very specific hardware and software environment. Getting WWT-II to run on another cluster system would require some changes to the lower layers of the communication and synchronization package (SAM). Likewise, these instructions would need to be adapted to whatever scheduler is running on the cluster.

If you have access to the COW, you could use the following scripts and commands to run the matrix multiplication example:

Shell Command **`crun lam_srvr_script -contig -np 8 -time 10m`**

- *crun* - Command to run an interactive job on the COW[2]
- *lam_srvr_script* - Command to run on the COW server node

1. For details on submitting jobs to the COW, see the COW web page at **`http://www.cs.wisc.edu/~cow/djm.html`**

Note that the simulator and the target program are a single process from the point of view of the operating system. If your target program crashes, the simulator crashes with it, and a single *core* file will be produced. If that happens, you may want to examine target program's stack or data areas. You can do this easily by specifying the target executable rather than the simulator executable on the gdb command line. Your invocation would be something like

```
gdb target_program core
```

Unfortunately, current versions of gdb can not properly switch between the simulator and target program's symbol tables, so you cannot put breakpoints in the target code and run it with the simulator under the debugger.

8.2 Getting Repeatable Results

WWT-II schedules all target events (including communication events) at specific points in virtual time. The WWT-II host programs synchronize frequently enough to ensure that all target events occur in the correct order, and that causality is maintained at the target level. As a result, you can run the same benchmark many times, (including in different host execution environments), and always get *exactly* the same results. However, one must be aware of a few caveats regarding how target programs interact with their environment to obtain such consistent results. First, you should be aware that a program's command line arguments (including its pathname in argv[0]) are pushed onto the execution stack before the stack frame for the routine "main" is pushed onto the stack. This means that the size of the command line arguments and pathname will affect the alignment of every stack based variable which is used by the program. For example, if you have two identical copies of a program, one located at "/foo/bar/pgm", and the other located at "/tmp/pgm", their execution times would most likely vary somewhat due to the different alignment of stack variables, and the resulting cache effects. Similarly the program's environment variables are pushed onto the stack, and any minor difference here could cause perturbations in virtual running time. Note that these effects are not artifacts of the simulator; they reflect real differences in cache conflicts which occur in the target system being modeled.

8.3 Dependence on the myrinet

The *myrinet* is the only system area network supported by the current version of WWT-II, but since all the communication and synchronization is done in the SAM layer, it should not be difficult to get the simulator running on another network. The SAM layer as currently implemented uses the LAM package (Lanai version of Active Messages) from the *NOW* group at Berkeley. Other implementations on top of an active message layer would be especially easy to do. While it is not supported now, previous versions of SAM ran on top of an MPI layer, and an MPI implementation would also be pretty straightforward.

References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfieldand Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., 1987.

[3] Babak Falsafi and David A. Wood. Cost/Performance of a Parallel Computer Simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, July 1994.

[4] Babak Falsafi and David A. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, Feb 1997.

```
Breakpoint 1 at 0x931c0: file ../../../host_node.common.C, line
254.
(gdb) run -n 4 -ns 4 -hwstache mm.16
Wisconsin Wind Tunnel II
wwt2: Cluster of SMP version
Copyright (c) 1995-1996 Mark D. Hill, James R. Larus, and David A.
Wood. All rights reserved.
Authors: Babak Falsafi, Michael J. Litzkow, Shubhendu S. Mukher-
jee, and Steven K. Reinhardt

Breakpoint 1, HostNodeCommon::simulate (this=0x11dff0)
    at ../../../host_node.common.C:254
254    hostSimStartTime = hostTimer();
```

The simulator has run until it hit the breakpoint in the main processing loop. Now we can schedule our special event, set a breakpoint in the code that handles the event, and continue execution until we hit that breakpoint. The arguments to *schedDebugEvent* are *host_id* and *VT.*

```
(gdb) call schedDebugEvent( 0, 1000 );
schedDebugEvent: event scheduled...
   don't forget to set a breakpoint at DebugEvent::process()
(gdb) break DebugEvent::process
Breakpoint 2 at 0x4f264: file ../../../node.C, line 79.
(gdb) continue

Breakpoint 2, DebugEvent::process (this=0xd13d30) at ../../../
node.C:79
79    cout << "Debug event" << endl;
```

The simulator has now stopped in the processing of our special event. We are stopped in the execution of simulator code, and can examine the simulator's state directly. For example we can look at the simulator's stack. Notice that the low part of the stack shows functions at addresses above 0x90000000. These are target program routines (they have been relocated by *elsie*). The debugger cannot give much information about them because it only knows about the symbol table for the simulator. When the target program encounters an instruction that cannot be executed directly, such as a memory reference, it calls the simulator, hence you see target text addresses in the lower part of the stack, and simulator text addresses in the higher part. Simulator routines are called from target code via a jump table, which looks like a function call to the debugger.

```
(gdb) where
#0  DebugEvent::process (this=0xd13d30) at ../../../node.C:79
#1  0x8cc40 in Processor_counterExpiration (vt_ctr=0xd13d30)
    at ../../../node.H:198
#2  0x11f1b4 in jump_table ()
#3  0x91007064 in ?? ()
#4  0x9100783c in ?? ()
#5  0x91008500 in ?? ()
#6  0x9100edf4 in ?? ()
#7  0x91009f44 in ?? ()
#8  0x91002eec in ?? ()
```

8.1.1 Using WWT-II's built in tracing facility

WWT-II has an extensive built in tracing facility. Many types of events such as target program loads and stores or TLB misses can be traced. The tracing facility is controlled by three command line arguments to the simulator.

