MacSim: A CPU-GPU Heterogeneous Simulation Framework

Hyesoon Kim
Jaekyu Lee
Nagesh B. Laksiminarayana
Jaewoong Sim
Jieun Lim

December 8, 2011

Contents

1 Introduction

MacSim is a heterogeneous architecture simulator, specifically supporting x86 ISA and NVIDIA PTX ISA. It is a trace-driven cycle-level simulator. It can simulate homogeneous ISA multicore simulations, heterogeneous ISA multicore simulations. It uses Ocelot for PTX trace generation and Pin to generate x86 traces. Both traces are converted internal RISC style uops and those uops are simulated. MacSim is a microarchitecture simulator that simulates detailed pipeline (in-order and out-of-order) and a memory system including caches, NoC, memory controllers. It supports, asymmetric multicore configurations (small cores + medium cores+ big cores) and SMT or MT architectures as well.

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

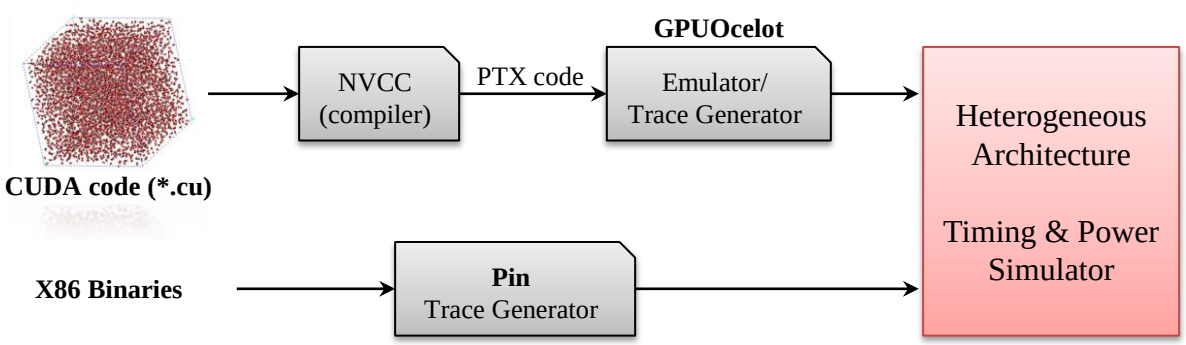

Figure 1. The overview of MacSim Simulator

Figure 1 shows the overview of Macsim simulator.

2 Macsim Installation and Run

2.1 Download

Macsim source code is maintained using the subversion. You can check out the Macsim copy by

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

2.2 Wiki and Other Supports

We manage the google project page in the following url:

```
http://code.google.com/p/macsim/
```

2.3 Build Requirement

Macsim requires following to build properly.

1. Operating System - Currently, we only support linux distributions. Tested systems are as follows.

   ```
   Ubuntu
   Redhat (todo)
   ```

2. Compiler - Any compiler that can supports the C++0x (or C++11) standard library. Currently, we tested

   ```
   gcc
   icc (todo)
   ```

3. Autotools - You need to have autotools (automake, autoconf, ...) version 2.65 or higher. You can install autotools by

   ```
   Ubuntu: apt-get install autotools-dev automake autoconf
   Redhat: todo
   ```

2.4 Installation

The GNU Autotools (automake, autoconf) have been used for building Macsim simulator. After initial check out of the Macsim copy, following commands are necessary.

```
aclocal
automake
--add-missing
autoconf
./configure
make
```

You can combine above commands in a line:

```
aclocal && automake --add-missing && autoconf && ./configure && make
```

We provide autogen.sh script file to simplify the building process.

```
./autogen.sh
make
```

The binary *macsim* will be generated in the *trunk/bin/* directory.

2.5 Build Types

We provide three different build types.

- opt : default, optimized version (-O3 flag)
- dbg : debug version (-g3 flag)
- gpf : gprof version (-pg flag)

To build a certain type, you need to specify the option after *make* command. For example,

```
make opt
make dbg
make gpf
```

2.6 How To Run Macsim

To run *macsim* binary, two additional files are required in the same directory.

