MacSim: A CPU-GPU Heterogeneous Simulation Framework

User Guide

Hyesoon Kim
Jaekyu Lee
Nagesh B. Lakshminarayana
Jaewoong Sim
Jieun Lim
Tri Pho

HPArch research group
(http://comparch.gatech.edu/hparch/index.html)
Georgia Institute of Technology

Contents

1

Introduction

MacSim is a heterogeneous architecture simulator, which is trace-driven and cycle-level. It thoroughly models architectural behaviors, including detailed pipeline stages, multi-threading, and memory systems. Currently, MacSim support x86 and NVIDIA PTX instruction set architectures (ISA). MacSim is capable of simulating a variety of architecreus, such as Intel's Sandy Bridge [**?**] and NVIDIA's Fermi [**?**]. It can simulate homogeneous ISA multicore simulations as well as heterogeneous ISA multicore simulations.

MacSim is a microarchitecture simulator that simulates detailed pipeline stages (in-order and out-of-order) and the memory system components including caches, NoC, and memory controllers. It supports asymmetric multicore configurations as well as SMT or MT architectures.

Currently interconnection network model (based on IRIS) and power model (based on McPat [**?**]) are implemented. ARM ISA support is on-progress. MacSim is also one of the components of SST [**?**] so multiple MacSim simulators can run concurrently.

Macsim version information

2.0.4 - October, 2014 macsim/tags/macsim-2.0.4/

2.0.3 - August, 2014 macsim/tags/macsim-2.0.3/

2.0.2 - April, 2014 macsim/tags/macsim-2.0.2/

2.0 - September, 2013 macsim/tags/macsim-2.0/

1.2.1 - April, 2013 macsim/tags/macsim-1.2.1/

1.2 - October, 2012 macsim/tags/macsim_1.2

1.1 - October, 2012 macsim/tags/macsim_1.1

1.0 - February, 2012 Initial release macsim/tags/macsim_1.0

2

Getting Started

This chapter provides instructions for building, installing, and running MacSim.

2.1 Documentation and Other Support

MacSim is hosted on github at the following URL: https://github.com/macsimgt/macsim-public. The project page for MacSim on Google Code provides stable version of MacSim, detailed documentation, issue/bug tracking, sample traces and so on. Users can use the project page for filing issues and for contacting the maintainers of MacSim.

2.2 Building MacSim

2.2.1 Obtaining Source

MacSim source code is maintained using subversion. Users can obtain a copy of the source code using this command:

```
svn co https://svn.research.cc.gatech.edu/macsim/trunk macsim-readonly --username readonly
```

Note that currently password is not set for readonly account, so 'enter' when you prompt password. Due to the technical issue, we do not allow anonymous checkout now. As soon as the issue has been resolved, we will update this documentation.

Users will have read-only permission by default and users interested in contributing to MacSim must contact macsim-dev@googlegroups.com.

2.2.2 Requirements

To build MacSim the following are required:

Operating System At present only Linux distributions are supported. MacSim has been tested on: Ubuntu, Redhat (TODO)

Compiler Any compiler that supports the C++0x (or C++11) standard. MacSim has been verified to work with: gcc 4.4 or higher, icc (will work on)

SConstruct apt-get install scons

Libraries - The zlib library is required and it can be installed using the command:

```
Ubuntu: apt-get install zlib1g-dev
Redhat: TODO
```

2.2.3 Build

The SConstruct is used to build MacSim. After checking out a copy of the MacSim source code, the following commands have to be executed (These instructions are also available in INSTALL file included in the MacSim source). Section 2.2.5 details all available build options.

```
./build.py
```

On a successful build, the binary macsim will be generated in the bin/ directory. Please note that SST-Macsim still uses GNU Autotools, so we maintain some related files (Makefile.am).

2.2.4 Build Types

Three types of build, each of which uses different compiler flags are supported. The build types are:

- Optimized version (-O3 flag) - default
- Debug version (-g3 flag)
- Gprof version (-pg flag)

2.2.5 Build Options

There are several options in build.py.

- -j : specify the number of threads to be used for building MacSim.
- -d : debug version
- -p : gprof version
- -c : cleanup
- -t : build test
- - -dramsim : macsim with DRAMSim2 simulator (DRAM memory)
- - -iris : macsim with Iris simulator (interconnection)
- - -power : macsim with power module (currently, not supported)

We also provide macsim.config file to set the configuration instead of specifying options with the commandline. The following is the content of the macsim.config file.

```
[Build]
debug: 0
gprof: 0

[Library]
dram: 0
power: 0
iris: 0
```

Users can change the value to 1 for the corresponding option.

2.3 Running MacSim

To run macsim binary, two additional files are required to be present the same directory as the binary.

1 **params.in** - defines architectural parameter values that will overwrite the default parameter values.

2 **trace_file_list** - specifies the number of traces to run and the location of each trace.

Several pre-defined architectural parameter configuration files are provided in params/ directory. Table 1 lists the included pre-defined parameter files. To run MacSim with a particular architectural configuration, users should copy the corresponding parameter file to bin/ directory and rename it as params.in. For example, to use NVIDIA's Fermi [**?**] GPU architecture, params/params_gtx465 should be copied to bin and renamed as params.in.

Table 1. Parameter Templates.

File Name	Description	Architecture
params_8800gt		NVIDIA GeForce 8800 GT (G80)
params_gtx280		NVIDIA GeForce GTX 280 (GT200)
params_gtx465		NVIDIA GeForce GTX 465 (Fermi)
params_x86		Intel's Sandy Bridge (CPU part only)
params_hetero_4c_4g		Intel's Sandy Bridge (CPU + GPU)

trace_file_list specifies the list of traces to be simulated. Following is the content of a sample trace_file_list:

```
2                                                  <-- number of traces
/trace/ptx/cuda2.2/FastWalshTransform/kernel_config.txt   <-- trace #1 path
/trace/ptx/cuda2.2/BlackScholes/kernel_config.txt         <-- trace #2 path
```

Above, the first line specifies the number of traces (applications) that will be simulated and each line thereafter specifies the path to the trace configuration file in the trace directory of an application. The contents of a sample trace configuration file are shown below.

```
1 x86
0 0
```

In the first line, the first field indicates the number of threads in the application (1 in this case) and the second field specifies the type of the application (x86 in this case). Other lines contain a thread id and the starting point of the thread in terms of the instruction count of the main thread (thread 0).

Several sample trace files are available at http://code.google.com/p/macsim. Users can generate their own traces by following instructions in Chapter 3.

Executing

The MacSim simulator can be run by executing the macsim binary located in the bin/ directory.

```
./macsim
```

2.4 Output Files

MacSim generates a params.out file and several files *.stat.out containing statistics at the end of a successful simulation. Users can specify the output directory for these files by setting the STATISTICS_OUT_DIRECTORY parameter whose default value is the directory containing the macsim binary.
params.out enumerates all parameters with their values used for the simulation. Simulation parameters are discussed in detail in section 4. The *.stat.out files contain statistics such as the number of simulation cycles and so on. Chapter 5 details the kinds of statistics supported, how to add new statistics and how to examine and interpret the contents of .stat.out files.

2.5 SST-Macsim

Macsim is a part of the SST simulation framework [**?**] and this section is intended for SST users who want to use MacSim for GPU simulations.

2.5.1 Installing SST

Instructions for installing SST are provided in the Wiki page at the SST Google Code repository [**?**] at http://code.google.com/p/sst-simulator.

2.5.2 Building MacSim as a SST component

Check out a copy of MacSim from SVN in *sst-top/sst/elements* and apply the macsim-sst.patch.

```
cd sst-top/sst/elements
svn co https://svn.research.cc.gatech.edu/macsim/trunk macsim
cd macsim
patch -p0 -i macsim-sst.patch
```

Then, re-run the SST build procedure.

```
cd sst-top
./autogen.sh
./configure --prefix=/usr/local --with-boost=/usr/local --with-zoltan=/usr/local --with-parmetis=/usr/local
```

2.5.3 Configuring the SST-MacSim component

An example SST sdl configuration file setup for simulating MacSim is shown below:

```
<?xml version="1.0"?>
<sdl version="2.0"/>

<config>
  stopAtCycle=1000s
  partitioner=self
</config>

<sst>
  <component name=gpu0 type=macsimComponent.macsimComponent rank=0 >
    <params>
      <paramPath>./params.in</paramPath>
      <tracePath>./trace_file_list</tracePath>
```

```
    <outputPath>./results/</outputPath>
    <clock>1.4Ghz</clock>
   </params>
  </component>
</sst>
```

In this manner, an SST simulation configuration file can declare multiple instances of MacSim as well as define the traces are run on each MacSim instance.

2.5.4 Running a MacSim simulation in SST

Ensure that SST and MacSim components are compiled and/or installed. Ensure that the paths and contents of both SST configuration sdl file and MacSim *params.in* configuration file are correct. Start the SST simulation either standalone or through MPI.

```
sst.x [sdl-file]
```

or

```
mpirun -np [x] sst.x [sdl-file]
```

3

Traces

For simulations using MacSim, x86 traces are generated using Pin [**?**] and PTX traces are generated using GPUOcelot [**?**]. Internally, MacSim converts both x86 and PTX trace instructions into RISC style micro-ops (uop) for simulation. Figure 1 shows a high-level overview of this procedure.

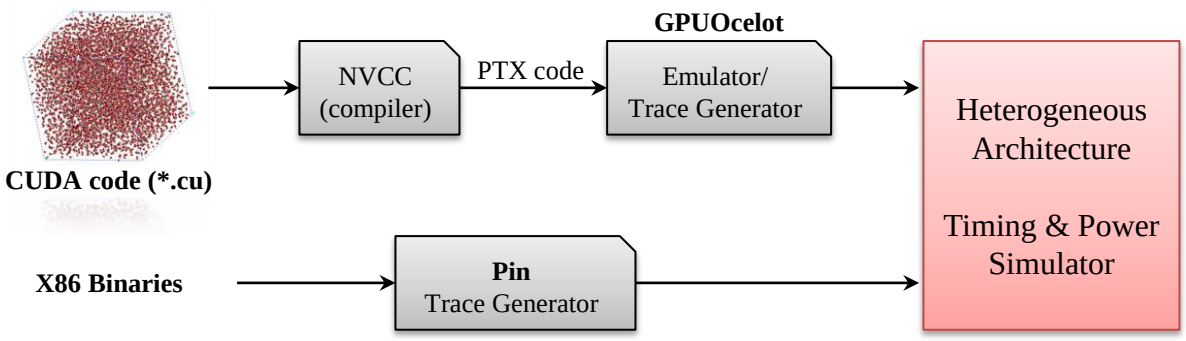

Figure 1. The overview of MacSim Traces Generation

3.1 CPU (x86) Traces

MacSim includes a CPU (x86) trace generator which is based on Pin [**?**], a binary instrumentation tool. Documentation regarding Pin can be found at http://www.pintool.org. After installing Pin[1], the x86 trace generator module has to be built. The command for doing so is:

```
cd toos/x86_trace_generator
make
```

This will generate trace_generator.so in the tools/x86_trace_generator/obj-intel64 directory. x86 traces for MacSim can then be generated by running Pin with the generated module.

```
pin -t trace_generator.so -- $BIN $ARGS (for single-threaded applications)
pin -t trace_generator.so -thread N -- $BIN $ARGS (for multi-threaded applications with N threads)
```

The following example shows the generation of traces for an execution of /bin/ls.

```
pin -t trace\_generator.so -- /bin/ls
```

[1] Note that our trace generator may not be backward/forward compatible with different Pin versions. Currently, Pin 41150 revision (Jun 07, 2011) must be used.

