Molasses: An Ultracomputer Simulator (Preliminary Version)

Jan Edler

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #189
March 8, 1994

Abstract

Molasses is a simulator for a complete Ultracomputer system, containing processors, memory modules, and a multi-stage interconnection network supporting combining. The simulated machine is quite close to the Ultra-3 prototype. A large number of parameters may be set by the user to alter the characteristics of the simulated machine and the data to be collected.

Molasses relies on careful implementation, parallelism, checkpoint/restart, and specialized modes of simulation to achieve high performance. Performance may be maximized by automatically building a custom version of Molasses with most parameters fixed to constant values; this allows the compiler to generate superior code, typically doubling performance.

Although the preliminary version of Molasses does not yet simulate the Ultra-3 processors, its good performance and flexibility have already made it useful for network studies.

1. Introduction

A central topic in computer systems design is how to achieve scalability in general purpose multiprocessors. This encompasses every aspect of computer science, including theory, hardware, architecture, operating system, programming language, and algorithm design. Ultimately, the evaluation of scalable multiprocessing solutions must rest upon the implementation and use of complete large scale systems, but there is a cart and horse problem: we don't really know the best way to do it yet. Development of such large scale prototypes is very expensive, moreover the design space is too large to systematically explore with full implementations. As a result of this quandary, many different approaches are being taken to explore the design space, including analysis, simulation, and both small and large scale prototype construction.

We seek to expand the role of simulation as a tool in exploring scalability of shared memory multiprocessors. It is now possible to simulate systems of this type, with thousands of processing elements (PEs), more accurately and extensively than ever before. Key to the viability of this approach are recent advances in hardware that have dramatically improved performance and reduced the cost of computing.

Molasses is a simulator for a complete Ultracomputer system (Gottlieb [3]). Figure 1 shows a high level view of a 16 processor Ultracomputer, like the Ultra-3 prototype (Bianchini *et al.* [1]). As in a real Ultracomputer, there are three major components to the

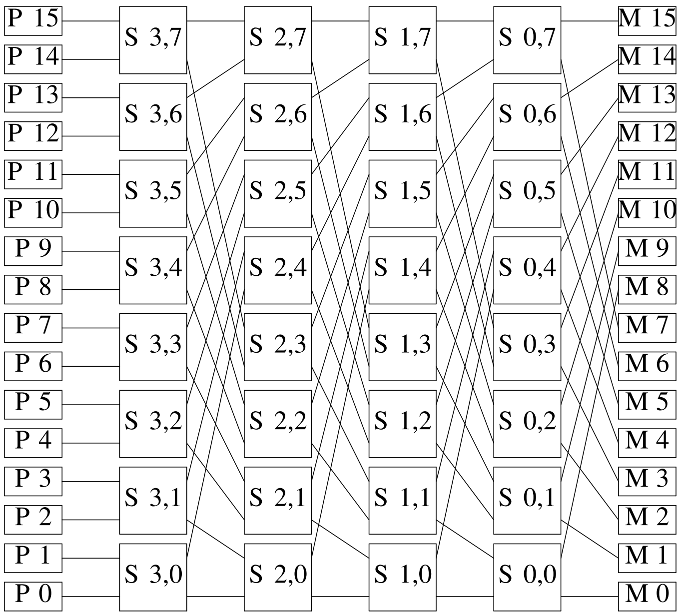

Figure 1: A 16 Processor Ultracomputer. Processing Elements are marked with P, Memory Modules are marked with M, and Switches are marked with S. I/O devices are not shown, but can be connected to either the PEs or to the MMs.

simulator:

- *The processing elements (PEs).* The current version of Molasses actually encompasses three different simulators, only one of which may be used at a time: A very faithful simulator of the PE in the Ultra-3 prototype, an address pattern generator, and an interactive mode, in which the user is prompted for a function and address whenever a PE is able to make a memory reference.
- *The interconnection network.* The network is composed of $N \times N$ crossbar switches, connected to form an Ω network.
- *The memory modules (MMs).*

Each component supports a large number of settable parameters to alter the characteristics of the simulated machine and the data to be collected. Performance may be maximized by automatically building a custom version of the simulator with all of those parameters fixed to constant values; this allows an optimizing compiler to generate superior code, typically doubling performance. For this reason, access to the source code of Molasses is essential.

Because Molasses is supposed to be an accurate whole-system simulator, it simulates each target machine cycle individually. Molasses uses the following techniques to improve performance:

(1) *Careful programming.* The components of Molasses were written to work together, with efficiency in mind; they were not pasted together using parts written for other projects.

(2) *Parallelism.* Molasses is written to run on MIMD shared memory multiprocessors. The granularity of parallelism is the component cycle, i.e., a single cycle of a PE, switch, or MM is executed serially. A simple dataflow model is used to synchronize different components, so all components need not be simulating the same cycle at the same time. A large machine simulation has lots of parallelism, for example a 1024 PE machine has 1024 PEs, 1024 MMs, and 5120 switches[1], all of which can be simulated in parallel.