- *-pt* — Print trace information to standard out. Unless this flag is specified, trace records are stored in a buffer. If the simulator exits abnormally (via the *die* function), the buffer is dumped to standard out. Note that the buffer is of a fixed size, and space gets re-used if the buffer overflows, hence the trace buffer may only contain a small number of the most recent events.
- *-ts n* — Specifies the size of the trace buffer as *n* records.
- *-tr m* — Specifies the trace mask. Each type of event which can be traced is signified by a bit in the mask. If the bit is set in *m*, then events of that type will be traced. The bit values are specified in the file "trace.H".

The file "trace.H" contains the following two definitions

```
#define Trace_LdSt      TRACE_MASK(1)   /* 0x00002 */
#define Trace_Tlb       TRACE_MASK(3)   /* 0x00008 */
```

You can trace multiple types of events by or'ing their respective bit masks together. Hence the simulator invocation

```
wwt2 -n 4 -ns 4 -hwstache -tr 0xa -pt mm
```

specifies that all target loads and stores to memory as well as TLB misses should be traced, and those traces are to be printed directly to stdout, (rather than being saved in a buffer).

8.1.2 Running WWT-II under a general purpose debugger

Since WWT-II is a discrete event simulator, its major functions are the scheduling and processing of events. Each event is scheduled to occur at a given point in virtual time (VT). You can schedule a special event called a *debug event* for whatever VT you like. For instance, you might want to schedule such an event at a time just before your target program attempts an invalid memory reference. You could then examine the state of the simulator or target memory just prior to this calamitous event. To do this, you would need to get control of the simulator at a point when the event list has been initialized, but before event processing begins. You would then schedule a debug event for the VT of interest. Next, you would set a breakpoint in the code that will be executed when this event is processed. Finally, you would continue execution of the simulator, and wait for the breakpoint to be hit. The following annotated dialog shows how you could do this with the debugger *gdb.*

First we start up the debugger, set a breakpoint which will happen before any events are processed, but at a point where we can schedule an event, and start the simulator running. The simulator arguments shown will cause it to simulate an S-COMA system with 4 target nodes, where each target node has one CPU. The target program is called "mm.16".

```
padauk(mike) gdb chicago
(gdb) break HostNodeCommon::simulate
```

- *Typhoon* — Each target node is an SMP with a Typhoon accelerator board. The typhoon board has its own CPU, so general purpose CPUs are not used for protocol processing. Hence the *fixed* and *floating* policies don't apply.
- *hwstache* — S-COMA machine with fine grain access control provided in hardware. The general purpose CPUs are not used for protocol processing, so *fixed* and *floating* policies don't apply.

7.4 Selecting a target system at run time

When you run the simulator you must specify the target system you want to run on the command line. It is imperative that the target system you specify matches the target system for which you build your target program.

t0 — Specify *-asic* on the command line.

t1 — Specify *-t1* on the command line.

typhoon — Specify *-maskarb -typhoon* on the command line

hwstache — Specify *-hwstache* on the command line

For *t0* and *t1* systems you also need to decide whether you want *fixed* or *floating* protocol processing. This is done by linking in the appropriate library with your target program, as discussed in the last section. For *typhoon* and *hwstache* systems the general purpose processors are never used for protocol processing, so this consideration does not apply. You must also specify *-maskarb* for *typhoon* systems to mask bus arbitration.

By default each node in your target system will contain only one general purpose cpu. You can specify more cpus with the *-ncpu X* command line parameter. The number of cpus for *t0* and *t1* systems with fixed protocol processing *includes* the protocol processor. This is also true of target systems utilizing the *typhoon* processor, but does not apply to *hwstache* systems. For example, if you wanted two run 2 target program threads with a fixed protocol processing policy on a *t0*, *t1*, or *typhoon* system, you would specify *-ncpu 3* on the simulator command line. However, if you wanted to run 2 target program threads on a *hwstache* system, you should specify *-ncpu 2* on the command line. Finally, you must also specify the number of threads in the target program's portion of the command line with the *-nt X* argument. For this example you would specify *-nt 2* after the name of the target binary.

8 Bugs and Limitations

This version of the simulator is known to have a bug in simulating the *typhoon* systems. A livelock condition is sometimes encountered when running multi-threaded applications on *typhoon*.

8.1 Debugging the Simulator

If you need to debug the simulator (or if you just want to use debugging techniques to become more familiar with its operation), there are several ways to proceed. WWT-II has a built in tracing facility, special functions to assist in debugging with a general purpose debugger, and a mechanism for examining the target program's memory.

7 Building Your Own Target Programs

The *parmacs* directives used by WWT-II adhere quite closely to the standard described in [2]. Because of this, many of the standard benchmarks available from various sources in the parallel architecture community can be easily made to run on WWT-II. We suggest you start with the *Makefile* from the matrix multiplication example, if you want to develop your own target programs.

7.1 Structuring Your Target Program

You need to create your program as a set of **.U** (corresponding to **.c**) and **.H** (corresponding to **.h**) files in the C language. The following actions will be taken by the **Makefile**. The **.U** and **.H** files will be preprocessed by the **m4** macro processor and converted to the corresponding **.c** and **.h** files. A C compiler will convert these into object files, which will be linked by the linker **ld**. Finally, **elsie** will instrument the target binary to keep track of virtual time on the target machine. This process is illustrated in figure 2. The command **make** produces an executable ready to run on WWT-II.

7.2 Modifying the Example Makefile

Modifying the *Makefile* to build your own target programs is straightforward.

1. Set **WWT_ROOT** to the root of the WWT-II distribution at your site, (either by changing the *Makefile* or setting an environment variable).
2. Set **TARGET** to the name of the target binary you want to generate.
3. Set **SRC** to the list of *.U* files (corresponding to *.c* files) from which your target program is constructed.
4. Set **INC_SRC** to the list of *.H* files (corresponding to *.h* files) which are to be included in your *.U* files.
5. Use an **include** statement to pick the target machine for which you want to build your program. Target machine choices are discussed in the next section.