- params.in - defines architectural parameters that will overwrite the default value.
- trace_file_list - defines the number of traces to run and the path of each trace

2.6.1 params.in

We provide sample parameter files for various architectures (Intel CPUs, NVIDIA GPUs, ...) in the *macsim-top/trunk/params*. Following is sample content of params.in file.

```
# Simulation Configuration
num_sim_cores 1
num_sim_small_cores 0
num_sim_medium_cores 0
num_sim_large_cores 1
core_type ptx
large_core_type x86
cpu_frequency 4
gpu_frequency 1.5
sim_cycle_count 0
max_insts 500000000
heartbeat_interval 1000000
forward_progress_limit 50000
```

```
# Common Core Configuration
fetch_policy rr
mt_no_fetch_br 1
one_cycle_exec 0
```

The first literal is the name of parameter and the second literal is a value that will overwrite the default value. Section 8 details how to add, modify, and use these parameters.

2.6.2 trace_file_list

Following is the content of a sample trace_file_list.

```
2
/trace/ptx/cuda2.2/FastWalshTransform/kernel_config.txt
/trace/ptx/cuda2.2/BlackScholes/kernel_config.txt
```

The first line is the number of traces that Macsim will run and following lines are the path of each traces. Each top level trace configuration file contains various information. Following is the content of a top-level trace information file.

```
1 x86
0 0
```

In the first line, the first number indicates the number of threads in the application and second literal specifies the type of the application. Trailing lines are the information of each thread (thread id, thread starting point in terms of the instruction count of the main thread (thread 0)).

2.6.3 Run

The following is the sample command lines to run Macsim simulator.

```
./macsim [additional commands]
./macsim --l2_assoc=8 --stdout=stdout --stderr=stderr
./macsim --l1_assoc=4
```

3 Trace Generation

Explain how to generate traces and the format of the trace.

3.1 CPU (X86) Traces

Describe pin

3.2 GPU (PTX) Traces

GPU traces are generated using Ocelot [], which is a dynamic compilation framework for heterogenous system.

3.2.1 Installing Ocelot

Ocelot can be installed either using ubuntu packages provided by the official page or via its SVN repository. To install the latest version, check out source files using the following command:

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

After checking out source files, you can build Ocelot and trace generator libraries using the following commands:

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

3.2.2 Generating Traces

CUDA executables which are targeted to generate traces should be linked against libocelot.so and libocelotTrace.so.

In order to generate traces, the following four environment variables need to be set.

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

The following shows an example of how to set up the environment variables.

```
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
```

The kernel information file (kernel_info) should contain the following information: kernel name, register usage (per thread), and shared memory usage (per thread).

```
_Z9Memcpy_SWPfS_i 14 52
```

Running CUDA application creates a directory with the kernel name, where traces are generated, and kernel_config.txt which is a configuration used for MacSim.

```
drwxr-xr-x 2 anonymous group 69632 Sep 00 22:03 _Z9Memcpy_SWPfS_i_0
-rw-r--r-- 1 anonymous group 66 Sep 09 22:29 kernel_config.txt
```

3.3 Trace Format

The generated MacSim traces contain various information to execute an instruction. Tables 1 and 2 show the trace format for each instruction and the description of each field, respectively.

Table 1. MacSim trace format.

nSR	nDR	SR_IDs	DR_IDs	BrType	bImm	Opcode	bStore	bFP	WF	nLD
InstSize	LAddr_1	LAddr_2	SAddr	PCAddr	BrAddr	MemRSize	MemWSize	RepDir	BrActT	

Table 2. Descriptions of each field in the trace.