The binary (ls) is actually executed on top of Pin and the instructions executed by the binary are written to the trace file. The output on the screen when generating traces for *ls* is shown.

```
pin -t trace_generator.so -- /bin/ls
-> Thread[0->0] begins.
-> Trace Generation Starts at icount 0
dump.txt_0.dump  pin.log  trace_0.raw  trace_generator.o  trace_generator.so
xed_extractor.o  xed_extractor.so
-> Trace Generation Done at icount 475195
```

The trace generator generates two files (in case of a single threaded application) - Trace.txt and trace_0.raw, in the current directory. Section 3.4 provides details of the generated files.

3.2 GPU (PTX) Traces

GPU (PTX) traces are generated using GPUOcelot [**?**], a dynamic compilation framework for heterogeneous systems.

3.2.1 Installing Ocelot

Ocelot can be installed using the Ubuntu package provided at the Google Code page for Ocelot, or the source can be downloaded, built and installed. Below is the sequence of commands to executed to build and install from the source. First, checkout a copy of Ocelot from its SVN repository.

```
svn checkout http://gpuocelot.googlecode.com/svn/trunk/gpuocelot
```

Then build Ocelot and the trace generator libraries.

```
cd gpuocelot/ocelot; sudo ./build.py --install
cd gpuocelot/trace-generators; libtoolize; aclocal; autoconf; automake; ./configure;
make; sudo make install
```

All libraries will be installed in the system library path directory (`/usr/local/lib`). More detailed instructions for installing Ocelot are available at Ocelot's Google Code project page.

3.2.2 Generating Traces

CUDA executables targeted for trace generation must be linked against libocelot.so and libocelotTrace.so. libocelot.so provides the functionality of the CUDA runtime library (libcudart.so) and libocelotTrace.so contains the trace generator for MacSim and tools provided by Ocelot.

To execute a binary linked against libocelot.so a configuration file, `configure.ocelot` (which is in JSON format), is required in the same directory as the binary. A copy of this file with some default settings can be obtained from the Ocelot source. To enable trace generation, open your copy of the configure.ocelot and add the pair `x86Trace: true` on a new line under the trace member as shown below. Make sure that there is a comma at the end of each pair that is not the last pair of a member.

```
trace: {
  database: "traces-ipdom/database.trace",
  memoryChecker: {
    enabled: true,
    checkInitialization: false
  },
```

```
  raceDetector: {
    enabled: true,
    ignoreIrrelevantWrites: true
  },
  cacheSimulator: {
    enabled: false,
  },
  branch: false,
  memory: false,
  x86Trace: true
},
```

In addition to editing configure.ocelot, the following four environment variables have to be set:

- TRACE_PATH : directory to store generated traces. If not specified, current directory is used by default.
- TRACE_NAME : prefix used in the file name for generated traces. If not specified, "Trace" is used by default.
- KERNEL_INFO_PATH : name of file that should contain kernel information (must be specified)
- COMPUTE_VERSION : compute capability (default 2.0)

Examples for setting up these environment variables are shown below.

```
export TRACE_PATH="/storage/traces/"  # Create a trace directory in the /storage/traces
export KERNEL_INFO_PATH="kernel_info" # kernel_info has the kernel information
export COMPUTE_VERSION="2.0"          # Calculate occupancy based on compute capability 2.0
```

On executing the binary of a kernel, an information file and a directory for each kernel invocation by the binary are generated. These directories contain traces from different invocations of the kernel. The directories path are addressed by `TRACE\_PATH`. The kernel information file contains the version of generated traces and path of each trace configuration file. The trace configuration file contains the number of warps, the trace version, maximum number of thread blocks that can be assigned to each SM and the ID, and starting information of each warp. More information regarding the generated files can be found in Sections 3.4 and 3.4.3.

3.3 Supporting Other Architectures

To support other ISAs such as ARM, a frontend simulator or a functional emulator is necessary to provide the executed instruction stream. Instructions in this stream can be translated by MacSim into its internal RISC style micro-ops and simulated. Currently, plans for supporting ARM ISA and OpenGL programs are in progress.

3.4 Trace Formats

Note: *Although different trace generators use the same data structure for storing instruction traces, the meaning of some of the members of this common data structure is different in PTX traces. Note that efforts to have different data structures for x86 and PTX traces for sake of extensibility and clarity are underway.*

On a successful trace generation, the x86 trace generator generates a configuration file called Trace.txt and one Trace_xx.raw file (assuming that TRACE_PATH was set to "Trace") for each thread of the application in the output directory. On the other hand, the PTX trace generator generates a kernel information file and several directories containing traces of individual kernel invocations. The directory for each kernel invocation contains a Trace.txt file and one Trace_xx.raw file for each warp in the kernel. Note that in case of PTX kernels, one trace file is generated for each warp, there are no per thread trace files. Further details regarding trace generation for PTX kernels can be found in Section 9.1. The purposes of Trace.txt and Trace_xx.raw are shown below and their formats are explained in detail in subsequent sections.

- Trace.txt (info trace): contains information about the generated trace files (#threads, trace type, ...).
- Trace_xx.raw (raw trace): contains instruction trace for a thread and is generated for each thread.[2]

3.4.1 Trace.txt

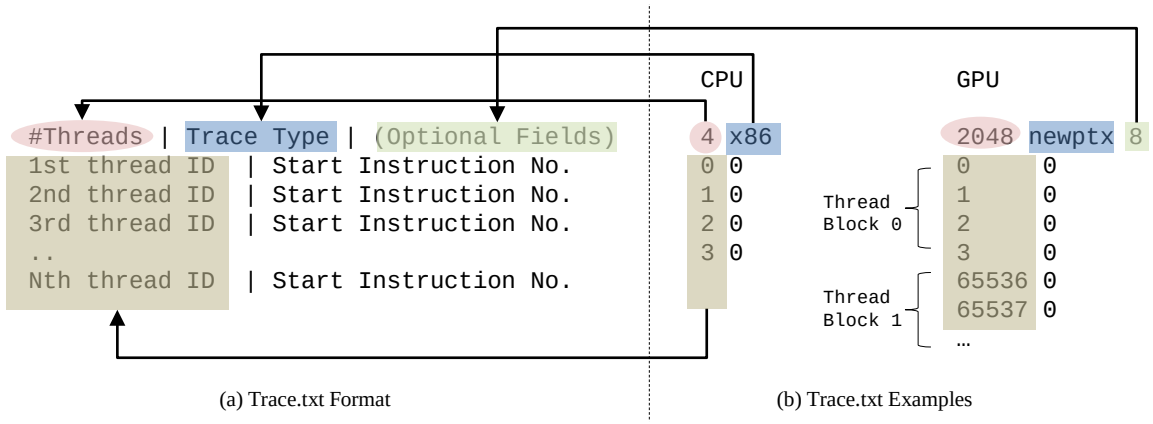

Figure 2. Trace.txt format

Figure 2-(a) shows the format of Trace.txt and its CPU and GPU examples. As shown in Figure 2-(a), the first line in Trace.txt has different fields from the rest of the lines.

- #Threads: indicates the number of threads for which traces have been generated, and this value is equal to the number of lines in the file excluding the first line.
- Trace Type: indicates whether the generated traces are for an x86 application or a PTX kernel.
- Optional Field(s): currently used for PTX traces only and indicates the number of thread blocks that can be assigned to a streaming multiprocessor(SM) core (occupancy).

From the second line onwards, there are two fields in each line: thread id and start instruction number. For each thread, there is a Trace_<thread_id>.raw file which contains the dynamic instruction trace for the thread. Finally, start instruction number indicates when each thread should be started in terms of the number of instructions simulated for the main thread of the application. In a PTX kernel since all warps are ready for execution at the launch of the kernel, the start instruction number for all threads is zero. On the other hand, for a x86 application, the start instruction is non-zero for all threads except thread 0, which is the main (or parent) thread in the application. This is because in most multi-threaded CPU applications, main thread (thread id 0) spawns children threads.

[2] In case of GPU, a warp is mapped to a MacSim thread, where a warp consists of 32 threads in CUDA.

In Figure 2-(b), the CPU trace has four threads and its type is set to x86. The ids of the threads are 0-3 with the corresponding trace files being Trace_0.raw−Trace_3.raw . Thread 0 is ready at the start of simulation, while Threads 1, 2 and 3 become ready when Thread 0 has fetched x, y and z instructions respectively.

In the GPU example, the number of traces files is 2048 since #Threads (representing #Warps in case of GPUs) is 2048. The optional field indicates that eight thread blocks can be assigned to a SM core.

For GPU traces, the id in the file encodes thread block information as well. The warp id and thread block id can be decoded from this id as follows:

```
warp_id  = id % (1 << 16)
block_id = id / (1 << 16)
```

3.4.2 Trace_xx.raw

Trace_xx.raw is generated for each thread/warp and contains the dynamic instruction trace for the thread-/warp in binary format. The structure/format for encoding instructions is the same in both x86 and PTX traces and looks like as follows (in order):

Type	Size (Bytes)	Field	Description
uint8_t	1	m_num_read_regs	number of source registers
uint8_t	1	m_num_dest_regs	number of destination registers
uint8_t	9	m_src[MAX_SRC_NUM]	source register IDs
uint8_t	6	m_dst[MAX_DST_NUM]	destination register IDs
uint8_t	1	m_cf_type	branch type
bool	1	m_has_immediate	indicates whether this instruction has immediate field
uint8_t	1	m_opcode	opcode
bool	1	m_has_st	indicates whether this instruction has store operation
bool	1	m_is_fp	indicates whether this instruction is a FP operation
bool	1	m_write_flg	write flag
uint8_t	1	m_num_ld	number of load operations
uint8_t	1	m_size	instruction size
uint32_t	4	m_ld_vaddr1	load address 1
uint32_t	4	m_ld_vaddr2	load address 2
uint32_t	4	m_st_vaddr	store address
uint32_t	4	m_instruction_addr	PC address
uint32_t	4	m_branch_target	branch target address
uint8_t	1	m_mem_read_size	memory read size
uint8_t	1	m_mem_write_size	memory write size
bool	1	m_rep_dir	repetition direction
bool	1	m_actually_taken	indicates whether branch is actually taken

Note that the raw trace is compressed with zlib to reduce the sizes of the generated trace files, and the size of each field is the size before the compression.

3.4.3 kernel_config.txt (only for PTX)

For PTX traces, as described in Section 3.2, a directory is created for each kernel invocation, where Trace.txt and Trace_xx.raw are generated. Since typical GPU applications usually invoke several kernels (or execute the same kernel repeatedly), PTX traces can have multiple kernel directories. Thus, in order to simulate all invoked kernels for a GPU application, the PTX trace generator creates kernel_config.txt which contains information of the invoked kernels.

```
Contents of output directory after trace generation

ll /trace/ptx/parboil/bfs

drwxr-xr-x  4  4096 Sep 21 13:21 .
drwxr-xr-x 11  4096 Sep 13 18:02 ..
drwxr-xr-x  2  4096 Apr  7  2011 _Z17BFS_in_GPU_kernelPiS_P4int2S1_S_S_iS_ii_0
drwxr-xr-x  2  4096 Apr  7  2011 _Z26BFS_kernel_multi_blk_inGPUPiS_P4int2S1_S_S_S_S_iiS_S_S__0
-rw-r--r--  1   184 Apr  7  2011 kernel_config.txt

In kernel_config.txt

-1 newptx
/trace/ptx/parboil/bfs/_Z17BFS_in_GPU_kernelPiS_P4int2S1_S_S_iS_ii_0/Trace.txt
/trace/ptx/parboil/bfs/_Z26BFS_kernel_multi_blk_inGPUPiS_P4int2S1_S_S_S_S_iiS_S_S__0/Trace.txt
```

As shown above, all the invoked kernels are enumerated in kernel_config.txt. The first line indicates that this is a wrapper file which points to (multiple) trace.txt files, one for each kernel invocation. MacSim reads and simulates the traces sequentially, one kernel at a time. In the above example (bfs), kernel_config.txt indicates that there are two different kernels in bfs, each of which was invoked once. When running a GPU simulation, the path to the kernel_config.txt file is specified trace_file_list (Section 2.3). Also, we can simulate specific kernels by modifying the kernel_config.txt file.