7.3 Choosing a Target System

You can use an **include** statement in your **Makefile** to pick the target machine for which you want to build your program. Target machine choices are:

- *t0_fix* — Each target node is an SMP with a *T0* accelerator board and one of the processors is dedicated to protocol processing.
- *t0_float* — Each target node is an SMP with a *T0* accelerator board and protocol processing is shared among the CPUs.
- *t1_fix* — Each target node is an SMP with a *T1* accelerator board and one of the processors is dedicated to protocol processing.
- *t1_float* — Each target node is an SMP with a *T1* accelerator board and protocol processing is shared among the CPUs.

COW. T1 and Typhoon exist as specifications for more complex and expensive accelerators which could be added to networks of workstations to aid in the implementation of a DSM system. All three boards can be simulated by WWT-II.

- T0 — provides a custom access control module per target node to aid in DSM operations.
- T1 — provides both a custom access control module and an integrated network interface.
- Typhoon — provides access control, an integrated network interface, and a custom protocol processor.

5.3 Hwstache

Hwstache is a simulation of an S-COMA machine, as proposed by Hagersten et al. The S-COMA target system utilizes a Stache-like allocation policy (i.e., pages in main memory) but provides hardware support for maintaining coherence on finer grain cache blocks.

5.4 Allocation of Protocol Processor

When simulating a target system that has multi-CPU nodes and does not have a dedicated protocol processor, e.g. *T0* or *T1* systems, you have a policy decision to make. You could dedicate one of the general purpose CPUs to protocol processing. We refer to this policy as the *fixed* protocol processing policy. Alternatively, you can use all the CPUs for target program execution, and allocate CPUs for protocol processing on the fly via an interrupt. We refer to this policy as the *floating* protocol processing policy. In the *fixed* case, you give up use of the CPU for target execution in exchange for low latency in protocol processing. In the *floating* case, you can use all the CPUs for target execution, but you pay a higher price for arbitration whenever a protocol event occurs[4].

6 Ready-To-Make Target Programs

To help you become familiar with the simulator, several benchmarks are provided with the WWT-II source distribution. Each of these benchmarks has its own directory under **`$(TOP)/benchmarks/`**. Those benchmarks which require input data files have sample files in their subdirectories. Each benchmark has a subdirectory called **src.mt**, which contains multi-threaded source code for the application. Each **src.mt** directory contains a **Makefile** which is set up to build a copy of the benchmark that will run on a *hwstache* (S-COMA) target system. You can also switch the **Makefiles** to build a target program for a different target system. See section 7.2 and section 7.3 for instructions on how to do this.

The benchmarks provided with the distribution are:

- FFT — A benchmark "kernel" which does fast Fourier transforms
- LU — A Blocked LU Decomposition
- RADIX — An Integer Radix Sort
- WATER-SP — A Hydrodynamics Simulation
- TOMCATV — A Program that generates a vectorized mesh

Fft, lu, radix, and water-sp are from the SPLASH-2 benchmarks.

```
item    max             min             ave
Fill    4.791660e+05    4.791580e+05    4.791628e+05
MM      1.185092e+07    1.106721e+07    1.169294e+07
Check   4.076710e+05    3.656770e+05    3.873071e+05

matrix multiply answer correct, max error = 0.000000e+00

Execution time for parallel section in cycles/1000 = 12767896
```

Note that *max, min, ave,* and *execution time* are *virtual* cycles on the target machine. You can examine the matrix multiplication source code to see how to generate timing statistics for your target program.

4.4.2 Output From WWT-II When Running the Matrix Multiplication Program

The statistics file, *WWT.stat.N*, contains a wealth of information about the execution of your program on the target machine. You should examine the sample WWT.stat.2429 file in **`$(TOP)/example/parallel`**. The following information is in the *WWT.stat.N* file:

- WWT-II execution parameters.
- Virtual time statistics and breakdown for computation, TLB misses, cache misses etc.
- Protocol dependent statistics
- Message counts and protocol transitions
- Cache statistics, like number of shared and private misses etc.

5 Description of Simulated Target Systems

WWT-II can simulate a wide range of target systems. To run on a particular target system you must first link your target binary with the correct target library, i.e. you must build a target program to run on the machine you want to simulate. When you run the simulator, you must also supply command line switches to tell the simulator what kind of target system you want to use. Note that the system you build your target binary for and the target system you specify on the simulator command line must match, otherwise your target program will probably crash.

5.1 Network Simulation

The network is assumed to transmit messages in 100 cycles. The simulator does have some capabilities to model contention for injecting and draining messages from the network. However, these capabilities are not complete and have been used differently in the different models. Those interested in the details should contact us.

5.2 Typhoon "family"

The typhoon family of accelerator boards provide various levels of hardware assistance in the implementation of Distributed Shared Memory (DSM) systems[11,12]. These boards are intended to be added to "off the shelf" workstations and networks. The T0 board exists as a real implementation which was done by two members of the WWT group. These boards are installed and working in the Wisconsin

- ***mm*** — the name of our target executable.

If your target executable requires any command line arguments, these can be typed after the name of the target executable.

4.3.3 Execution on a SMP

To run on an SMP, such as a SUN Enterprise server, you might type

```
wwt2 -smp 8 -n 16 -ns 2 -hwstache mm
```

- *-smp 8* — run on 8 host processors.
- *-n 16* — simulate a 16 processor target system.
- *-ns 2* — simulate 2 target processors on each host processor.
- *-hwstache* — simulate a target machine which runs a *stache* protocol in hardware (S-COMA).
- *mm* — the name of our target executable.