(3) *Custom Compilation.* A command line option causes the entire simulator to be recompiled with many user-settable parameters set to constant values. The actual performance increase depends on the exact combination of parameters, but is typically about a factor of two. Almost all parameters described in section 2.1 are subject to ‘‘constantizing’’; the main exceptions are those controlling the pseudo-random address generator (§2.1.4.1).

(4) *Checkpoint and Restart.* The state of a simulation can be preserved and resumed later. This allows costly warm-up or initialization sequences to be simulated once, and resumed many times with different simulation parameters or user inputs.

(5) *Modes of Simulation.* Molasses components offer many parameters that affect simulation accuracy and speed. More specifically, however, Molasses offers two special modes of simulation:

    - A mode that skips the switches and MMs, permitting single-cycle conflict-free access to simulated memory.

[1] Assuming 2×2 switches, that’s 10 stages of 512 switches each.

- A mode that focuses on a single PE reducing overhead to simple serial instruction interpretation.

These options are particularly useful in combination with the checkpoint/restart feature; e.g., serial initialization code can execute on one simulated PE with single-cycle access to memory.

(6) *Unrealistic extensions.* Being a software simulator, Molasses can offer features a real machine couldn't. Sometimes these can improve performance, instrumentation, or user convenience. For example, new instructions allow selection of some of the above-mentioned modes under control of the simulated program. Other new instructions can provide access to system calls on the host computer, to use for formatted I/O, for example. Simpler-than-real-life I/O devices can simplify implementation and improve performance when simulating an operating system.

2. Using Molasses

It is important to understand the network stage numbering notation used by Molasses. The MMs are considered to be stage −1. The switch stage closest to the MMs is stage 0. Switch stages closer to the PEs are progressively higher. The PEs are sometimes referred to as having a stage number 1 higher than their connected switches.

Molasses has two kinds of output, ''logging'' output and ''statistics'' output. Options beginning with `-log-` produce output to the log file, and options beginning with `-stat-` produce output to the statistics file. By default, both of these ''files'' are connected to the standard output.

2.1. Options

Molasses is controlled by command line arguments. Because there are so many command line options, long names are used. The most important one to remember is `-help`, which gives a synopsis of all available options. The exact set of options available can vary, not only on account of future modification or extension of Molasses, but also due to the customization feature, which creates a new executable with many fewer options; `-help` will tell you just which options are still available. A summary of all options and their values is included in both the statistics and log files.

There are several generic types of options, but the main distinctions are boolean (with no argument), integer (with one integral argument), float (with one floating point argument), and string (with one string argument). The integer options generally accept decimal, octal (with leading `0`) and hexadecimal (with leading `0x`). In appropriate cases, a trailing `k` or `K` multiplies by 2^{10}, and a trailing `m` or `M` multiplies by 2^{20}.

Default values for options have been chosen to correspond with the Ultra-3 prototype whenever relevant.

Options are organized into the following groups:

- *General options.* These control the overall structure of the simulated machine, statistics collection, log output, checkpoint restart, and custom simulator construction.
- *MM options.* These control the structure and timing behavior of the simulated memory modules.
- *Switch options.* These control structural parameters of network switches.

- *General PE options.* These control general PE structure, statistics collection, and allow selection of the major mode of simulation, thus allowing use of the next three groups of specific options.
- *Interactive mode.* This mode is so simple that there are currently no options specific to it, however the possibility exists to add them to a future version of Molasses.
- *Address pattern generation.* These control parameters of the pseudo-random address pattern generator. As described in section 3.2, this portion of the simulator may be easily replaced by the user, including the exact set of options supported.
- *Faithful PE mode.* These control variations on the basic Ultra-3 PE model supported.

All these options are described more fully in the following subsections.

With as many tunable parameters as Molasses has, there can be many invalid combinations. Molasses checks for these at startup time, printing an error message such as the following:

```
molasses: maxcomb, nswports, and rqinputs are invalid parameters:
  ((2+1)*2) % 4 != 0
```

which means that `-maxcomb 2 -nswports 2 -rqinputs 4` is an unacceptable combination.

2.1.1. General Options. Options in this section pertain to the simulation itself, rather than the simulated machine, or they modify parameters that affect all major components of Molasses.

`-help` Produces a synopsis of all available options. No simulation is done; Molasses terminates after printing the synopsis to the standard output.

`-numpes` *n*

Sets the number of simulated PEs (and MMs) to n. The default is 1. The value specified must be a power of 2.

`-nswports` *n*

Sets the number of bidirectional switch ports in each direction to n. The default is 2, for a network of 2×2 switches. The value specified must be a power of 2.

`-cyalign` *n*

Sets the cycle alignment of messages to n. The first part of a message may only be transmitted in cycle c by stage s in direction d when

$$s \bmod n = \begin{cases} n-1-((c-1) \bmod n) & \text{if } d = \mathit{toward\ MMs} \\ (c+frskew-1) \bmod n & \text{if } d = \mathit{toward\ PEs} \end{cases}$$

where $frskew$ is described next. The default value for cyalign is 2. The value specified must be a positive integer.