3.4.4 Translation into micro-ops

During simulation, each instruction in a *raw trace* file is converted into one or more micro-ops internally. MacSim stores such decoded micro-uops in the MacSim-specific structure shown in Table 2.

Table 2. MacSim-specific data structure for micro-ops.

Type	Variable	Description
uint8_t	m_opcode	opcode
Uop_Type	m_op_type	type of operation
Mem_Type	m_mem_type	type of memory instruction
Cf_Type	m_cf_type	type of control flow instruction
Bar_Type	m_bar_type	type of barrier caused by instruction
uns	m_num_dest_regs	number of destination registers written
uns	m_num_src_regs	number of source registers read
uns	m_mem_size	number of bytes read/written by a memory instruction
uns	m_inst_size	instruction size
Addr	m_addr	PC address
reg_info_s	m_srcs[MAX_SRCS]	source register information
reg_info_s	m_dests[MAX_DESTS]	destination register information
Addr	m_va;	virtual address
bool	m_actual_taken	branch actually taken
Addr	m_target	branch target address
Addr	m_npc	next PC address
bool	m_pin_2nd_mem	has second memory operation
inst_info_s	*m_info	pointer to the instruction hash table
int	m_rep_uop_num	repeated uop number
bool	m_eom	end of macro
bool	m_alu_uop	ALU uop
uint32_t	m_active_mask	active mask
uint32_t	m_taken_mask	branch taken mask
Addr	m_reconverge_addr	address of reconvergence
bool	m_mul_mem_uops	multiple memory transactions

4

Configuring MacSim

To control simulation and architectural parameters, knob variables defined in *def/*.param.def* files are used. The build process automatically converts the knob definitions in these files into c++ source that gets included in the compilation of the MacSim binary. Using different parameter values for the knob variables MacSim can be configured to simulate different CPU, GPU and even heterogeneous architectures.

4.1 Adding a new knob

A new knob variable can be defined by adding a line in the following format in one of the *param.def* files:

```
param<{name used in MacSim}, {name used in the command line or params file}, {knob type}, {default value}>
```

It is recommended that the first two arguments be the same, except that the former be in upper case and the latter in lower case. For example,

```
param<L2_ASSOC, l2_assoc, int, 8>
```

Knobs can be pretty much of any type, to support knobs of type other than the basic data types users may have to add code to *src/knob.h*. Knobs of type string are already supported, this is helpful for specifying policies and configurations such as branch predictor type, instruction scheduler type, dram scheduling policy and so on as strings instead of integers.

4.2 Accessing the value of a knob in MacSim

In the MacSim code, a knob variable is accessed by prefixing its name (in uppercase) with *KNOB_*. For example, L2_ASSOC defined in the example in Section 4.1 can be accessed in MacSim using the name KNOB_L2_ASSOC. To access the value of a knob either the *getValue()* function or the name of the knob (works because of operator overloading) can be used. (e.g. **KNOB(KNOB_L2_ASSOC)*)

4.3 Assigning values to knobs

When defining a knob in one of the **.param.def* files, a default value for the knob has to be specified. If a user does not set the value of a knob for a simulation, then the knob assumes its default value. If a user wishes to change the value of a knob for a simulation, there are two ways in which the user can accomplish this:

1. edit params.in - this file is read by the MacSim binary on startup for parameter values. Sample parameter files with parameter values for different configurations can be found in *params/* directory. Each line in a parameter file consists of a knob name (in lowercase) followed by the parameter value. For example:

   ```
   l1_assoc 8
   l2_assoc 16
   ```

2. specify parameter values from the command line - an user can specify knobs and their values from the command line as shown below.

   ```
   ./macsim --l1_assoc=8 --l2_assoc=16
   ```

For a knob with values specified in the params.in file as well as the command line, the value specified in the command line takes priority over the value specified in the params.in file.

4.4 Setting up Parameters

This section shows how different kinds of simulations and simulation configurations can be achieved using knobs and parameter values. MacSim can model up to three types of cores, SMALL, MEDIUM and LARGE, in a simulation. Equivalent knobs are provided for each core type for configuration purposes. Knobs for medium and large cores use `medium` and `large` in their names, while knobs for small cores do not use any such identifiers in their names. For example, the knob rob_size sets the length of the ROB for a small core. The equivalent knobs for medium and large cores are rob_medium_size and rob_large_size.

4.4.1 Repeating Traces

For multiple-application simulations, sometimes, early-terminating applications have to be re-executed to model resource contention (cache, on-chip interconnection, and memory controller) until all applications finish. This is a common methodology adopted in evaluating multi-program workloads and is supported by MacSim also. To enable this feature, the repeate_trace knob must be turned on i.e. set repeat_trace to 1. This can be done either via params.in file or from the command line.

4.4.2 x86 Experiments

4.4.2.1 One x86 Core

Usually, cores of type large are configured as x86 cores, however, this is not mandatory.

```
// params/params_x86
num_sim_cores 1
num_sim_small_cores 0
num_sim_medium_cores 0
num_sim_large_cores 1
large_core_type x86
```

4.4.2.2 Multiple x86 cores

For multi-core simulations users have to specify the number of cores as greater than one.

```
// 4-core simulation
num_sim_cores 4
num_sim_small_cores 0
num_sim_medium_cores 0
num_sim_large_cores 4
large_core_type x86
repeat_trace 1
```

4.4.2.3 2-way SMT x86 Core

MacSim also supports the simultaneous multi-threading (SMT) features. These parameters are used for the SMT configurations:

`max_threads_per_core`, `max_threads_per_medium_core`, and `max_thread_per_large_core`.

For example:

```
// 1-core 2-way SMT configuration
num_sim_cores 1
num_sim_small_cores 0
num_sim_medium_cores 0
num_sim_large_cores 1
large_core_type x86
max_threads_per_large_core 2
repeat_trace 1
```

4.4.2.4 x86 Core Parameters

By default these parameter values are applied to large cores.

```
large_width 2                // pipeline width (the entire pipelines use the same width)
large_core_fetch_latency 5   // front-end depth
large_core_alloc_latency 5   // decode/allocation depth
bp_dir_mech gshare           // this is common to all core types
bp_hist_length 14            // branch history length
isched_large_rate 4          // # of integer instructions that can be executed per cycle
msched_large_rate 2          // # of memory instructions that can be executed per cycle
fsched_large_rate 2          // # of FP instructions that can be executed per cycle
large_core_schedule  io      // in order instruction scheduling, set to "ooo" for out of order scheduling
rob_large_size 96            // ROB size
fetch_policy rr              // SMT(MT) thread fetch policy  by default: round-robin, common to all core types
```

4.4.3 GPU Simulations

Usually cores of type small are configured as GPU cores. Several pre-defined parameter files for simulating NVIDIA architectures are provided, below is the list of GPU parameter files provided with MacSim.

File Name	Description
params_8800gt	NVIDIA GeForce 8800GT (G80 architecture)
params_gtx280	NVIDIA GeForce GTX280 (GT200 architecture)
params_gtx465, params_gtx480	NVIDIA GeForece GTX465, GTX480 (Fermi architecture)

Below is some sample configurations:

GPU with One Application

```
// 12-SM simulations
num_sim_small_cores 12
core_type ptx
max_threads_per_core 80    // set the max number of warps per SM
```

GPU with Multiple Applications

```
// 12-SM simulations, 6 SMs for each application
num_sim_cores 12
num_sim_small_cores 12
core_type ptx
max_threads_per_core 80                 // set the max number of warps
max_num_core_per_appl 6                 // 6 SMs for each application
repeat_trace 1                          // for multi-program workload simulation
```

4.4.3.1 Number of Thread Blocks Per Core

The Maximum number of thread blocks per core for a kernel is determined by several factors:

1. number of threads in each thread block
2. number of registers used by each thread
3. amount of shared memory required
4. GPU architecture (the CUDA compute version)[1].

The PTX trace generator calculates the maximum thread blocks per core based on the values provided by Ocelot for these variables includes it the trace output. The calculation is similar to what is done by the CUDA occupancy calculator. Users can override this value for all GPU applications by setting the `max_block_per_core_super` knob.

4.4.3.2 GPU Core Parameters

```
schedule_ratio 4          \\ schedule instructions on every 4 cycle
fetch_ratio  4            \\ fetch new instructions on every 4 cycle
gpu_sched 1               \\ use GPU scheduler for GPU cores
const_cache_size  1024    \\ 1024B constant cache
texture_cache_size 1024   \\ 1024B texture cache (currently, each core has a private texture cache)
shared_mem_size 4096      \\ 4096B shared memory size
ptx_exec_ratio  4         \\ factor by which latency values defined in uoplatency_ptx.def
                          \\ must be multiplied for actual PTX instruction latency
num_warp_scheduler 2      \\ the number of warps to schedule whenever instruction scheduler is run
```

[1] note that the CUDA compute version is set by the user while generating traces

4.4.4 Heterogeneous Architecture Simulations

For CPU-GPU heterogeneous simulations, an architecture similar to Intel's Sandy Bridge [**?**] is modeled. However, the GPU cores in this model are similar to NVIDIA's Fermi [**?**] architecture. Two sample parameter files for a heterogeneous configuration is also provided.

File Name	Description
params_hetero_1_6	1-CPU, 6-GPU cores
params_hetero_4c_4g	4-CPU, 4-GPU cores

Following example shows a simple heterogeneous configuration.

One CPU application + One GPU application

```
num_sim_cores 2
num_sim_small_cores 1
num_sim_medium_cores 0
num_sim_large_cores 1
core_type ptx
large_core_type x86
cpu_frequency 3
gpu_frequency 1.5
repeat_trace 1
```

Although the above configuration sets up the number of CPU and GPU cores correctly, users still have to setup each core types individually. Please refer to sample files for other parameter values.