Note the *-smp 8* argument, by default WWT-II will query the system for the number of processors it has, and use them all.

4.4 Output From the Simulator and Target Programs

Note that in each of the above examples we have run the same simulator binary, the same target binary, and simulated the same target system. A feature of WWT-II is that we get the same output from both the simulator and target programs, *regardless of which host system we run on*. This allows us to utilize multiple host platforms to speed up the running of experiments, and simplifies debugging when porting the simulator to new platforms.

4.4.1 Output From the Matrix Multiplication Program

Following is the output you should get from the Matrix Multiplication program running on a target system of 16 single-CPU nodes using the hwstache coherence mechanism.

```
Wisconsin Wind Tunnel II
wwt2: Cluster of SMP version
Copyright (c) 1995-1996 Mark D. Hill, James R. Larus, and David A.
Wood.
             All rights reserved.

Authors: Babak Falsafi, Michael J. Litzkow,
         Shubhendu S. Mukherjee, and Steven K. Reinhardt


performing matrix multiplication on a matrix of order 128

Using stats.12837 for stats output.
```

commented out. These choices are discussed in section 5. Before attempting to build the executable, you need to set a couple of environment variables which the supplied *Makefiles* depend on. Set the environment variable **WWT_ROOT** to point to the root of the WWT-II source tree at your site. Also, set the environment variable **GNU_M4** to the pathname of the **GNU** version of the *m4* macro interpreter at your site (other versions of *m4* will not work properly). We suggest you do this in your login script, but you could also do it by editing your own copies of the *Makefiles* if you prefer.

The example *Makefile* in **`$(TOP)/example/parallel`** is set up to build the matrix multiplication program for running on a target system with a hardware stache. You should now build the example program by typing in

```
.../make[1]
```

4.3 Execution

How you will get WWT-II to execute (and run your target binary) will depend on the kind of host system you want to execute on. The WWT-II binary will run on an SMP-Cluster (a group of SMPs connected by a system area network). The only system area network supported at this time is the *myrinet*, but that does not mean you must have a *myrinet* to run the simulator. Both an SMP and a single-CPU workstation are degenerate cases of an SMP-Cluster. A single WWT-II binary supports all of these configurations. The normal location of the **`wwt2`** program is in **`$(TOP)/tzero/mbussim/parallel/cow/build.`** In what follows, when we refer to **`wwt2`** we are referring to this executable.

4.3.1 Command line options

WWT-II supports a large number of command line options. Only a few are illustrated in the examples which follow. To see a listing of all the options, you can run

```
wwt2 -help
```

4.3.2 Execution on a single CPU workstation

If your host is a single CPU workstation, you can run the sample matrix multiplication program by typing:

```
wwt2 -n 4 -ns 4 -hwstache mm
```

- wwt2 — the name of the WWT-II binary.
- ***`-n 4`*** — simulate a 4 processor target system.
- ***`-ns 4`*** — simulate 4 target processors on each host processor.
- ***`-hwstache`*** — simulate a target machine which runs a *stache* protocol in hardware (S-COMA). **The target architecture specified here must match the target library for which your program is linked.**

1. Be sure to use whatever pathname leads to the GNU version of *make* on your system.

4 Compiling and Running the Matrix Multiplication Program

There are a number of steps involved in preparing and executing your WWT-II target program (*benchmark*). This section explains the procedure using the matrix multiplication program as an example.

4.1 Overview

Your source code files will contain *parmacs* macros which must be expanded by *m4* in conjunction with the pre-defined set of *parmacs* macros provided in this software package. Next, your expanded source code is compiled and linked with the WWT-II user libraries. This target binary is then *instrumented* with instructions to keep track of virtual time and simulate memory references by a tool called *elsie* (also supplied with WWT-II). The process of preparing a target binary is illustrated in figure 2.

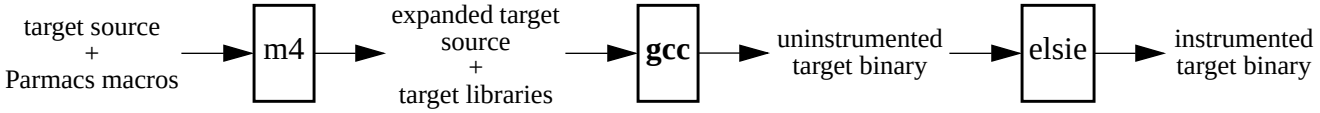

FIGURE 2. Target Program Preparation

Once you have prepared your instrumented binary, you can execute it on the simulator. Your instrumented binary and any required command line arguments, environment variables, and input data files all become input to the simulator. The output of the simulator is whatever output your target program produces, and a file containing statistics pertaining to your run. The contents of the statistics file are described in section 4.4.2. The process of executing your target binary is summarized in figure 3.

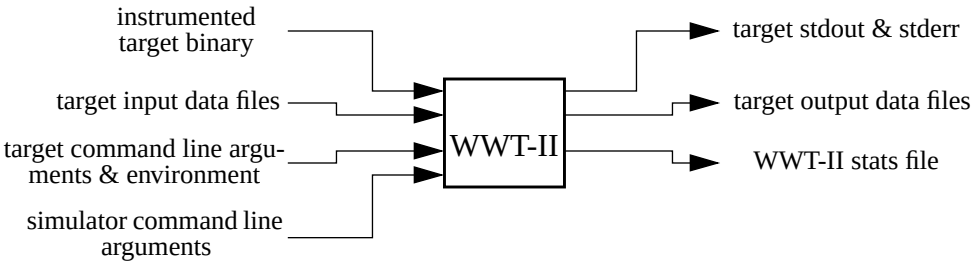

FIGURE 3. Simulator (and Target) Execution

4.2 Compilation

As explained above, preparing a WWT-II target program involves several steps. However, we have provided a set of *Makefiles* which automate this process for you. We use the *GNU make* **`include`** facility as well as other *GNU* specific features, so we assume you are familiar with *GNU make*, and can use it to build your WWT-II target programs. You should now examine the file **`$(TOP)/example/parallel/Makefile`**. You should use this file as a template for compiling your code for WWT-II.