`-frskew` *n*

Sets the cycle alignment offset for the return path to n. The role of frskew is explained under the `-cyalign` option. The default is 0. The value specified must be a non-negative integer.

`-fmulticts`
`-rmulticts`

These boolean options alter the handshaking protocol between ports on the forward and return path, respectively, so that multiple clear-to-send signals are used (Liu [4]). The default is to use one clear-to-send signal per port. Clear-to-send is a signal presented by the receiver to indicate whether or not it is able to accept a packet during the current cycle. Multiple clear-to-send signals indicate acceptability of packets destined for each of the receiver's output ports towards the next stage. For example, with $N \times N$ switches, each port presents N CTS bits. In the forward direction, a switch at stage i is separately indicating acceptability of packets destined for each of its N output ports sending to stage $i-1$.

`-mmbytes` *n*
: Set the size of each MM to *n* bytes. The default is 1 kilobyte (0x400). The value specified must be a power of 2.

`-mminterleave` *n*
: Memory is interleaved with *n* consecutive bytes stored in each MM in sequence. For example, with the default value of 8, bytes 0–7 are stored in MM0, 8–15 are stored in MM1, etc. The value *n* must be a power of 2 and must be at least 8 (the number of bytes in two words; this restriction is required to support the atomic load-double and store-double operations).

`-no-real-mem`
: This boolean option disables simulation of actual memory. The only effect is that stored values are discarded at the MMs, and returned values are garbage. The option is useful with synthetic workloads, where the values in memory are unimportant. The obvious advantage is saving significant virtual memory on the computer running Molasses.

`-pkt_bit` *n*
: Set the message packet size in bits to *n*. The default is 32. The value specified may be any positive integer; it doesn't need to be realizable. Message routing is not affected by this parameter: the first packet is always assumed to carry enough information for routing, even in the degenerate case of `-pkt_bit 1`.

`-verbose` *n*
: Set general verbosity level to *n*. The default is 0. Larger values are more verbose. Currently, the effect is limited to the `-custom` option, where larger values cause compilation information to be displayed.

`-debug` *n*
: Set general debug level to *n*. The default is 1, which is fairly quiet. A value of 0 is better for performance, especially with `-custom`. Higher values may produce some useful information.

`-custom` *f*
: Produce a customized version of Molasses in file *f*, and execute it. The environment variable `MOLASSES` gives the directory containing source code. The recompilation is accomplished by executing the shell script `makeit` in that directory. The shell script is responsible for figuring out how to invoke `make`. Try `-verbose 3` to learn more about how this all works.

`-restore`
: Restore simulation state. This option is not yet fully implemented.

`-statfile` *f*
: Set the file for statistics output to *f*. The default is the standard output.

`-stat-cycles` *n*
: Produce statistics for every *n* cycle interval. The default is 0, which means to produce statistics only at the end of the simulation.

`-stat-totreq`
: Compute the total number of requests arriving at each stage.

`-stat-portblock`
: Compute the total number of times output was blocked for each stage.

`-stat-latency`
: Compute message latency statistics. Round trip, forward, and return path mean and standard deviation are produced.

`-logfile` *f*
: Set the file for logging output to *f*. The default is the standard output.

`-log-connect`
: Print port connection information. This shows exactly which PE is connect to which switch, which switch is connected to which switch, and which switch is connected to which MM.

`-log-summary`
: Print a cycle-by-cycle summary of messages coming and going per switch per cycle. Combining information is also included. This data has been used to produce animations of network activity, using IBM DataExplorer.

2.1.2. MM Options. Some parameters that you might expect to be MM options are described under general options instead; this is because they actually affect more than just the MM. For example, the MM size and interleaving affect global routing. The remaining MM options are:

`-mmdelay` *n*
: The delay before responding to a request is set to *n* cycles. That is, if the first packet of a request arrives at an MM in cycle *t*, the first packet of the response won't be sent until at least cycle $t+n$. This affects most memory requests; some exceptions are provided by special options, described below, which can override `-mmdelay` for certain operations. The default is 2. Any positive integer may be specified for `-mmdelay` or any derivative option.

`-mmbusy` *n*
: The busy time before another request can be accepted is *n* cycles. That is, if the first packet of a request arrives at an MM in cycle *t*, the first packet of another message won't be accepted until at least cycle $t+n$. The default is 4. This affects most memory requests; some exceptions are provided by special options, described below, which can override `-mmbusy` for certain operations. Also, note that messages composed of more than *n* packets will automatically have a longer busy time. Any positive integer may be specified for `-mmbusy` or any derivative option.

`-mmdelayloadd` *n*
: The delay before responding to a load-double request is *n*. The default is 2. This

overrides the value of `-mmdelay`.