Multiple CPU applications + Multiple GPU applications

```
num_sim_cores 8
num_sim_small_cores 4
num_sim_medium_cores 0
num_sim_large_cores 4
core_type ptx
large_core_type x86
cpu_frequency 3
gpu_frequency 1.5
repeat_trace 1
```

4.4.5 Cache Configuration

The cache can be configured using the following knobs:

```
l{1,2}_{small, medium, large}_num_set      // number of sets
l{1,2}_{small, medium, large}_assoc        // associativity
l{1,2}_{small, medium, large}_line_size    // cache line size
l{1,2}_{small, medium, large}_num_bank     // number of banks
l{1,2}_{small, medium, large}_latency      // cache latency
l{1,2}_{small, medium, large}_bypass       // cache bypass (if set, always miss)
num_l3                                     // number of l3 cache tiles
l3_num_Set                                 // number of l3 cache sets
l3_assoc                                   // l3 associativity
l3_line_size                               // l3 line size
l3_num_bank                                // l3 number of banks
l3_latency                                 // l3 latency
l{1,2,3}_{read,write}_port                 // the number of read / write port
icache_num_set 8                           // 4KB I-cache number of set
icache_assoc   8 /                         / I cache set associativity
```

The effective cache size can be calculated using Equation 1.

$$cache\_size = num\_set \times assoc \times line\_size \times num\_tiles(l3only, otherwise1) \quad (1)$$

For instance, the size of a cache with 256 sets, 16 ways per set, 64B per cache lines and 4 tiles is $256 \times 16 \times 64 \times 4 = 1MB$

Cache latency is determined by several factors including the size, technology, and the number of ports. Cacti [**?**] can be used to model cache latency accurately. The cache line size is set to 64B by default. Although the cache line size can be any power of 2, all cache levels must have the same cache line size.

4.4.6 DRAM configuration

For configuring the DRAM system including the memory controllers and the DRAM itself, the knobs shown below are available.

```
dram_frequency 0.8              // dram frequency
dram_bus_width 4                // dram bus width
dram_column 11                  // column access (CL) latency
dram_activate 25                // row activate (RCD) latency
dram_precharge 10               // precharge (RP) latency
dram_num_mc 2                   // number of memory controllers
dram_num_banks 8                // number of banks per controller
dram_num_channel 2              // number of dram channels per controller
dram_rowbuffer_size 2048        // row buffer size
dram_scheduling_policy FRFCFS   // dram scheduling policy
```

MacSim models three DRAM timing parameters - precharge (t_{RP}), activate (t_{RCD}), and column access (t_{CL}). While DRAM bandwidth is modeled using the parameters dram_frequency, dram_bus_width, and dram_num_channel. The maximum DRAM bandwidth can be calculated using Equation 2.

$$max\_bandwidth = dram\_frequency \times dram\_bus\_width \times dram\_num\_mc \times dram\_num\_channel \quad (2)$$

For example, the maximum bandwidth of a DRAM system with the above parameter values is

$$800\ \text{MHz}\ (0.8\ \text{GHz})\ *\ 4\ \text{Bytes}\ *\ 2\ \text{MCs}\ *\ 2\ \text{Channels} = 12.8\text{GB/s}.$$

Currently, MacSim provides two memory scheduling policies: FCFS (First-Come-First-Serve) and FR-FCFS (First-Ready First-Come-First-Serve).

5

Simulation Statistics

A simple framework for collecting statistics (hereafter referred to as stats) during simulation is provided. Stats can be either global (includes data from all cores) or per core.

5.1 Stat Types

The following stat types are supported:

COUNT for counting the number of occurrences of an event. (Eg. number of cache hits)

RATIO for calculating the ratio of number of occurrences of one event over another. (Eg. (number of cache hits / number of cache accesses) i.e., cache hit ratio)

DIST for calculating the proportion of each event in a group of events. (Eg. If the user wants to know what percent of L1 data cache accesses (in a 2-level hierarchy) resulted in L1 hits, L2 hits or memory accesses, then the user should define a distribution consisting on three events - L1 hits, L2 hits and L2 misses - and update the counter for each event separately)

Note that a simulation will output two values for each stat. First one is the raw value i.e. the number of occurrences of the event associated with the stat and the second value is the value calculated based on the type of the stat.

5.2 Adding a new stat

New stats can be defined by adding DEF_STAT statements to any of the **.stat.def* files in the *def/* directory or by creating a *.stat.def* file including the definitions in the same directory. To define a per core statistic specify PER_CORE at the end of each DEF_STAT statement. Below is a description and an example of defining stats for different types of stats.

COUNT Stat:

DEF_STAT(STAT_NAME, COUNT, NO_RATIO [, PER_CORE])

Eg:

```
DEF_STAT(INST_COUNT_TOT, COUNT, NO_RATIO)
DEF_STAT(INST_COUNT, COUNT, NO_RATIO, PER_CORE)
```

RATIO Stat:

```
DEF_STAT(STAT_NAME, RATIO, BASE_STAT_NAME [, PER_CORE])
```

In addition to defining the RATIO stat itself, a base stat of type COUNT has to be defined as well. The value of the base stat is used as the denominator in calculating the ratio.

Eg:

```
DEF_STAT(DISPATCHED_INST, COUNT, NO_RATIO)
DEF_STAT(DISPATCH_WAIT, RATIO, DISPATCHED_INST)
```

DIST Stat:

```
DEF_STAT(STAT_NAME_START, DIST, NO_RATIO [, PER_CORE])
DEF_STAT(STAT_NAME, COUNT, NO_RATIO [, PER_CORE])*
DEF_STAT(STAT_NAME_END, DIST, NO_RATIO [, PER_CORE])
```

The definition of a DIST stat requires at least two stats. Eg:

```
DEF_STAT(SCHED_FAILED_REASON_SUCCESS, DIST, NO_RATIO, PER_CORE)
DEF_STAT(SCHED_FAILED_OPERANDS_NOT_READY, COUNT, NO_RATIO, PER_CORE)
DEF_STAT(SCHED_FAILED_NO_PORTS, DIST, NO_RATIO, PER_CORE)
```

5.3 Updating Stats

Macros are provided to update the value of stats. STAT_EVENT and STAT_EVENT_M increment and decrement the value of a global stat by 1 and take the name of the stat to be updated as their argument. STAT_EVENT_N is used to increment the value of a global stat by more than than 1. It takes the name of the stat to be incremented and the value to be added as its arguments. STAT_CORE_EVENT and STAT_CORE_EVENT_M increment and decrement the value of a per core stat by 1. These take core id and the name of the stat to be incremented/decremented as their parameters. For example,

```
STAT_EVENT(INST_COUNT_TOT);          // increments global stat INST_COUNT_STAT by 1
STAT_EVENT_N(INST_COUNT_TOT, 2);     // increments global stat INST_COUNT_STAT by 2
STAT_EVENT_M(INST_COUNT_TOT);        // decrements global stat INST_COUNT_STAT by 1
STAT_CORE_EVENT(0, INST_COUNT);      // increments stat INST_COUNT for core 0 by 1
```

5.4 Simulation output

At the end of a simulation several files with the extension stat.out are generated, these files include the stat values at the end of the simulation. As mentioned, for each stat two values are generated, one is the raw stat value and other is a value calculated based on the type of the stat. For simulations with multiple applications, multiple sets of stat files are generated. Each simulated application is assigned an integer id (these ids are assigned according to the order in which the applications appear in the trace_file_list), when an application terminates (for the first time, note that applications may be repeated), stat files suffixed with the the id of the application, i.e. *.stat.out.<appl_id>, are generated. These stat files contain the value of the stats until that point in the simulation. At the end of the simulation, *.stat.out files are generated as usual.

5.5 Important Stats

In table 3 some important stats are listed.

Table 3. Some important Stats

INST_COUNT_TOT	# of instructions		general.stat.out
INST_COUNT_CORE_[0-11]	# of instructions in only the specificed core [0-11]	core	general.stat.out
CYC_COUNT_TOT	simulated cycles		general.stat.out
CYC_COUNT_CORE_[0-11]	simulated cycles in only the specificed core [0-11]		general.stat.out
CYC_COUNT_X86	simulated cycles for x86 only		general.stat.out
CYC_COUNT_PTX	simulated cycles for ptx only		general.stat.out
	# of fp instructions		
	# of int instructions		
	# of load instructions		
	# of store instructions		
BP_ON_PATH_CORRECT	# of correctly predicted branches (DIST)	core	bp.stat.def
BP_ON_PATH_MISPREDICT	# of mis-predicted branches (DIST)	core	bp.stat.def
BP_ON_PATH_MISFETCHT	# of mis-fetch branches (BTB MISS)(DIST)	core	bp.stat.def
ICACHE_HIT, ICACHE_MISS	# of I-cache hitt,miss (DIST)		memory.stat.def
L[1-3]_HIT_CPU	# of l[1-3]cache hits from CPU		memory.stat.def
L[1-3]_HIT_GPU	# of l[1-3]cache hits from GPU		memory.stat.def
L[1-3]_MISS_CPU	# of l[1-3]cache misses from CPU		memory.stat.def
L[1-3]_MISS_GPU	# of l[1-3]cache misses from GPU		memory.stat.def
AVG_MEMORY_LATENCY	average memory latency		memory.stat.def
TOTAL_DRAM	# of DRAM accesses		memory.stat.def
TOTAL_DRAM_READ	# of DRAM reads		memory.stat.def
TOTAL_DRAM_WB	# of DRAM write backs		memory.stat.def
	# of register reads		
	# of register writes		
COAL_INST, UNCOAL_INST	coalesced/uncoalesced mem requests (DIST)		memory.stat.def

6

MacSim Directory Structure and Source Code

6.1 Directory Structure

The top-level source directory of MacSim contains several directories. The list of directories is following:

bin/	Build output directory
def/	Contains definitions of parameters (see Sections 2.3 and 4) and events for statistics (see Section 5)
doc/	Contains MacSim documentation.
params/	Contains sample parameter (see Sections 2.3 and 4) configuration files.
scripts/	Contains scripts used in build of MacSim.
src/	Contains MacSim source files (.cc and .h files).
tools/	Contains x86 trace generator and trace reader.

Table 4 shows the list of source file and the purpose/content of each file.

Table 4. Source files and their purpose/content

File(s)	Purpose
frontend.cc/h, fetch_factory.cc/h, bp*.cc/h	Fetch stage
allocate*.cc/h, rob*.cc/h, map.cc/h	Decode and Allocate stages
schedule*.cc/h	Schedule stage
exec.cc/h	Execution stage
retire.cc/h	Retire stage
port.cc/h, cache.cc/h dram.cc/h, memory*.cc/h, memreq_info.cc/h, readonly_cache.cc/h, sw_managed_cache.cc/h	Memory system
pref*.cc/h	Prefetchers
trace_read.cc/h inst_info.h	Reading traces
core.cc/h	Class representing a core being simulated
process_manager.cc/h	Process Manager/thread scheduler
uop.cc/h	Uop structure and related enums
macsim.cc/h	Class containing pointers to the simulated cores, NoC, memory system, knobs and other objects
knob.cc/h	Classes for supporting knobs
statistics.cc/h	Classes for supporting knobs
factory_class.cc/h	Implementation of different factory classes
bug_detector.cc/h	Class useful for debugging forward progress errors happen
utils.cc/h	Utility classes and functions
debug_macros.h	Macros for debugging
assert_macros.h	Macros for assert statements with debug information
global*.h	Forward declarations and typedefs

7

Pipeline Stages in MacSim

MacSim implements a five stage processor whose pipeline depth can be varied using knobs. The stages are - Fetch, Decode, Schedule and Execute (includes Memory) and Retire. This chapter describes the basic feature of each stage.

7.1 Fetch Stage

This stage models the instruction fetch stage, its main tasks are as follows:

1. Determine the thread from which instructions will be fetched next (for processors that are not multi-threaded, the same thread is selected always), this involves checking whether a thread can actually fetch a new instruction or not.
2. access I-cache (when there is an icache miss, the front-end cannot fetch instructions for the selected thread).
3. read a uop from trace (call get_uops_from_traces())
4. BTB access and branch prediction - Different branch predictors are implemented using the *bp_factory* and the type of branch predictor to be used is set using the bp_dir_mech knob.
5. send uop to queue meant for modeling the depth of the processor front-end - The depth of the front-end stage is set using one of the fetch_latency knobs.[1] Different thread fetch policies can be implemented, which is described in Section 12.4. The default fetch policy is round-robin and fetch_policy knob will specify the policy to be used.