We have filled in the *Makefile* for the matrix multiplication program in *parallel*. This *Makefile* builds a *hwstache* version of the matrix multiplication program. Other choices are listed in the *Makefile*, but are

TPPI_shm_spin(bool (*func(int)), int arg)

These calls perform a semaphore. The first version periodically calls **`func`** and only returns when the **`func`** returns **`true`**. The second version must be used if **`func`** accesses shared memory. The **`int`** provided in the call is passed to **`func`** on each call.

3.5.4 createThread(void(*func()))

This call is similar to **`CREATE(func)`** except that it spawns the function onto a single thread on the calling node. Since **`CREATE_ALL(func)`** only spawns one process per node, the function called is the one that typically spawns the other threads. Typical code would be:

```
CREATE_ALL(start);
start();
.....

void start()
{
   int thread;
   for (thread = 1; thread < TPPI_num_threads; thread++) {
       creatThread(driver_threaded);
   }
   driver_threaded();
}
```

3.5.5 void mark_pages_for_migration()

This function is declared in **`hwstache.h`**. It may be called once in the parallel section of the code. Once it is called a page will migrate to the first node that touches a value on that page.

3.5.6 void dumpStats(char *)

Causes the current statistics about the simulator to be dumped. The string passed is used as a label in the output produced.

3.6 Discussion of the Example Code

There are many ways to parallelize matrix multiplication. We have chosen to decompose the problem into computations of complete rows of the result matrix. Most of the code is straightforward. However, a couple of things require some explanation. The work scheduler allows dynamic load balancing by distributing work where there is demand. Tasks compute a row of the result matrix at a time, picking a row in order, starting at the row with the smallest index. A shared variable, **`I`**, records the lowest row number that still requires computation. An idle task reads **`I`**'s value and increments it. A lock ensures correctness by guaranteeing that these operations occur atomically. At the beginning of execution the matrices are filled with a set of known values which are unique, and non-zero. At the end of execution the answer is checked for correctness, based on the properties of the known values generated at the beginning. Both the initialization of the matrices and the check for correctness are done in parallel.

```
unsigned start_cycles, end_cycles, total_cycles;
...

main()
{
  INITENV();
  ...
  CLOCK(start_cycles);
  CREATE_ALL(...);
  ...
  WAIT_FOR_END();
  CLOCK(end_cycles);
  /* total_time = total number of cycles to execute the parallel
  section */
  total_cycles = (end_cycles - start_cycles) * 1000;
}
```

3.5 Multithreaded Applications and Other Issues

Although the example matrix multiplication code is not multithreaded, most of the benchmarks are programmed this way. Furthermore, several additional features are used. These are described in this section. Several of the calls are contained in the TPPI (Tempest Parallel Programming Interface) package. Codes using these routines need to include the **`tppi_thread.h`** header file. Having the word thread in the name can be somewhat confusing. The header file does contain some thread calls but also includes some calls used in unthreaded code. Codes using routines dealing with multithreading that are not in the TPPI package need to include the **`cyklos/interface.h`** header file.

3.5.1 IDs and Number of Threads

The external variable **`TPPI_num_threads`** holds the number of threads available on a node. The external variable **`TPPI_num_nodes`** holds the number of processors in the application. Using **`XX_NUM_NODES`** yields the same result. The external variable **`TPPI_self_address`** holds the processor number of the calling processor in the application. Using **`XX_NODE_NUM`** yields the same result.

The functions

```
int threadGlobalId()
int ThreadId()
```

return the thread id of the calling process. The first returns the id of the thread which is unique for all threads in an applications (across all **`TPPI_num_nodes`** processors). The second returns a thread id that is unique only for the node of the calling process.

3.5.2 void threadGlobalBarrier()
void threadBarrier()

`threadGlobalBarrier()` performs a barrier across all threads in the program. **`threadBarrier()`**, on the other hand, only performs a barrier across the threads on the node making the call.

3.5.3 TPPI_spin(bool (*func(int)), int arg)

```
test_function()
{
  LOCK(g->lock);
  g->I++;
  UNLOCK(g->lock);
  ...
}
```

3.4.8 void BARRIER(dummy, num_procs)

The directive **BARRIER**(*dummy*, *num_procs*) sets up a barrier, which holds back processors until *num_procs* - 1 other processors reach any **BARRIER** directive. There is, however, a caveat here. *num_procs* must be equal to **XX_NUM_NODES**. Or, in other words, you cannot have a partial barrier. *dummy* is a dummy argument for our purpose. You *do not* have to declare it as any variable. The original *Parmacs* specification requires the name of a barrier variable in its place. For example, you can synchronize all processors at a point in the following way:

```
ENV
...
main()
{
  INITENV();
  CREATE_ALL(test_function);
  test_function();
  WAIT_FOR_END();
  ...
}

test_function()
{
  ...
  BARRIER(dummy, XX_NUM_NODES);
  ...
}
```

BARDEC(*dummy*) and **BARINIT(*dummy*)** are provided for compatibility with other versions of *Parmacs*. In this version they are provided but are noops due to the above definition of a barrier.