`-mmdelaystored` *n*
: The delay before responding to a store-double request is n. The default is 2. This overrides the value of `-mmdelay`.

`-mmdelaybcast` *n*
: The delay before responding to a broadcast request is n. The default is 2. This overrides the value of `-mmdelay`.

`-mmbusyfaa0` *n*
: The busy time before another request can be accepted after a Fetch&Add with operand 0 is n cycles. The default is 4. This overrides the value of `-mmbusy`.

`-mmbusystored` *n*
: The busy time before another request can be accepted after a store-double is n cycles. The default is 4. This overrides the value of `-mmbusy`.

`-mmbusybcast` *n*
: The busy time before another request can be accepted after a broadcast is n cycles. The default is 4. This overrides the value of `-mmbusy`.

`-mmbusyreflect` *n*
: The busy time before another request can be accepted after a reflect is n cycles. The default is 4. This overrides the value of `-mmbusy`.

`-mmbankclash` *n*
: Assume two banks per MM if $n > 0$. The default is 0. The penalty is n cycles of busy time between two consecutive requests that hit the same bank. Two 32-bit words with addresses 0 and 4 (mod 8) always fall in different banks within the same MM. The value specified may be any positive integer.

`-stat-mm-usage`
: Compute usage statistics for each MM.

`-stat-flatency-hist` *n*
: Compute forward trip latency histogram, for latencies up to n. The default is 0. See also `-stat-rlatency-hist`, in the PE section.

`-log-mm-dumpcycle`
: Print lots of MM info to the log file each cycle.

`-log-mm-msg`
: Print a running account of MM–network interactions to the log file.

2.1.3. Switch Options. Again, some parameters that you might expect to be switch options are described under general options instead; this is because they actually affect more than just the switches. For example, `-nswports` together with `-numpes` determines the number of stages in the network. Remaining switch options are:

`-fqinputs` *n*
: Configure switches so forward path queues have n inputs. The default is 1. Thus, the number of forward path queues per $N \times N$ switch is $\frac{N^2}{n}$. The value specified must be a power of 2 no greater than N. Figure 2, on the next page, shows how the

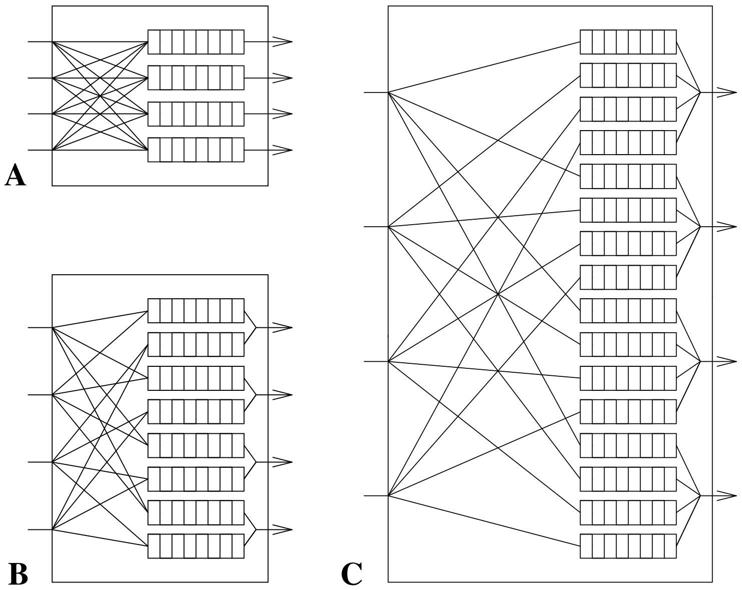

Figure 2: Forward Queue Organization. (Shown for `-nswports 4`)
A: `-fqinputs 4`
B: `-fqinputs 2`
C: `-fqinputs 1`

queues are organized for a 4×4 switch with fqinputs set to 1, 2, and 4.

`-rqinputs` *n*

Configure switches so return path queues have *n* inputs. The default is 1. Note, however, that this parameter controls both ''normal'' and ''decombine'' queues in the return path. Each PE-side port receives messages from each MM-side port and also $maxcomb + 1$ decombined messages from the wait buffer(s), so the total number of return path queues per $N \times N$ switch is $(maxcomb + 1)\dfrac{N^2}{n}$, where *maxcomb* is as described under the `-maxcomb` option. The values of N, *maxcomb*, and *n* must be such that this expression is integral; in addition, *n* must be a power of 2. Figure 3, on the next page, shows some examples of how the queues and wait buffers are organized.