7.1.1 Reading Traces

Reading of traces is mainly performed in trace_read.cc. The main function of this step during simulation is to read a macro instruction from the trace file and convert it into a sequence of one or more micro-ops (uops). A simple decoding algorithm is used for generating uops. To improve the speed of simulation, decoded uops are stored in a hash table for reuse when the same static instruction is encountered again during simulation. Note that to obtain dynamic information such as addresses accessed, uops must be decoded partially everytime.

[1] There is one knob for each core type, small, medium, and large.

7.2 Decode and Allocate Stage

In this stage the resource requirement for each uop is calculated and space is allocated for the uop in the required structures. At the end of the stage, each uop occupies an entry in ROB and one of the issues queues - general purpose (integer), floating point or memory. Depending on the knob values, there could a unified issue queue or distributed issue queues. For conditional branches with BTB misses, the branch target is resolved in this stage as well.

7.3 Schedule Stage

The schedule stage implements the instruction scheduling logic. Currently in-order, out-of-order and GPU schedulers are supported. The GPU scheduler issues uops from the same warp/SIMD thread in order, but uops from different warps/SIMD threads can be issued out of order in comparison to the order in which they were dispatched. Before executing an uop the instruction scheduler first checks for the availabilty of source operands, and then for the availability of ports (functional units) for uop execution.

7.4 Execution Stage

In this stage, all instructions - both non-memory and memory - are executed. The member variable *m_done_cycle* of the uop structure indicates the cycle in which the uop will complete. The following steps are taken for execution of instructions:

1. For non-memory uops, the latency of the the instruction is determined, an execution port is marked as in use and the *m_done_cycle* of the uop is set. The latencies of uops are defined in the file *../def/uoplatency_ptx.def* which gets included during the compilation of the MacSim binary.

2. Branch instructions are resolved in this stage.

3. For memory uops, D-cache (or the appropriate memory structure such as Const cache, Shared Memory and so on) is accessed and an execution port is marked as in use. If a D-cache port is available, then the port is marked as used, however, if a port is unavailable, the execution of the uop is attempted again later on. On D-cache hit, the *m_done_cycle* of the uop is set. Handling of D-cache misses is explained in Section 8.

4. Uncoalesced memory instructions result in multiple accesses to the the D-cache.

7.5 Retire Stage

This stage handles the in-order retirement of uops. Memory allocated to a uop is freed (returned to memory pool). This stage also checks of completion of warps, blocks, kernels and the entire application itself and triggers the repetition of applications if the knob for repetition is set.

7.6 Queues

frontend queue ia queue rob scheduler

8

The Memory System

8.1 Caches

Each cache structure consists of the cache storage (tag and data) and multiple queues. Figure 3 shows the overall structure of a cache in MacSim. There are two flows: 1) cache access flow: from a processor or upper level cache miss, to access the cache and 2) cache fill flow: in case of a cache miss, data is supplied from the lower level cache or DRAM. Section 8.1.1 details all queues and Section 8.1.2 lists the flows through the queues and the cache.

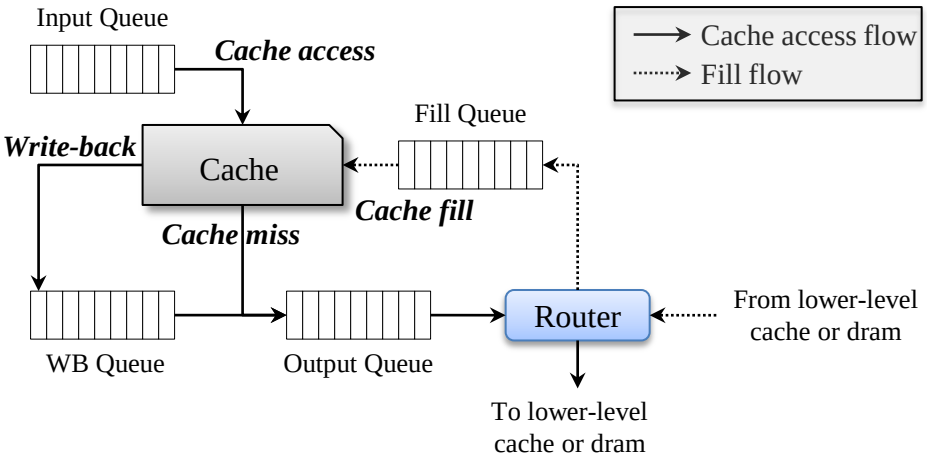

Figure 3. The cache structure.

8.1.1 The Queues

All cache accesses flow from one queue to another.

- input queue - requests forwarded to this cache due to upper-level cache misses are inserted into this queue.
- output queue - requests that miss in this cache are inserted into the output queue to be forwarded to a lower-level cache . If no lower-level cache is available, then requests are forwarded to DRAM.
- write-back queue - MacSim models write-back caches. When a dirty cache line is evicted, the line must be written back into the next level cache (or DRAM). All write-back requests are initially inserted into the write-back queue.

- fill queue - data returned from the next level cache or DRAM is inserted into the fill queue before updating the cache.

8.1.2 The Flows

- Upper-level cache to input queue : forwarding of upper-level cache misses
- Upper-level cache to fill queue : forwarding of upper-level write-back requests
- input queue to cache : accessing the cache
- cache to output queue : cache miss, access lower-level cache
- cache to write-back queue : generating write-back requests
- write-back queue to output queue : write-back request, access the lower-level cache
- output queue to the router : access the lower-level cache through the on-chip interconnection network
- router to the fill queue : the data from the lower-level cache or DRAM

8.2 The Hierarchy

MacSim is very flexible in its support for different memory hierarchies. Each level in the cache hierarchy can be configured independently of other levels (see Section 4.4.5). Figure 4 shows the base memory hierarchy without DRAM memory.

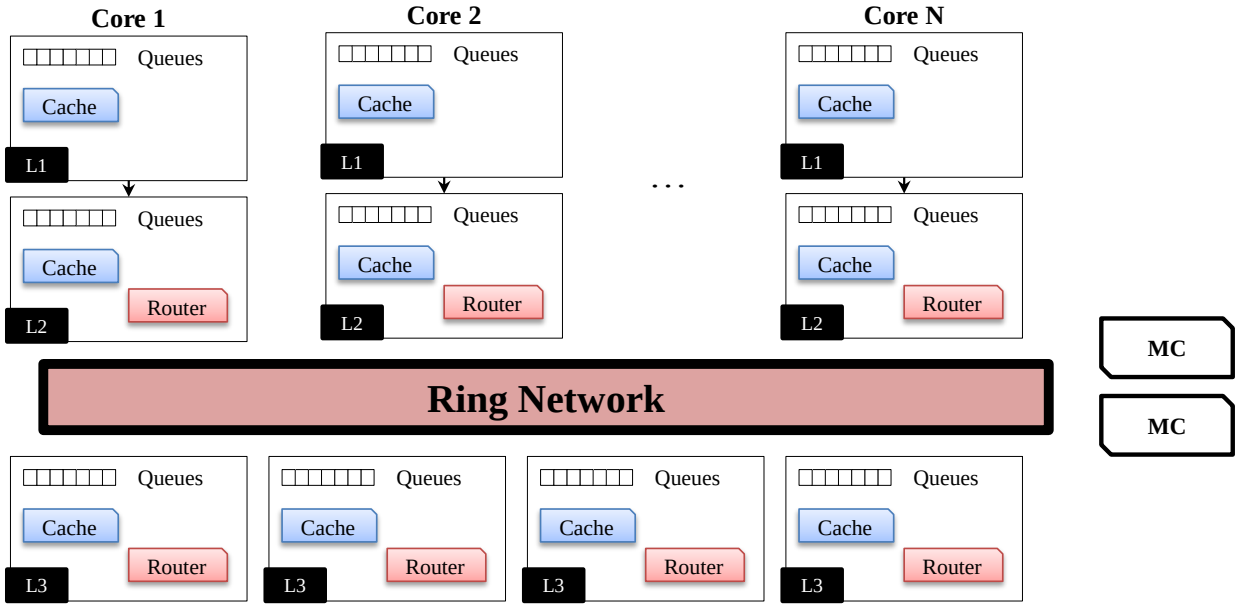

Figure 4. The memory system in MacSim.

- There are three levels (L1, L2, and L3) of caches in the MacSim, the L2 cache can be disabled if needed by some configurations.

- All the caches and memory controllers can be connected via an on-chip interconnection network (currently, the default topology is ring), if the configuration allows.
- L1 and L2 caches are always private to each core.
- The local router within a cache structure is enabled only when necessary.
- L3 cache is unified (shared by all cores), but sub banked. In other words, address regions are statically partitioned and each tile is responsible for sub regions. Each cache tile has multiple banks as well.

Figure 5 shows how to configure a 2-level cache hierarchy. Even though there are 3-levels of cache, the L2 cache is disabled and only its router is used by the L1 cache to access the interconnection network. Since there is no additional latency between L1 and L2 when the L2 is disabled, we can flexibly configure 2-level cache hierarchies.

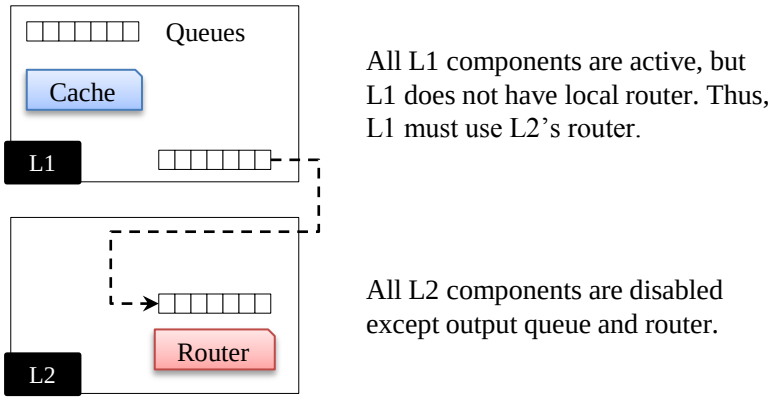

Figure 5. 2-Level cache hierarchy.

8.3 Configuring the Cache Hierarchy

The cache hierarchy can be configured by 1) setting the link between different cache levels, 2) disabling cache levels and 3) enabling/disabling routers. The following code is used in MacSim for cache initialization.

```
void dcu_c::init(
  int next_id, // next level cache id
  int prev_id, // previous level cache id
  bool done,
  bool coupled_up, // direct link with upper level cache
  bool coupled_down, // direct link with lower level cache
  bool disable, // disable cache
  bool has_router // router
);
```

Enabling/disabling router When `has_router` is set to *false*, the cache cannot directly access to the on-chip interconnection. Instead, it has to go through lower-level cache's interface. Therefore, `coupled_down` must set to *true* and appropriate `next_id` must be set. In this way, the output queue of the cache is connected directly to the input queue of the next level (lower) cache.

Disabling cache levels When disable is set to *true*, the cache is disabled. When a request is inserted into the input queue of a disabled cache, the request will be directly inserted into the output queue of the cache. This feature has been used for modeling 2-level cache hierarchies. As Figure 5 shows, L1 and L3 caches are active, but L2 cache is disabled. However, since the L1 cache does not have a router, it needs to use the router of the L2 cache. Therefore, has_router must be set to *true* for the L2 cache.