3.4.9 Timing

void CLOCK(**unsigned cycles**)

A call to the macro **CLOCK**(cycles) returns the *virtual time*[1] in thousands of cycles in *cycles*. This macro can be used to find the virtual time at any point during execution. For example, you can measure the virtual time for the parallel section of your code in the following way:

```
ENV
```

1. The virtual time is the number of simulated target machine cycles since the beginning of execution. It is unaffected by how fast WWT-II is running on its host.

```
{
  INITENV();
  A = (int *)G_MALLOC(sizeof(int) * XX_NUM_NODES);
  set_fork_semantics( &A,sizeof(A) );
  ...
  CREATE_ALL(Update);
  Update();
  ...
  WAIT_FOR_END(XX_NUM_NODES - 1);
  ...
}
Update()
{
  A[XX_NODE_NUMBER] = 0;
  ...
}
```

3.4.7 Locks

- **void LOCKDEC(***lock***)**
- **void LOCKINIT(***lock***)**
- **void LOCK(***lock***)**
- **void UNLOCK(***lock***)**
- **void ALOCKDEC(***lock, number***)**

WWT-II implements both *MCS locks*[7] and message locks. Declare a shared lock variable *lock*, using **LOCKDEC**(*lock*). Initialize it using **LOCKINIT**(*lock*). Use the directives **LOCK**(*lock*) and **UNLOCK**(*lock*) to lock and unlock, respectively, the variable *lock*. **ALOCKINIT(***lock, number***)** is similar to **LOCK-DEC**(*lock*) except it creates an array of locks of length *number*. For example, you can atomically increment a shared variable, **I**, in the following way:

```
ENV
struct GlobalSpace
{
  int I;
  LOCKDEC(lock);
} *g;

main()
{
  INITENV();
  g = (struct GlobalSpace *)G_MALLOC(sizeof(struct GlobalSpace));
  g->I = 0;
  LOCKINIT(g->lock);
  set_fork_semantics( &g, sizeof(g) );
  CREATE_ALL(test_function);
  test_function();
  WAIT_FOR_END();
  ...
}
```

```
shared_var = (int *)G_MALLOC(sizeof(int))
set_fork_semantics(&shared_var,sizeof(shared_var));
```

Note the use of **set_fork_semantics**, which provides each slave process with a copy of the *pointer* to the allocated space. **G_MALLOC** guarantees that the address it delivers is aligned on a cache block boundary. *Parmacs* does not define a directive to free up shared space.

3.4.5 void CREATE_ALL(void (*func())) void WAIT_FOR_END(int N)

In the *Parmacs* world, only one task is active (on processor zero) when the parallel program starts. We will call this task the *master.* The directive **CREATE_ALL**[1]activates the other tasks (one on each of the other processors), which we call the *slaves*. This directive is passed the name of a function, *func*, that will be executed in parallel by the *slaves*. The call to **CREATE_ALL** returns immediately.

Often a program requires a post-processing phase, which cannot operate in parallel and cannot proceed until all parallel tasks complete. The directive **WAIT_FOR_END** synchronizes **N** *slaves* and deactivates them as soon as they return from *func*. When **WAIT_FOR_END** returns, only the *master* is active. This directive should only be invoked by the *master.*

Here is an example from **parallel/mm.U**, where the *master* creates a process to execute the function **Driver** on each of the other processors. **CREATE_ALL(Driver)** returns immediately and the *master* does its share of the work by calling **Driver**. At the end, the *master* synchronizes the *slaves* by calling **WAIT_FOR_END** and then prints the output.

```
CREATE_ALL(Driver);
Driver();
WAIT_FOR_END(XX_NUM_NODES - 1);
printf(...);
...
```

3.4.6 const int XX_NUM_NODES const int XX_NODE_NUMBER

The constant **XX_NUM_NODES** denotes the number of processors available to the parallel program. The constant **XX_NODE_NUMBER** denotes the processor id on which a process is running, where

$$\mathbf{0} \leq XX\_\mathbf{NODE\_NUMBER} < XX\_\mathbf{NUM\_NODES}$$

A process created by the macro **CREATE_ALL** can read the processor id from **XX_NODE_NUMBER** and decide to do its own share of work. For example, parallel update of a linear array **A** of size **XX_NUM_NODES** can be done by the following code:

```
ENV
int *A;
main()
```

1. The *Parmacs* model supports a different macro, **CREATE**, that creates one process on one processor. Hence, multiple calls to **CREATE** are necessary to fork multiple tasks, one on each processor. Although this macro is supported on WWT-II, you should use **CREATE_ALL** instead of **CREATE** because forking one task at a time is inefficient.

rect usage of the shared memory. Also a timing function is provided (see section 3.4.9). In many programs the master task will have some work to do which can be completed *only* after all the slave tasks have exited (for example, printing out an answer). Once the master task has finished executing the parallel function, it should use the **WAIT_FOR_END** *parmacs* directive (described in section 3.4.5). The call to **WAIT_FOR_END** will not return until all the slave tasks have terminated.

3.4 Parmacs directives supported by WWT-II

The *Parmacs* directives supported by WWT-II are described in detail below. Note that the syntax of some of these directives are slightly different from the standard *Parmacs* specification.

3.4.1 ENV: declaration

ENV declares various data structures required by the WWT-II to support the *Parmacs* model. These data structures *must* be visible to all the non-header files. For example, at the beginning of the **parallel/mm.U** file you will find the following statements:

```
/* mm.U - matrix multiply */
/* Setup environment for WWT-II */
ENV
```

3.4.2 void INITENV()

INITENV initializes the data structures declared by the **ENV** statement, and must execute before any other non-declarative *Parmacs* directives. See section 3.4.6 for an example of its usage.