`-wbpeports` *np*
`-wbmmports` *nm*

Configure the switch so that wait buffers are associated with *np* ports on the towards-PE side, and *nm* ports on the towards-MM side. The default is 2 for both options. The actual number of wait buffers in a $N \times N$ switch is $\dfrac{N^2}{np \times nm}$. The values of N, *np*, and *nm* must be such that this expression is integral; in addition, we require *np* and *nm* to be powers of 2 no greater than N, and $np \geq$ fqinputs. Figure 3

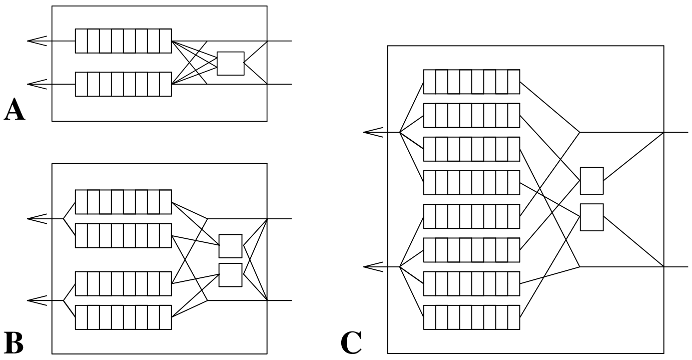

Figure 3: Return Queue and Wait Buffer Organization. Small squares are wait buffers.
A: `-rqinputs 4 -wbpeports 2 -wbmmports 2`
B: `-rqinputs 2 -wbpeports 1 -wbmmports 2`
C: `-rqinputs 1 -wbpeports 2 -wbmmports 1`
(`-nswports 2 -maxcomb 1` for all cases.)

shows some examples of how return queues and wait buffers are organized.

`-maxcomb` *n*
: Allow messages to combine at most *n* times when passing through a switch in the forward direction. The default is 1. Any non-negative integer may be specified; combining can be disabled with `-maxcomb 0`.

`-comb-per-inp`
: Limit combining to 1 from each input port. We require maxcomb to equal nswports when this option is in effect. When maxcomb and nswports are both 2, this is called ‘‘2½-way’’ combining (Dickey and Kenner [2]).

`-suppress-combine` *x*
: Suppress combining at stages indicated by hexadecimal bitmask *x*. The least significant bit corresponds to stage 0, closest to memory. The default is 0, combining allowed at any stage.

`-fqsizp` *n*
: Configure the switches so forward path queues hold *n* packets. The default is 10. The value *n* must be a positive integer, such that $n \geq \text{fqinputs} \times max\_f\_pkts$, where max_f_pkts is the largest number of packets in any forward path message, i.e.,

$$\left\lceil \frac{3 \text{ words} \times 32 \text{ bits}}{\text{pkt\_bit}} \right\rceil$$

(3 words is for the largest forward path message currently supported–store-double).

`-rqsizp` *n*
: Configure the switches so return path queues hold *n* packets. The default is 10. The value *n* must be a positive integer, such that $n \geq \text{rqinputs} \times max\_r\_pkts$, where max_r_pkts is the largest number of packets in any return path message, i.e.,

$$\left\lceil \frac{3 \text{ words} \times 32 \text{ bits}}{\text{pkt\_bit}} \right\rceil$$

(3 words is for the largest return path message currently supported–load-double).

`-wbsizp` *n*
: Configure the switches so wait buffers hold *n* packets. The default is 16. The value n must be a positive integer, such that $n \geq \text{wbpeports} \times max\_f\_pkts$, where max_f_pkts is as described for `-fqsizp`.

`-log-sw-dumpcycle`
: Print lots of switch information every cycle to the log file.

`-log-sw-msg`
: Print a running account of switch message transfers to the log file.

`-stat-fq-len-m`
: Compute a histogram of forward path queue lengths in messages.

`-stat-rq-len-m`
: Compute a histogram of return path queue lengths in messages.

`-stat-wb-len-m`
: Compute a histogram of wait buffer lengths in messages.

`-stat-fq-len-p`
: Compute a histogram of forward path queue lengths in packets.

`-stat-rq-len-p`
: Compute a histogram of return path queue lengths in packets.

`-stat-wb-len-p`
: Compute a histogram of wait buffer lengths in packets.

`-stat-ncomb`
: Compute a histogram of how many times messages combine in each stage, i.e., how many times a message didn't combine at all, how many times a message combined once, how many times a message combined twice, etc.

2.1.4. PE Options. PE options are further divided into general and specific categories. Three forms of PE simulation are available in the current version of Molasses: an interactive mode, an address pattern generator, and a faithful Ultra-3 PE simulator. All of them share the general PE options:

`-pemaxout` *n*
: Limit the total number of memory requests a PE may have outstanding to n. The default is 16. Any positive integer is acceptable.

`-pemaxpend` *n*
: Limit the total number of pending memory requests a PE may have to n. The default is 1. Any positive integer is acceptable. Pending requests are requests that are ready to be sent on the network but haven't gone yet, e.g., due to port blockage, cycle alignment, etc.

`-log-pe-dumpcycle`
: Print lots of PE information every cycle to the log file.

`-log-pe-msg`
: Print a running account of PE–switch message transfers to the log file.

`-stat-latency-hist` *n*
: Compute a histogram of round trip latencies up to n. The default is 0.