Links As mentioned in the Section 8.2, the L1 and L2 caches are always private to a core. All L1 misses should go through the L2 cache without accessing the interconnection network. To this end, a direct link must be set between the L1 and L2 caches. Therefore, coupled_down and next_id must be set for the L1 cache and coupled_up and prev_id must be set for the L2 cache. Note that the link is always bi-directional.

8.3.1 Configuring Different Cache Hierarchies with MacSim

- Intel Nehalem [**?**] and Sandy Bridge [**?**] microarchitectures have three-levels of caches. The last-level cache is tiled and to access the closest L3 tile, a L2 cache does not have to go through the interconnection, it can access the L3 tile directly. However, if a L2 cache accesses a remote L3 tile, it goes through the interconnection. Note that the number of cores (L1 and L2 caches) and the number of the L3 tiles must be the same in this configuration.

```
// class l3_coupled_network_c
for (int ii = 0; ii < m_num_core; ++ii) {
  // next_id, prev_id, done, coupled_up, coupled_down, disable, router
  m_l1_cache[ii]->init(ii, -1, false, false, TRUE, false, false);
  m_l2_cache[ii]->init(ii, ii, true,  TRUE,  TRUE, false, TRUE);
  m_l3_cache[ii]->init(-1, ii, false, TRUE,  false,false, TRUE);
}
```

- NVIDIA G80 [**?**] architecture does not have hardware-managed caches. Thus, all three levels are connected each other and disabled. Only the L3 cache has a router to access the DRAM.

```
// class no_cache_c
for (int ii = 0; ii < m_num_core; ++ii) {
  // next_id, prev_id, done, coupled_up, coupled_down, disable, router
  m_l1_cache[ii]->init(ii, -1, false, false, TRUE,  TRUE, false);
  m_l2_cache[ii]->init(ii, ii, true,  TRUE,  TRUE,  TRUE, false);
  m_l3_cache[ii]->init(-1, ii, false, TRUE,  false, TRUE, TRUE);
}
```

- NVIDIA Fermi [**?**] architecture has private L1 caches and a unified L2 cache shared by all cores. The L1 and L2 caches are linked. The L2 cache is been disabled, but has is router enabled. The L3 is not linked with others (it is connected via the interconnect), but it is enabled and has a router.

```
// class l2_decoupled_network_c
// next_id, prev_id, done, coupled_up, coupled_down, disable, router
for (int ii = 0; ii < m_num_core; ++ii) {
  m_l1_cache[ii]->init(ii, -1, false, false, TRUE,  false, false);
  m_l2_cache[ii]->init(-1, ii, true,  TRUE,  false, true,  true);
}

for (int ii = 0; ii < m_num_l3; ++ii) {
```

```
  // next_id, prev_id, done, coupled_up, coupled_down, disable, router
  m_l3_cache[ii]->init(-1, -1, false, false, false, false, true);
}
```

- In a general 2-D Topology (Mesh, Torus), it is assumed that each core has private L1 and L2 caches, but access to the L3 cache must be through the interconnection network. The L1 and L2 caches are both enabled and linked, but only the L2 cache has a router. The L3 cache is not linked with other caches and the communication is made through the interconnection network.

```
// class l3_decoupled_network_c
// next_id, prev_id, done, coupled_up, coupled_down, disable, router
for (int ii = 0; ii < m_num_core; ++ii) {
  m_l1_cache[ii]->init(ii, -1, false, false, TRUE,  false, false);
  m_l2_cache[ii]->init(-1, ii, true,  TRUE,  false, false, TRUE);
}

for (int ii = 0; ii < m_num_l3; ++ii) {
  // next_id, prev_id, done, coupled_up, coupled_down, disable, router
  m_l3_cache[ii]->init(-1, -1, false, false, false, false, TRUE);
}
```

8.4 DRAM Module

MacSim also models detailed memory controllers which consider DRAM timing constraints and bandwidth specifications when scheduling requests. Section 4.4.6 describes how to configure DRAM parameters.

8.4.1 DRAM Timing Constraints

We model following three timing constrainsts of DRAM.

Timing	Knob variable in MacSim	Symbol	Description
Precharge	KNOB_DRAM_PRECHARGE	T_{RP}	Row precharge time
Activate	KNOB_DRAM_ACTIVATE	T_{RCD}	Row address to column address delay
Column Access	KNOB_DRAM_COLUMN	T_{CL}	CAS latency

8.4.2 DRAM Bandwidth

DRAM bandwidth can be modeled using several factors:

Factor	Knob variable in MacSim
DRAM Frequency	KNOB_DRAM_FREQUENCY
DRAM Data bus width	KNOB_DRAM_BUS_WIDTH
Number of dram controllers	KNOB_DRAM_NUM_MC
Number of dram channels Column Access	KNOB_DRAM_NUM_CHANNEL

, where Bandwidth = FREQUENCY * BUS_WIDTH * NUM_MC * NUM_CHANNEL.

8.4.3 The Structure of DRAM Controller

A DRAM controller consists of multiple banks and one or more channels. Each bank has its own request buffer and request scheduler. The bank scheduler picks a request to service based on the policy.

- The bank scheduler picks a request based on the policy (FCFS, FRFCFS, ...) if no request is being served currently.
- The channel scheduler picks a request from command-ready banks usually based on the oldest request firs. Based on the command, appropriate timing constraint (precharge, activate, or column access) is enforced on the bank.
- Once the column access signal is sent, the data is prepared from the DRAM chip (load) or the data is sent to the DRAM chip (store). Among multiple data-ready banks, the channel scheduler picks a request based on the policy (oldest-first).
- When the data is ready/sent for a request, the data is supplied to the cache (load) or the request is completed (store).
- If there are memory requests with the same address, these requests are merged into one dram request.

9

Supporting GPUs

MacSim can model GPUs similar to NVIDIA's Tesla [**?**] and Fermi [**?**] architectures. In these architectures the GPU consists of a scalable number of *Streaming Multiprocessors* (SMs), each containing several*Scalar Processors* (SP) and Special Function Units (SFUs), a multithreaded instruction fetch and issue unit, a read-only constant cache, and a read/write scratch pad memory called Shared Memory. In MacSim, SMs are modeled as cores and SPs as just one of the functional units.

The behavior of certain components is different when simulating GPUs, these differences are described briefly below.

9.1 Traces

In NVIDIA GPUs threads are executed in batches of 32 threads called *warps*. For simulating execution at warp granularity, for GPU applications, we generate traces at warp granularity. What this means is there is one trace file per warp and each instruction in a trace file represents an instruction executed by a warp. The instruction in the trace file encodes which threads in the warp were active during the execution of the instruction. For divergent branches, all instructions from one path are written to the trace file first and only then are instructions from the other path written to the trace file. For memory instructions, multiple instructions are written to the trace file since each thread in a warp can access different addresses. Handling of memory instructions inside the simulator is explained below.

9.2 Process Manager

The Process Manager (explained in Chapter 11) does assignment of threads to cores (SMs) at block granularity - all warps in the same thread block are assigned to the same core. The number of blocks that can be assigned to each core is determined from the resource requirement of each block and the GPU architecture being simulated. This number is calculated at the time of trace generation for compute devices of capabilility 2.0 and is included in the trace files; it can also be specified by setting the knob `max_block_per_core_super`. For each block, the Process Manager tracks the number of completed warps and when all warps in a block have completed, the Process Manager retires the block and assigns a new block to the core. The policy for assigning blocks to cores can be varied - blocks could either be assigned statically to cores, or they could be assigned to the first core that can accomodate a new block.

9.3 Memory Hierarchy

The following new components are added to the memory hierarchy - Shared Memory, Const Cache, Texture Cache.

9.4 Instruction Scheduler

A GPU-specific (or a multithreaded architecture specific) instruction scheduler is necessary, since neither single-threaded in-order nor single-threaded out-of-order schedulers match the requirement of a GPU scheduler. A GPU scheduler must be able to schedule instructions from different warps while scheduling instructions from the same warp in-order. A GPU-specific scheduler is implemented in schedule_smc.cc, this scheduler chooses instructions from different warps in .

9.5 Simulating Memory Instructions

Due to the execution of threads at warp granularity, the execution of a memory instruction generates multiple addresses, one from each thread. In the trace file all the memory accesses are included as individual trace instructions, but the accesses are marked as generated from the same memory instruction. During simulation when it is detected that a sequence of memory accesses are from the same memory instruction, the trace is read ahead until the end of the sequence and the individual memory accesses are attached to a main uop as child uops. It is the main uop that flows through the pipeline stages until execution. In the execution stage it is detected that the main uop has several child uops and the child uops that are issued to the memory hierarchy. When all the child uops have completed, the main uop is ready for retirement.

10

MacSim Internals

This chapter briefly discusses some internals of MacSim including data structures and their purposes, important functions and components that will help users understand MacSim better.

10.1 run_a_cycle() function

All pipeline stages, units such as cache and memory controller, components such as memory system and core implement the *run_a_cycle()* function which is called every cycle. Within the *run_a_cycle()* function of each component is the processing done by the component for every simulation cycle. *macsim_c*, the simulation object, calls *run_a_cycle()* for the memory system, interconnect and cores in the simulation (see *macsim.cc*). Each core in turn calls *run_a_cycle()* for the individual pipeline stages (see *core.cc*).

10.2 Important data structures and classes

10.2.1 macsim_c

For each simulation, an instance of *macsim_c* is created. This instance is responsible for initializing the cores, the memory system, the interconnection network and knobs, for registering classes/functions implementing various policies with the factory classes and so on. Every cycle, *macsim_c* ticks all the other simulation components via the *run_a_cycle()* function.

10.2.2 core_c

Each core being simulated is represented by an object of *core_c* declared in *core.h*. Besides pointers to various pipeline stages and hardware units, *core_c* tracks information such as how many threads are assigned to the core, how many instructions have been fetched for each thread and so on.

10.2.3 trace_read_c

An single instance of *trace_read_c* is created at startup and is used for reading the traces of all threads/warp in a simulation. To amortize the overhead of reading traces, instructions are read from the trace file in chunks instead of single instructions.

Some of the important functions in *trace_read_c* are:

get_uops_from_traces() called by front-end stage to fetch uops for a thread. If a decoded instruction is available, the next uop from the decoded instruction is returned to the front-end, otherwise the next instruction from the trace is decoded and the first uop from the decoded instruction is returned to the front-end.

read_trace() returns the next instruction from the trace buffer which contains a chunk of instructions read from the trace file. If the trace buffer is empty, a chunk is read from the trace file and the first instruction is returned.

convert_pinuop_to_t_uop() decodes an instruction from a trace into a sequence of *trace_uop_s* instances and returns the sequence. Instructions being decoded for the first time have their decoded information stored in a hashtable so that decoding does not have to be repeated when the same instruction is encountered again in the trace.

10.2.4 map_c

An instance of *map_c* is created for each core in the simulation. This instance tracks data and memory dependences between the uops of a thread for all threads assigned to a core. After a dependence is identified, the source uop is added to a list of uops which should complete execution before the dependent uop becomes ready for execution.

10.2.5 uop_c

An instance of *uop_c* defined in *uop.h* is allocated (from a pool) for each uop and is populated with all the information about the uop. Each uop is assigned a unique number, *m_unique_num*, that is unique across all uops in a core. Each uop is also assigned a unique number, *m_uop_num*, that is unique across a uops of a thread. To identify a uop from a thread of a core, *m_core_id*, *m_thread_id*, *m_uop_num* member variables of the uop have to checked. Users can define new member variables to collect/track information as a uop moves through the pipeline. The *uop_c* object allocated for a uop is returned to the *uop_c* pool when the uop retires. Whenever a new member variable is added to *uop_c* users should make sure that the member variable is initialized to a default value in *uop_c::init()*.