3.4.3 void set_fork_semantics(void *addr, unsigned len)

Set_fork_semantics causes the contents of memory at the specified addresses to be broadcast from the master process to all the slave processes when they are created by **CREATE_ALL** (explained in section 3.4.5). Note that this does *not* cause the memory to become shared memory, it only initializes the memory in the slave processes with the current values from the master process. This is similar to the semantics of the *UNIX*[1] **fork** system call. This routine *cannot* be used to broadcast values stored in memory, which the master process has allocated on the heap; it can only be used for memory in the *initialized data* and *BSS* sections. Also note that all calls to **set_fork_semantics** *must* be done before **CREATE_ALL** is called.

3.4.4 void *G_MALLOC(sh_mem)

Unless otherwise specified, all data are allocated in memory private to the allocating task created by **CREATE_ALL** (explained later), and inaccessible to all other tasks. If some data needs to be shared, the only option is to allocate shared memory using the directive **G_MALLOC**. The **G_MALLOC** directive has the same interface as the standard Unix library call **malloc**. For example, you can allocate shared space for an integer variable **shared_var** as

1. *UNIX*® is a trademark of UNIX System Laboratories, Inc.

3.1 A Sequential Version

You will find a simple program in the **`$(TOP)/example/sequential`** directory that implements multiplication of two square matrices. You should examine, compile, and execute it. A *Makefile* is provided for this purpose.

3.2 A Parallel Version

When preparing a target program to run on WWT-II, you must parallelize your code by hand. The *Parmacs* directives allow you to do this. The *Parmacs* directives are a set of macros that provides constructs for writing parallel programs. We have provided a parallel version of the matrix multiplication program in **`$(TOP)/example/parallel`**. The rest of this section explains the Parmacs programming model and how the matrix multiplication program has been parallelized using that model. Section 4 describes how to compile and run it.

3.3 The Parmacs programming model

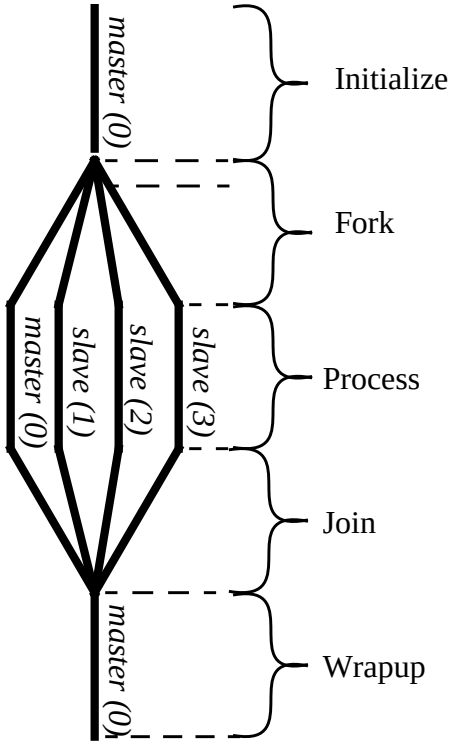

FIGURE 1. Parmacs Processing Model

Currently, WWT-II supports only the *Single Program Multiple Data (SPMD)* paradigm. Loosely speaking, in this paradigm all tasks execute the same code on different portions of the data. This model is illustrated in figure 1. During the *Initialization* phase only one task, (the master task), is active on one node (node zero) of the target machine. This task is responsible for any initialization which must be done before the parallel section begins. Typically this task will allocate memory which will be shared with the slave processes during the parallel section. The allocation of shared memory is done with the **`G_MALLOC`** *parmacs* directive (described in section section 3.4.4). Once the initialization has been completed, the master task can create the slaves. This is done by the **`CREATE_ALL`** *parmacs* directive (described in section section 3.4.5). The slave tasks are given a pointer to a function which they will execute in parallel. When an instance of that function returns, the slave which was executing it terminates. Usually the master task will call the same function, so that all tasks execute the same code during the *Parallel* phase. (The master task can execute different code from the slaves during the *Parallel* phase, if your algorithm is structured that way.) Both the master and slave tasks can use the **`XX_NUM_NODES`** and **`XX_NODE_NUMBER`** directives to determine the number of parallel tasks and their own task number (see section 3.4.6). The slave tasks get private copies of the initialized data and BSS sections and shared copies of any memory allocated by the master task during the *Initialization* phase. Note that, by default, the copies of the initialized data and BSS sections passed to the slave tasks are as they were at process startup time. Changes made to those sections by the master task will be copied to the slave tasks' data areas *only* if those areas are specified with the **`set_fork_semantics`** directive (see section 3.4.3). Note that this is different from the standard *parmacs* model. During execution of the parallel phase the tasks can use locks (section 3.4.7) and barriers(section 3.4.8) to synchronize and ensure cor-

2 General Description of the Simulator

WWT-II is a discrete event, parallel architecture simulator, which uses both direct execution and parallel computation to speed up its operation. Because of these two mechanisms, it is practical to simulate large target systems running realistic workloads.

In direct execution [10], a program from the system under study (the *target*) runs on an existing system (the *host*). For example, a target's floating-point multiplication executes as a floating-point multiplication instruction on the host. The host calculates the target's execution time and only simulates operations unavailable on the host.

Parallel simulation of a parallel computer further speeds simulation by exploiting the parallelism inherent in the target parallel computer and the parallel host's large memory to hold the simulator's working set and reduce paging. The advent of low-cost parallel computers, such as symmetric multiprocessors (SMPs) and clusters of workstations (COWs), make parallel simulation very attractive[3].

2.1 Supported Host Environments

WWT-II runs in a variety of environments, all members of SUN's family of SPARC based machines. These machines differ widely in the communication and synchronization mechanisms which are available for processors needing to communicate. We deal with this variation in communication and synchronization in a layer called Synchronized Active Messages (*SAM*). *SAM* isolates higher levels of the simulator code from various implementations of the messaging and synchronization layer, and provides a uniform interface to these services. *SAM* implementations exist for clusters of workstations on a local area network, processors in an SMP which can share memory, and combinations of the two (SMP clusters).