`-stat-rlatency-hist` *n*
: Compute a histogram of return trip latencies up to n. The default is 0. See also `-stat-flatency-hist`, in the MM section.

`-stat-platency-hist` *n*
: Compute a histogram of pending latencies up to size n, i.e., how long messages remain pending. The default is 0.

`-stat-noutstand`
: Compute a histogram of outstanding requests per PE.

`-stat-npending`
: Compute a histogram of pending requests per PE.

`-stat-noutpend`
: Compute a histogram of outstanding + pending requests per PE.

`-stat-msg-opcode`
: Compute a histogram of message opcode usage. The message opcodes supported correspond to those of the Ultra-3 prototype. Note that the pseudo-random address pattern generator doesn't generate opcodes other than Fetch&Add.

`-stat-pe-mm-usage`
: Compute a histogram of MM usage from the PE perspective. This will differ from `-stat-mm-usage` if combining takes place, or if the simulation terminates with messages still outstanding.

The following three options select the type of PE simulation. Only one may be specified:

`-interactive-pe`
: Perform interactive address generation. This is the default. There are currently no options associated with this style of PE simulation. A prompt is printed on standard output for each cycle for each PE that may have a chance to issue a request the following cycle. This mode ignores `-pemaxpend`. A blank input line declines the opportunity; lines of the following form make memory requests:

`pws(`a`,`d`)`	Partial Word Store
`reflect(`a`,`d`)`	Reflect
`bcast(`a`,`d`)`	Broadcast
`op3(`a`,`d`)`	Unnamed opcode 3
`op4(`a`,`d`)`	Unnamed opcode 4
`op5(`a`,`d`)`	Unnamed opcode 5
`op6(`a`,`d`)`	Unnamed opcode 6
`op7(`a`,`d`)`	Unnamed opcode 7
`fase0(`a`,`d`)`	Fetch&Store if old value=0
`fasge0(`a`,`d`)`	Fetch&Store if old value≥0
`loadd(`a`)`	Load Double
`faandn(`a`,`d`)`	Fetch&And-Not
`faa(`a`,`d`)`	Fetch&Add
`faor(`a`,`d`)`	Fetch&Or
`fas(`a`,`d`)`	Fetch&Store
`stored(`a`,`$d0$`,`$d1$`)`	Store Double

`-pattern-pe`
Perform address pattern generation. This is described in section 2.1.4.1, unless replaced as described in section 3.2.

`-u3-pe` Simulate the Ultra-3 PE. This option is not yet available.

2.1.4.1. Pseudo-Random Address Generation Mode.

`-pattern-reqprob` p
Set the probability of making a request whenever possible to p. The default is .5. We require $0 < p \leq 1$. The probability applies on any cycle in which the PE is allowed to have additional pending requests (see `-pemaxpend`).

`-pattern-hotprob` p
Set the probability that a request is a ‘‘hot spot request’’ to p. The default is .05. We require $0 \leq p \leq 1$.

`-pattern-nhot` n
Set the total number of hot spots to n. The default is 1; setting it to 0 produces a fully uniform distribution. The value chosen must not be greater than the number of PEs.

`-pattern-hot-focus` n
Set the number of hot spots each PE will consider to n. The default is 1; setting it to 0 produces a fully uniform distribution. The probabilities for the n hot spots are all the same ($\frac{p}{n}$ for `-pattern-hotprob` p).

`-no-pattern-hot-unique-mm`
Choose hot spot addresses randomly over all simulated memory. The default is to first choose hot MMs randomly, and then choose a word randomly within each hot MM; thus no MM will have more than one hot spot.

`-pattern-hot-focus-indep`
When pattern-nhot > 1 and pattern-nhot > pattern-hot-focus, cause each PE to

choose its hot spots independently. The default is to choose so that each hot spot gets the same number of PEs, to the greatest extent possible.

`-pattern-cycles` *n*
: Simulate *n* cycles. The default is 1000.

`-pattern-quiesce`
: Wait for all requests to be satisfied before termination. This is used in conjunction with `-pattern-cycles`, and usually causes additional cycles to be simulated, during which the probability of making a new request is 0.

`-stat-hotspot-addr`
: Print hot spot addresses.

`-stat-hot-latency`
: Compute hot and cool message latency statistics.

`-pattern-debug` *n*
: Set address pattern generation debug bits to *n*. The default is 0.

2.2. Deviations from Reality

While Molasses is closely based on the Ultra-3 prototype, there are ways in which the simulation is not completely faithful. There are several general reasons for such deviations:

- Molasses is a functional rather than a device-level simulator. It is not derived from CAD files for Ultra-3.
- Performance considerations make a slightly unfaithful simulation attractive.
- The goal of exploring scalability and hardware/software tradeoffs in future large machines leads us to consider machines that *could be* built, rather than limiting ourselves to machines that *have been* built.