10.2.6 mem_req_s

For each memory request (L1 data cache miss, and write back requests), a instance of *mem_req_s* declared in *memreq_info.h* is allocated. *mem_req_s* contains all information relevant to a memory request and an instance allocated to a request is freed only when the memory request completes and data is filled in the cache. Users can define new member variables to collect/track information as a memory request moves through the memory hierarchy.

10.2.7 drb_entry_s

For each memory request that reaches the DRAM an instance of *drb_entry_s* declared in *dram.h* is allocated. This instance contains information about the bank, row and column of the DRAM request among other details and is deallocated (freed) when the DRAM request is serviced.

10.2.8 rob_c/smc_rob_c

rob_c declared in *rob.h* and *smc_rob_c* declared in *rob_smc.h* are for ROBs in CPUs and GPUs respectively. uops are allocated an entry in the ROB in the allocate stage and the allocated entry is reclaimed when the uop retires.

10.2.9 pool_c

pool_c defined *utils.h* is a utility class meant for creating memory pools. Using memory pools reduces the overhead of frequent memory allocation and deallocation. In Macsim memory pools are used for uops (*uop_c*), threads/warps (*thread_s*) and so on.

10.2.10 pqueue_c

pqueue_c is a utility class for varying the length of the simulated pipeline. In MacSim, it is used to vary the length of the fetch (and decode) and allocation stages. At the time of creation of a *pqueue_c* object, user has to specify the latency of the pqueue in cycles. Objects/values enqueued into the are ready to be dequeued after speficied latency. Besides the object to be enqueued, the enqueue operation takes a priority value as argument as well. Different priority values can be used for objects enqueued in the same cycle to control which object gets dequeued first from the pqueue.

10.3 Instructions Latencies

Latencies of X86 and PTX uops are defined in *uoplatency_x86.def* and *uoplatency_ptx.def* files in *trunk-/def*. Users must edit these files to modify the latencies of instructions. Note that for PTX instructions, the value defined in *uoplatency_ptx.def* is multiplied by the value of the knob *PTX_EXEC_RATIO*.

The process manager which is responsible for assigned threads (thread blocks) to cores and data structures related to it are discussed in Chapter 11.

11

Process Manager/Thread Scheduler

MacSim uses a common Process Manager/Thread Scheduler for both CPU threads and GPU warps. For each application that is to be simulated, the Process Manager creates a process and also creates the threads/warps in the application. Based on the simulation configuration, the Process Manager assigns cores to each application, these cores are dedicated to the application. In case of CPU applications, only the main thread of the application is launched first. The trace config that is input to the simulator specifies in terms of instructions executed by the main thread when each child thread should be started. When a child thread becomes ready for execution, the Process Manager is responsible for assigning the child threads to cores. In case of GPU applications, the Process Manager creates warps and forms thread blocks from these warps. The thread blocks are assigned to cores according to the maximum number of blocks supported by the core. Though the term core is commonly used for both x86 cores and GPU cores, x86 thread can run only on a core specified as x86 and a warp/block can run only a core specified as a GPU core. Threads/warps once assigned to a core, remain attached to the core until they terminate. When a thread or a warp terminates, the Process Manager is invoked again for updating bookkeeping information. When it is determined that an application can terminated, based on the simulation parameters, the Process Manager could repeat the simulation of the application until the termination condition is met.

Some of the key data structures involved are:

process_s For each application that is to be simulated, the process manager creates a instance of type process_s. This structure includes information such as process id, start information of threads, number of threads (warps) in the application, number of threads (warps) created, number of threads (warps) terminated, list of cores on which the application can run and so on.

thread_s This structure is analogous to the task struct maintained by an operating system kernel. Each CPU thread and GPU warp has an instance of thread_s structure. This structure includes fields for thread id (warp id), block id, pointer to trace file, process to which the thread belongs and other fields.

block_schedule_info_s Bookkeeping structure representing thread blocks in a GPGPU application. This structure contains fields for number of warps in a block, number of terminated warps, core to which the block is assigned and so on.

thread_trace_info_node_s Wrapper structure around thread_s used by Process manager to track threads/warps yet to be assigned cores.

process_manager_c The class representing the process manager, an instance of this class is created by the single instance of *macsim_c* that is created for each simulation. Its operation is explained above, some of the functions in *process_manager_c* are:

create_process() allocates *process_c* node for an application. For CPU applications, *create_process()* reads the trace config file and determines the number of threads in the application and their start information. On the other hand, for GPU applications, it reads the trace config file of the first kernel in the application and determines the number of warps and thread blocks in the kernel (calls *setup_process()* to do this).

terminate_process() cleans up some data structures allocated for a process and saves stats if it was the first run of a application. For GPU applications, *terminate_process()* terminates a kernel and calls *setup_process()* which traces the trace config file for the next kernel, determines the number warps and thread blocks in the kernel and calls *create_thread_node()*.

create_thread_node() is called for each thread/warp when it becomes ready for launch (note that all warps in a kernel are ready when the kernel is launched). *create_thread_node()* allocates a *thread_trace_info_node_s* node for a thread/warp and inserts the node into *m_thread_queue* for threads and *m_block_queue* for warps.

create_thread() allocates an initializes a *thread_s* node, it also opens the trace file for the thread-/warp for reading.

terminate_thread() terminates a thread/warp, deallocates (returns to memory pool) data structures allocated for the thread/warp and calls *sim_thread_schedule()*. For GPU applications, *terminate_thread()* retires a thread block if all warps in the block have completed.

sim_thread_schedule() assigns a thread to a core or a thread block to a core if the number of threads/blocks assigned to the core are fewer than the maximum allowed.

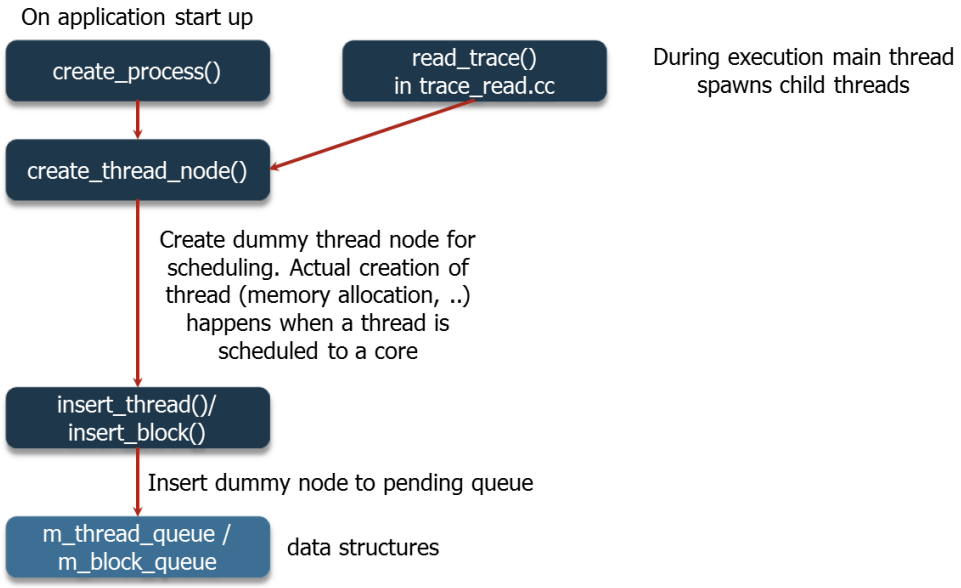

Figure 6. Process and Thread Creation

Figure 6 shows the control flow in the Process Manager on simulation start up and during simulation. *create_process()* is called for each application to be simulated. *create_process()* calls *setup_process()* which in turn calls *create_thread_node()* for the main thread of the application. *create_thread_node()* allocates a *thread_trace_info_node_s* instance for the main thread and inserts it into *m_thread_queue*. During the simulation of the main thread when it is detected in *read_trace()* that a child thread should be spawned, *create_thread_node()* for the child thread is called. The flow is similar for GPU kernels except that warps

are inserted into *m_block_queue* instead of *m_thread_queue*. Also, since GPU kernels do not have a main thread, on kernel start up, *create_thread_node()* is called for the first warp in the kernel.

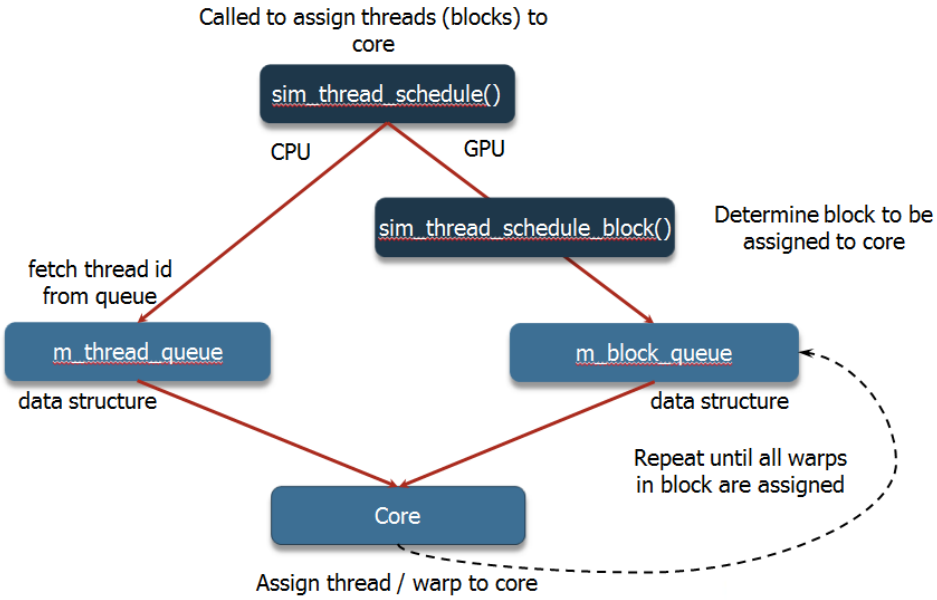

Figure 7. Thread/Thread Block Scheduling

When a thread/warp running on a core terminates, *sim_thread_schedule()* is called to schedule new threads/thread blocks onto the core. Note that a new thread block can be assigned to a core only when all warps of a previously assigned block have terminated. *sim_thread_schedule()* removes threads/warps from *m_thread_queue*/*m_block_queue* and assigns them to the core. This flow is shown in Figure 7.

12

Adding to MacSim

12.1 New DRAM Policy

To implement a new DRAM scheduling policy, a class that extends *dram_controller_c* has to be defined. This class should overload the *schedule()* function which should return the next DRAM request to be serviced according to the policy. Please refer to class *dc_frfcfs_s* defined in dram.cc/h for a sample. In addition to the class, define a wrapper function that returns an instance of the class (see *dram.cc* for an example) and register the class with the DRAM factory (see *see register_functions() in macsim.cc*.

12.2 New Instruction Scheduler

Define a class that extends *schedule_c* and overloads at least the *run_a_cycle()* function. Other functions may be overloaded to maintain a consistent code structure across the different instruction schedulers. After defining the new scheduler, add code to *core.cc* to instantiate the new scheduler instead of one of the provided schedulers.