Field	Size (Bytes)	Description
nSR	1	number of source registers
nDR	1	number of destiantion registers
SR_IDs	9	source register IDs
DR_IDs	6	destination register IDs
BrType	1	branch type
bImm	1	indicates whether this instruction has immediate field
Opcode	1	opcode
bStore	1	indicates whether this instruction has store operation
bFP	1	indicates whether this instruction is FP operation
WriteFlag	1	write flag
nLD	1	number of load opeartions
InstSize	1	instruction size
LAddr_1	4	load address 1
LAddr_2	4	load address 2
SAddr	4	store address
PCAddr	4	PC address
BrAddr	4	branch target address
MemRSize	4	memory read size
MemWSize	1	memory write size
RepDir	1	repetition direction
BrActT	1	indicates whether branch is actually taken

While running a simulation, each instruction in the above *raw trace* is converted into one or more micro-ops. MacSim stores such decoded micro-uops into a MacSim-specific trace structure to facilitate the simulation by populating each field, which is shown in Table 3.

Table 3. 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 Process and Thread Scheduler

Describe process and thread scheduler

5 The Pipeline Stages

Describe pipeline stages

5.1 Fetch Stage

5.2 Decode Stage

5.3 Allocate Stage

5.4 Schedule Stage

5.5 Execution Stage

5.6 Retire Stage

6 The GPU Model

We model our GPU cores similar to NVIDIA Fermi [1].

6.1 The Shared Memory (Scratchpad Memory)

6.2 The Texture Memory

6.3 The Constant Memory

7 The Memory System

7.1 The Cache Structure

Each cache structure consists of the cache and multiple queues. Figure 2 shows the overall cache structure in the Macsim. There are two flows: 1) cache access flow: from a processor or upper level cache miss, try to access the cache and 2) cache fill flow: in case of a cache miss, the data is supplied from the lower level cache or DRAM. Section 7.1.1 details all queues and Section 7.1.2 describes the flows between queues and a cache.

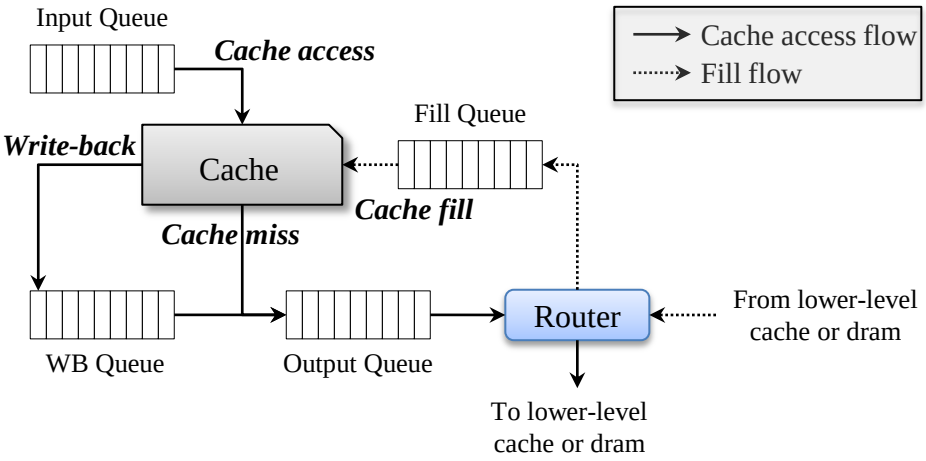

Figure 2. The cache structure.

7.1.1 The Queues

All cache accesses flow from one queue to the other queue.

- input queue - to access a cache. Requests due to upper-level cache misses are inserted in this queue.
- output queue - as a result of a cache-miss, a request will be inserted into the output queue to be supplied data from the lower-level caches or DRAM.
- write-back queue - In Macsim, we model write-back caches. When a cache line is evicted and dirty, this line needs to be written back in the lower level. All write-back requests are initially inserted into the write-back queue.
- fill queue - requests in this queue tries to fill the data that is returned from the lower-level cache or DRAM.
- coherence queue - this queue is intended for handling coherence traffics, but this is currently not modeled.

7.1.2 The Flows in the Cache Structure

- Upper-level cache to input queue : upper-level cache miss
- input queue to cache : access the cache
- cache to output queue : cache miss and lower-level cache access
- cache to write-back queue : write-back requests
- write-back queue to output queue : to access lower-level cache
- output queue to the router : to 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

7.2 The Hierarchy

Macsim maintains very flexible memory hierarchy.