Being able to correlate Molasses results with the observable performance of Ultra-3 is important, however, so actual differences should be small. The following are some of the specific ways in which Molasses deviates from the real Ultra-3:

- *Queue capacities.* The queues and wait buffers in the switches are not modeled after the systolic structures used in the actual chips. As a result, the concepts of queue capacity don't precisely match. This can alter the behavior of port blocking.
- *Counting queue contents.* The count of how many packets a queue (or wait buffer) currently holds is only approximate. When the first packet of a message is inserted, the count is incremented by the full number of packets of the message. Likewise, when the first packet of a message leaves, the count is decremented by the full number of packets of the message.
- *Combining in first queue slice.* We don't account for the real switch's inability to combine with a message in the first ''slice'' of the queue, i.e., about ready to be transmitted.
- *Memory refresh.* We don't account for MM busy cycles due to memory refresh.

3. Modifying Molasses

Molasses is amenable to modification in several ways. Certainly, it is merely a C program (although a fairly large one), and the source code is available, but we can say somewhat more than that. Table 1, on the next page, lists the contents of the Molasses source directory.

File	*Purpose*
`README`	Hopefully up-to-date information
`doc`	Unformatted source for this document
`doc.ps`	Printable (PostScript) version of `doc`
`16pe.pic`	Figure 1 in this document
`fq.fig`	Figure 2 in this document
`rqwb.fig`	Figure 3 in this document
`doc.refs`	References for this document
`makeit`	Shell script to run make
`Makefile`	Knows how to compile Molasses
`RCS`	Directory of rcs version files
`config.h`	Machine dependent definitions
`molasses.h`	Common macros and declarations
`cmdarg.h`	Command line argument handling declarations
`mm.h`	MM-related declarations
`pe.h`	General PE-related declarations
`realpe.h`	PE declarations sensitive to PE type
`u3pe.h`	PE declarations for Ultra-3 (incomplete)
`molasses.c`	Main loop, common support functions
`pe.c`	General PE simulator, also interactive
`switch.c`	Switch simulator
`mm.c`	MM simulator
`patrand.c`	Pseudo-random address pattern generator
`u3pe.c`	Ultra-3 faithful PE simulator (incomplete)
`u3*`	Other Ultra-3 files (incomplete)

Table 1: Contents of Molasses Source Directory.

3.1. Porting Molasses to Another Host

Molasses is written in C, staying fairly close to the ANSI standard. It also stays fairly close to the POSIX operating system standard. It should be possible to port to most modern environments by modifying only the files `makeit` and `config.h`.

3.2. Address Pattern Generation

The entire address pattern generator can be readily replaced with any other load generator desired. This could be used, for example, to provide a trace-driven capability to Molasses, or even to add an alternative microprocessor simulator. This is the easiest replacement option for the PE.

The address pattern generator is contained in a single file, `patrand.c`. A reasonable way to begin is to copy this file and make modifications to it. It defines a function for the following prototype:

```
extern int
pepattern (unsigned penum, unsigned long cycle, struct message *inm,
           int inwords, struct message *outm);
```

The generic portion of the PE simulator calls `pepattern` for every cycle for every PE. The PE number is given in `penum`. The current cycle number is given in `cycle`. The parameter `inm` is non-`NULL` whenever we receive a packet; `inwords` is a bitmask with bits set to indicate which portions of the message are newly available this cycle:

`01`	Opcode and address
`02`	First data word
`04`	Second data word

The parameter `outm` is non-`NULL` if it appears that a message can be sent the next cycle; `pepattern` can fill in the structure and return 1 to send it. To cause Molasses to terminate at the end of the current machine cycle, the global integer `pepattern` can be set to a non-zero value.

The message structure contains the following fields:

`opcode`	Type of message
`penum`	Requesting PE number
`combhist`	Combining history
`addr`	Memory address
`fstartcycle`	Cycle message started forward
`rstartcycle`	Cycle message started back
`u`	Union for data and avail list
`u.availnext`	Avail list pointer
`u.data[2]`	Signed data words
`u.udata[2]`	Unsigned data words

The message structure is defined in `molasses.h`, along with lots of other useful and important things; it should be read before starting serious Molasses work. The data words are of type `word` (or `uword`, for the unsigned view), a 32 bit integer defined in `config.h`. When sending a message, the only fields `pepattern` must set are `opcode` (typically set to `MSG_OP_FAA`, for Fetch&Add, or `MSG_OP_FAS`, for Fetch&Store), `addr`, and `u.data`. Management of the other fields, including initialization and statistics gathering (`fstartcycle` and `rstartcycle`) is performed by the generic PE simulator.

In addition to `pepattern`, the `patrand.c` file must define an array of `struct arglist` structures defining command line argument behavior, and several additional functions:

`void validate_pattern_pe_params(void)`
: Called after processing command line arguments to perform any necessary parameter constraint checking.

`void pattern_init()`
: Called after validation to perform arbitrary initialization.

`void startpepatternstats()`
: Called to (re)initialize any statistics-gathering structures at the beginning of every statistics interval (see `-stat-cycles`).