2.2 Supported Target Environments

WWT-II can model systems comprised of SPARC-like CPUs with various bus and network models. Three different hardware accelerators based on the Typhoon model are also available. Target programs are written in a dialect of PARMACS[2], which is used in the SPLASH and SPLASH-II benchmarks. Supported target environments include:

- A S-COMA [5] like system with fine grain access control implemented in hardware
- Hardware assisted shared memory based on the Typhoon family of accelerators (T0, T1, and Typhoon)
- The Tempest interface, which includes active messages.

2.3 Supported Target Interfaces

- PARMACS - This interface is described in section 3.
- Tempest Interface[9] - please see the referenced paper for details on this interface.

3 An Example Target Program - Matrix Multiplication

This section describes the features of a parallel target program using matrix multiplication as an example. To simplify the discussion, we will assume that the WWT-II distribution tree has been installed in its entirety at your site. We refer to the top level directory of this tree as **`$(TOP)`**. Please see the *WWT-II Installation Guide* for details on the directory layout.

tion. Please see the WARTS WEB page at *http://www.cs.wisc.edu/~larus/warts.html*, or send email to *warts@cs.wisc.edu* for more details.

1.2 Finding Other Documents

WWT-II and its precursor the Wisconsin Wind Tunnel [10] (WWT) have been described in various research papers and reports. Similarly the authors of the accompanying benchmarks have described their work in well known publications. This report doesn't attempt to duplicate that information, but only to fill in the gaps for beginning users of the simulator. This section tells how to get hold of other relevant documents.

1.2.1 WWT Project WEB Pages

The Wisconsin Wind Tunnel home page is at *http://www.cs.wisc.edu/~wwt/*. There you will find an *annotated bibliography* containing brief descriptions of all the papers produced by our group. All of the papers are available on-line by following the link called *WWT Technical Papers*.

- *Wisconsin Wind Tunnel* - papers on both WWT and WWT-II.
- *Tools* - papers on *EEL*, an executable editing library used to prepare benchmarks for use with the simulator.
- *Tempest, Typhoon, Blizzard* - Descriptions of the Tempest substrate and Typhoon family of hardware accelerators.

1.2.2 SPLASH and SPLASH-2 Documents

Several of the accompanying user programs are from the SPLASH-2 [13] benchmarks. The SPLASH home page is at *http://www-flash.stanford.edu/apps/SPLASH/*. Here you will find information on the accompanying benchmarks, *fft, lu, radix,* and *barnes.*

1.2.3 Berkeley Active Messages

One of the systems WWT-II can run on is a network of Sun workstations connected by a Myricom[1] system area network. Our implementation is built on top of the Berkeley Active Message [14] layer for this network.

- *http://now.cs.berkeley.edu/AM* - Description of the Active Message Layer
- *http://www.myri.com/* - Home site of Myricom, makers of the Myrinet[1]

1.3 Contacting WWT Personnel

To request help with installation and use of WWT-II or to report bugs, send email to *wwt@cs.wisc.edu*.

1. Myrinet® is a trade name of Myricom Inc.

1 Introduction

The Wisconsin Wind Tunnel II [8], (WWT-II) is a parallel, discrete-event driven, direct execution simulator for parallel architectures. WWT-II runs across a family of SPARC[1]-based machines, including single- and multi-CPU workstations, clusters of workstations connected by a system area network, and Sun SMPs. Supported target architectures include software- and hardware-based distributed shared memory (DSM) systems, and a family of hardware accelerators collectively known as "Typhoon". WWT-II also supports the Tempest Interface [6,9]which includes message passing via Active Messages[14].

WWT-II was written in the Computer Sciences Department at the University of Wisconsin - Madison. Its principal authors are Steven K. Reinhardt, Babak Falsafi, Shubhendu S. Mukherjee, and Michael Litzkow. Rob Pfile developed code for the memory bus modules.

This document describes WWT-II from the viewpoint of a simulator user. Topics covered include choosing a target architecture for modeling, building and running the accompanying benchmarks, and running the simulator in various environments. A separate document, *Wisconsin Wind Tunnel II: Installation Guide,* covers building and installing this software. This document, the *Installation Guide,* and instructions on getting the WWT-II distribution are available at *http://www.cs.wisc.edu/~wwt/wwt2.*

The remainder of this section explains the WWT-II licensing requirements, tells how to find related documents, and how to contact us if necessary. Section 2 gives a general description of the simulator. Section 3 introduces an example program (matrix multiplication), and describes the target programming model supported by the simulator. Section 4 explains how to compile and run the example program. Section 5 gives details about the target systems you can simulate. Section 6 describes some benchmark programs which are packaged with the simulator, and section 7 gives some tips on preparing your own target programs. Section 8 explains how to debug the simulator, and lists some of its limitations. Appendix A gives details on how to run the simulator on the Wisconsin COW.

1.1 Copyright and Licensing

WWT-II is part of the Wisconsin Architectural Research Tool Set (WARTS). WARTS is available without charge for university researchers and is available to other researchers for a modest research dona-

1. SPARC® is a registered trademark of SPARC International, Inc.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF Grants CCR-9101035, MIP-9225097, and MIPS-9625558, NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, DOE Grant DE-FG02-93ER25176, University of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from Digital Equipment Corporation, IBM, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

Wisconsin Wind Tunnel II: User's Guide

Mike Litzkow and Steve Huss-Lederman

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706-1685 USA
URL: http://www.cs.wisc.edu/~wwt