8 The Knobs

To control various architectural parameters, we use multiple knob variabls, which are defined in *trunk/def/*.param.def*.

8.1 How to add a new knob

A new knob variable can be defined in the following format:

```
param<{variable used in the \SIM}, {variable used in the commandline}, {type}, {default value}>
```

We recommend first two arguments are same, but the former is in the upper case and the latter is in the lower case. For example,

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

However, one restriction is this new varialbe must be defined in *trunk/def* directory and the file name should be **.param.def* to parse variables correctly.

8.2 How to use a new knob in the simulator

All knob variables will have *KNOB_* prefix. For example, L2_ASSOC in Section 8.1 will be KNOB_L2_ASSOC in the simulator code.

8.3 How to apply different value to a knob variables

There are two ways of modifying the default value of a knob variable.

1. params.in - this file must be supplied to the macsim binary for the execution. You can find sample parameter files in *trunk/params* directory. In this file, variable names and the values are paired in each line. For example,

   ```
   l1_assoc 8
   l2_assoc 16
   ```

2. command line - instead, you can also supply the knobs in the command line, for example,

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

9 The Statistics

We provide a simple framework for collecting statistics during simulation. Statistics can be either global (common counter for all cores) or per core (each core has its own counter).

9.1 Statistic Types

The following statistic types are supported:

COUNT for counting the number of occurances of an event. Eg. number of cache hits.

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

DIST for defining a group of related events and calculating the number of occurances of each event in the as a percent of the sum of the number of occurances of all events in the group. Eg. If we want to know what percent of L1 data cache accesses (in a 2-level hierarchy) resulted in L1 hits, L2 hits or memory accesses, we should define a distribution consisting on three events - L1 hits, L2 hits and L2 misses - and update the counter for each event correctly.

Note that the simulation will output two values for each statistic. One is the raw value i.e. the number of occurances of the statistic and the other output value is the value calculated based on the type of the statistic.

9.2 How to add a new stat

New statistics can be defined by adding definitions to any of the **.stat.def* files in the *trunk/def/* directory or by creating a *.stat.def* file including the definitions in the *trunk/def/* directory. In order to define a per core statistic specify PER_CORE at the end of each DEF_STAT statement below.

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)
```

9.3 How to update statistics

Macros are provided to update the value of statistics. STAT_EVENT and STAT_EVENT_M increment and decrement the value of a global statistic by 1 and take the name of the stat to be updated as their one and only argument. STAT_EVENT_N is used to increment the value of a global statistic by more than than 1. It takes the name of the statistic to be incremented and the value by which the statistic is to be incremented as its parameters. 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 statistic to be incremented/decremented as their parameters.

Eg:

```
STAT_EVENT(INST_COUNT_TOT)
STAT_EVENT_N(INST_COUNT_TOT, 2)
STAT_CORE_EVENT(1, INST_COUNT)
```

9.4 The Output

At the end of the simulation *.stat.out are generated, these files include the statistic values at the end of the simulation. As mentioned, for each stat two values are generated, one is the raw statistic value and other is a value calculated based on the type of the stat.

10 SST-Macsim

Macsim is a part of SST simulation framework [2].

10.1 How to install SST

Follow the instructions provided in wiki page at the *sst-simulator* GoogleCode repository [3].

10.2 How to install Macsim in SST

Check out the svn copy of Macsim in *sst-top/sst/elements* and apply a 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
```

10.3 How to configure the Macsim SST component

An example SST sdl configuration file is as follows:

```
<?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 what traces are run on each Macsim instance.

10.4 How to run a Macsim simulation in SST

Ensure 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]
```

References

[1] NVIDIA. Fermi: Nvidia's next generation cuda compute architecture. http://www.nvidia.com/fermi. 13

[2] Sandia National Laboratories. SST. http://sst.sandia.gov. 19

[3] Sandia National Laboratories. SST. http://code.google.com/p/sst-simulator. 19