`void writepepatternstats (FILE *f)`
: Called at the end of every statistics interval to write data on the indicated file.

3.3. Unit Replacement

Generally speaking, Molasses is structured in four major parts:

(1) The PE.

(2) The switch.

(3) The MM.

(4) Common data structures and functions.

Any of the first three can be replaced by observing a small number of interfaces. In addition, the PE has at least two sub-levels of unit replacement:

(1a) The microprocessor, of which the interactive, address pattern generator, and faithful PE simulator are instances. This excludes some generic PE functionality, such as interfacing with the network, keeping track of pending and outstanding requests, and performing instrumentation, which are all useful for just about any PE organization.

(1b) The address pattern generator has an even lower level for unit replacement, as described in section 3.2.

The central data structures tieing Molasses together are the message and the bidirectional port, defined in `molasses.h`. Each PE has one port, each $N \times N$ switch has $2N$ ports, and each MM has one port. If the PE, MM, or switch portion of Molasses is replaced with something else that operates on ports and messages in a compatible way, it will work.

The port data structure contains the following fields:

- A pointer to the connected peer.
- An input structure, containing clear-to-send signal(s) sent to the peer.
- An output structure, containing a message pointer and the number of packets remaining to send.

The input and output structures each appear twice, with one representing the even cycles and the other representing odd cycles. When simulating a component (PE, switch, or MM), the output port structure is set up one cycle ahead of time, so input ports can be read without races.

3.4. Major Surgery

Some reasonable changes cannot be made by unit replacement, although they might not necessarily be difficult. This includes such things as network topology changes, adding ports to components PE, switch, or MM, certain message enhancements, changes to the strategy for logging or statistics, etc.

4. Results and Performance

Statistical results are usually extracted by processing the statistics file with awk to extract desired data.

5. Current Status and Future Work

Molasses currently consists of 14582 lines of code, broken down in the following way:

all `.h` files	2872
all `.c` files	11710
generic `.h` files	1299
generic `.c` files	1757
switch	1750
MM	1100
generic PE	1611
Ultra-3 PE	6439
patrand.c	626

The faithful Ultra-3 PE simulator is not yet complete. Related features, such as checkpoint/restart and the simulation modes described in section 1 on page 2, are not yet complete either, but aren't needed until the faithful PE simulator is available.

The most important future work items are to complete the Ultra-3 PE portion of Molasses and the checkpoint/restart feature. Here are some additional non-trivial items for future investigation:

- The switch queue and wait buffer packet counts should be maintained more accurately (see section 2.2).
- Can we support direct execution as an unfaithful but fast way of performing the initialization portion of an Ultra-3 program run under simulation? Of course, this requires the host computer to execute the same instruction set as the target (e.g., Am29k for Ultra-3). What are the minimum host OS changes required to do this efficiently?
- Some work on visualizing the simulation has already been performed, in conjunction with David Wood at IBM, using the IBM DataExplorer. This should be further investigated. Performance is a major issue. Visualization should probably be more tightly integrated with Molasses; this could also allow some interactive graphical control over the simulation.
- Support for alternate network topologies should be added.
- An option for local access to a PE's corresponding MM should be added. This probably requires adding an additional port to the PE and MM structures, along with appropriate arbitration. Options for more memory interleaving options would be desirable as well.
- Options for a second level cache should be added.
- Options for cache coherence mechanisms should be added. The real challenge here is not only to support good coherence mechanisms, but to provide a relatively easy way to alter the coherence strategy, such as was provided for address pattern generation in section 3.2.
- Some way of exploring hardware multithreading should be provided. An alternative to this would be to allow more than one microprocessor for a single PE port. A reasonable set of options should be devised for this. Doing this without sacrificing simulation performance may be a challenge.

6. Conclusions

Careful design and implementation allows us to take advantage of recent computer hardware performance advances and conduct whole-machine simulations of non-trivial workloads.

7. Acknowledgements

Susan Dickey was the first brave user of Molasses, and her feedback was invaluable.

References

[1] Ronald Bianchini, Susan Dickey, Jan Edler, Gabriel Goodman, Allan Gottlieb, Richard Kenner, and Jiarui Wang, ''The Ultra III Prototype'', *Proceedings of the Parallel Systems Fair, 7th International Parallel Processing Symposium*, pp. 2–9 (April 14, 1993). Cited on page 1.

[2] Susan R. Dickey and Richard Kenner, ''Combining Switches for the NYU Ultracomputer'', *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computiation* , pp. 521–523 (October 1992). Cited on page 9.

[3] Allan Gottlieb, ''An Overview of the NYU Ultracomputer Project'', pp. 25–95 in *Experimental Parallel Computing Architectures*, ed. Jack J. Dongarra, North Holland (1987). Cited on page 1.

[4] Yue-Sheng Liu, ''Architecture and Performance of Processor-Memory Interconnection Networks for MIMD Shared Memory Parallel Processing Systems'', Ph.D. Thesis, Courant Institute, NYU (September 1990). Cited on page 5.