12.3 New policy for assigning thread blocks to GPU cores

In *process_manager_c::sim_schedule_thread_block()* add (or modify) code to determine the id of the block that will be assigned to the core based on the new policy. The list of ready blocks can be obtained via *m_block_schedule_info* member of the *macsim_c* object that is created for the simulation.

12.4 New Fetch Policy

There are several steps to add a new thread fetching policy. We will provide an example with round-robin fetch policy, which is already provided by MacSim.

Step 1. Define a new fetch policy in class frontend_c.

```
// frontend.h
class frontend_c {
  ...
  int fetch_rr(void);
  ...
}
```

Step 2. Implement this new policy.

Step 3. Register this new policy in macsim.cc.

```
fetch_factory_c::get()->register_class("rr", fetch_factory);
```

Step 4. Assign this policy to MT_fetch_Scheduler in frontend.cc.

```
if (policy == ``rr'') {
  MT_fetch_scheduler = &frontend_c::fetch_rr;
}
```

Step 5. Enable this new branch predictor by setting KNOB_FETCH_POLICY.

```
./macsim --fetch_policy=rr      -- in the commandline
fetch_policy rr                 -- in the params.in
```

12.5 New Branch Predictor

There are several steps to add a new branch predictor. We will provide an example with gshare [**?**] branch predictor, which is already provided by MacSim.

Step 1. Define a new class which inherits class bp_dir_base_c

```
class bp_gshare_c : public bp_dir_base_c {};
```

Step 2. Define and implement other necessary features of a new branch predictor. Especially, override pred(), update(), and recover() functions.

```
uns8 bp_gshare_c::pred(uop_c* uop) {...}
void bp_gshare_c::update(uop_c* uop) {...}
```

Step 3. Add a policy to the branch predictor factory in macsim.cc and to the allocator in bp.cc.

```
// macsim.cc macsim_c::register_functions()
bp_factory_c::get()->register_class(``gshare'', default_bp);

// bp.cc
bp_dir_base_c* default_bp(macsim_c* simBase)
{
  ...
  else if (bp_type == ''gshare'')
    new_bp = new bp_gshare_c(simBase);
  ...
}
```

Step 4. Enable this new branch predictor by setting KNOB_BP_DIR_MECH.

```
./macsim --bp_dir_mech=gshare      -- in the commandline
bp_dir_mech gshare                 -- in the params.in
```

12.6 New Hardware Prefetcher

There are several steps to add a new branch predictor. We will provide an example with a stride [**?**] branch predictor, which is already provided by MacSim.

Step 1. Define a new class which inherits class pref_base_c

```
class pref_stride_c : public pref_base_c {};
```

Step 2. Define and implement other necessary features of a new hardware prefetcher. Especially, override several function.

```
void init_func(int);                          // on initiation of prefetch request
void done_func();                             // when a prefetch request is installed
void l1_miss_func(int, Addr, Addr, uop_c*);   // on L1 miss
void l1_hit_func(int, Addr, Addr, uop_c*);    // on L1 hit
void l1_pref_hit_func(int, Addr, Addr, uop_c*); // on L1 prefetch hit
void l2_miss_func(int, Addr, Addr, uop_c*);   // on L2 miss
void l2_hit_func(int, Addr, Addr, uop_c*);    // on L2 hit
void l2_pref_hit_func(nt, Addr, Addr, uop_c*); // on L2 prefetch hit
```

Step 3. Add a prefetcher to the prefetcher factory in pref_factory.cc.

```
// pref_factory.cc
void pref_factory(...)
{
  pref_base_c* pref_stride = new pref_stride_c(...);
  pref_table.push_back(pref_stride);
  ...
}
```

Step 4. Enable this new hardware prefetcher by setting KNOB_PREF_STRIDE_ON (please refer to pref_stride.cc pref_stride_c::pref_stride_c()). *Note that the way we simulate hardware prefetchers are not same as other modules. We will provide the detail of hardware prefetcher in Section* **??**.

```
switch (type) {
  case UNIT_SMALL:
    knob_enable = *m_simBase->m_knobs->KNOB_PREF_STRIDE_ON;
      break;
    case UNIT_MEDIUM:
      knob_enable = *m_simBase->m_knobs->KNOB_PREF_STRIDE_ON_MEDIUM_CORE;
      break;
    case UNIT_LARGE:
      knob_enable = *m_simBase->m_knobs->KNOB_PREF_STRIDE_ON_LARGE_CORE;
      break;
}

./macsim --pref_stride_on[,medium_core, large_core]=1
pref_stride_on[,medium_core, large_core] 1
```

13

Debugging

TBD

13.1 Forward Progress Error

TBD

13.2 Debugging with Debugging Messages

To check the correctness of the module, We provide debugging features with text-based debugging messages. For example,

```
debug message example
```

You can define anywhere. Opt version will nullify these statement for performance purpose.
enable variable
To enable debug messages `make dbg` and

```
debug_cycle_start
debug_cycle_end
```

debug macros are defined in `macsim-top/trunk/def/src/debug_macros.h`
debug macro is very similar to `printf` function.

```
_debug(debug_variable, const char * format, ... )


#define DEBUG(args...) _DEBUG(*m_simBase->m_knobs->KNOB_DEBUG_MEM, ## args)
```

useful data for

```
core id
thread id
uop num
inst num
vaddr
req_id
```

To add a new debug stage, you need to add a debug knob variable in debug.param.def

14

FAQ

This chapter answers some frequently asked questions.

Q 1. I'd like to add a new file in Macsim. What should I do?

1. First add files in src directory.
2. Add an entry in macsim/trunk/.obj/Makefile.am.

```
all_sources = \
  ... \
  ../src/new_file.cc
```

Bibliography

A

Coding Style Guideline

A.1 Naming Conventions

Names should be as descriptive as possible, do not hesitate to use long names. Shorten names only when the purpose of the name can still be understood clearly.

A.1.1 Classes

- **Class names** should use lower case alphabets only. If the name consists of multiple words, then the words should be separated using the underscore character. Class names should end with c. Eg.

```
class my_class_c;
```

- **Class member variables** should use the prefix m . Member variable names should use lower case alphabets only, multiple words should be separated using the underscore character. Eg.

```
int m_instance_variable;
```

- **Accessor i.e, get/set, member functions** should use the names of the corresponding member variables without the m suffix. In addition, the set functions should use the prefix set , while no prefix is needed for the get functions. Eg.

```
int m_my_variable;
...
...
void set_my_variable(int);
int my_variable();
```

- **Other member functions** should always start with a verb, multiple words should be separated with the underscore character and no upper case letters are allowed. Eg.

```
int access_cache();
```

- Example class declaration

```
class my_class_c {
  private:
    int m_var_one;
    bool m_var_two;
    ...
```

```
    ...

    void set_var_one(int);
    int var_one();
    ...
    int check_values();
};
```

A.1.2 Structures

- Structure names should use lower case alphabets only. If the name consists of multiple words, then the words should be separated using the underscore character. Structure names should end with s. Eg.

```
struct my_struct_s;
```

- Do not use any prefix or suffix for structure member variables.
- Example structure declaration

```
typedef struct my_type_s {
  type field_one;
  type field_two;
} my_type_s;
```

A.1.3 Other Types

Other user defined types should use only lower case letters with multiple words being separated by the underscore character. These types should use the suffix t. Eg.

```
typedef std::map<int, std::list<int> > my_type_t;
```

A.1.4 Global Variables

Global variable names should use the prefix g and use lower case letters only, multiple words should be separated by the underscore character. Eg.

```
int g_my_global_variable;
```

A.1.5 Local Variables

Local variables follow the same convention as global variables, except that they do not require the g prefix. Eg.

```
int my_local_variable;
```

A.1.6 Constant Variables

Constant variables follow the same convention as global variables, except that they use the prefix k , instead of g . Eg.

```
const int k_my_const = 25;
```

A.1.7 Macros

Macro names should use upper case letters only with multiple words being separated my underscores. Eg.

```
#define MAX_NUM_THREADS 8192
```

A.2 Formatting and Readability

A.2.1 Indentation

Use only spaces for indentation, do not use tabs. The indentation must be two spaces wide.

A.2.2 Variable Declaration

Variables should be declared at the point in the code where they are needed, and should be assigned an initial value at the time of declaration.

A.2.3 Control Statements and Function Definitions

- For *if*, *for*, *while* and *switch* statements, there should be a single single space between the keyword and the opening parenthesis. The opening curly brace should appear on the same line as the statement and there should be a single space between the closing parenthesis and the opening curly brace. The opening curly brace following a do statement must also follow the same rules. The *while* of a *do while* loop should be on the same line as the closing parenthesis of the loop body and should be separated from the closing parenthesis by a single space. Eg.

```
//correct
if (condition) {
  statement 1;
  statement 2;
}

//correct
for (int ii = 0; ii < N; ++ii) {
  statement 1;
  statement 2;
}

//correct
while (condition) {
  statement 1;
  statement 2;
}
```

```
//correct
switch (variable) {
  condition 1:
    statement 1;
    break;
  condition 2:
    statement 2;
    break;
  default:
    statement 3;
    break;
}

//correct
do {
  statement 1;
  statement 2;
} while (condition);
```

- *else* statements should be on the line following the closing parenthesis of the corresponding if block. There should be a single space between the else keyword and the opening curly brace of the else block. The same rules apply for *else if* statements as well. Eg.

```
//correct
if (test_cond) {
...
}
else {
...

}

//correct
if (test_cond) {
...
}
else if (another_cond) {
...
}
```

- When defining functions, there should be no spaces between the function name and the opening parenthesis, there should be a single space between the closing parenthesis and the opening curly brace which should appear on the same line as the function name. Eg.

```
//correct
int my_func_defn(int arg_a, int arg_b) {
  ...
}
```

A.2.4 Horizontal Spacing

Except for when using the operators listed below use a single space between the operator and the operands (note the slight difference in using the comma operator). For the operands listed below, spaces should not be used. List of operators which do not need horizontal spacing:

```
() [] ->    ++ -- + - ! ~  (type)  *  &   sizeof
```

Eg.

```
i = j + k * i;      // correct
i = j+k*i;          // wrong

array[i]  = j;      // correct
array [i] = j;      // wrong

my_func(a, b, c);  // correct
my_func(a,b,c);    // wrong
```

A.2.5 Statements

Each statement should be on its own line, this applies to variable declarations as well. Eg.

```
int i = 0;          // correct
int j = 0;          // correct
int i = 0,  j = 0;  // wrong
i++;                // correct
j++;                // correct

i++;  j++;          // wrong
i = 0;  j = 0;      // Wrong
```

A.2.6 Column Length

Lines should not exceed more than 80 characters in length. If a line is longer than 80 characters, split it into multiple lines.

A.3 Header Files and Include Directives

- All header files should have an "include guard" to prevent accidental inclusion of the file multiple times in a single compilation unit. The include guard should be named after the file name. Eg.

  ```
  // in  pref_common.h
  #ifndef  PREF_COMMON_H  #define  PREF_COMMON_H
  ...
  ...
  #endif
  ```

- Avoid including header files in other header files by using forward declaration.
- All include directives should be grouped with system files listed first, followed by normal simulator header files.

A.4 Comments

Use Doxygen style comments. More to be added.

A.5 Other Rules

- Avoid C style coding, try to build and use classes as much as possible.
- Define types which will not have functions other that accessor functions (and constructor and destructor) as structures. Use classes only if functions other than set/get will be needed.
- Definition of constructors and destructors (for both classes and structs) and class member functions should not be included in header files, unless the definitions are very small.
- Declare all global variables in global vars.h and define them in main.cc.
- More to be added.