Manual for Saturn

March 21, 2018

Developed by:

Department of Informatics, Technische Universitt Dortmund, Germany

Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany

Contents

1 Introduction

1.1 What is Saturn

Saturn is an integrated, powerful, and efficient simulator. It consists of three well-known simulators, namely, Sniper [1], a many core processor simulator; McPAT [3], a power dissipation simulator, and HotSpot [2, 4], a thermal simulator. Saturn establishes an integrated environment for developing and evaluating power and thermal management algorithms for many-core processors, which realizes a dynamic closed loop processing control flow and provides completed interfaces as well as tracing mechanism for designers.

Due to the diversity of the hardware architectures and the uncertainty of the workloads, processor and system designers prefer to apply simulation-based method to design and evaluate power and thermal management algorithms. Simulation-based development relies on three key components: a processor simulator, a power dissipation simulator and a thermal simulator. The relationship among these three components is outlined in Figure 1, which moreover models the actual workflow of the simulation-based method. More precisely, the processor simulator generates a specific processor to which the power and thermal management algorithm is applied and on which, in addition, several benchmarks are executed. By means of the respective algorithm, the performance of each core encompassed by the processor can be traced as well, whereas the power simulator retrieves the power dissipation based on the current configuration (including, e.g.,frequencies) as well as the workload of distinct cores on the particular processor. After that, the thermal simulator can calculate the temperature for different cores of the processor according to the power dissipation vector. Subsequently, the temperature information is sent back to the processor whereas the management algorithm can change the workloads or adjust the DVFS level of different cores based on the received temperature information for the purpose of cooling down the many-core processor and improving its performance. However, the main disadvantage of the simulation-based method is that all three simulators are employed independently and the output data of one simulator needs to be passed on to the next one manually. The non-existent integration of the simulators makes the data transmission difficult and introduces additional overheads. For instance, the designers must consider the data format used in the transmission. In addition, if the temperature of at least one core cannot meet the requirements, the management algorithm must be modified and the simulation must be restarted manually. Furthermore, such a static developing process is unable to simulate the dynamic operation of power and thermal management algorithms, e.g.,the adjustment of the workloads and reduction of the frequencies for overheating cores.

In Saturn, these three simulators have been integrated. We provide all the necessary interfaces for implementing the power and thermal management algorithms inside, and customize interfaces are possible as well. After implementing the management algorithm, all the aforementioned processes can be executed periodically and automatically in Saturn. The modifications of the configuration e.g., frequency, the workload, and the temperature of each core are recorded automatically as well, so that the designers can evaluate their management algorithms after each simulation according to the all the tracing files.

1.2 Features

- Fully integrated: Three separated simulators that are used for power and thermal management algorithm development have been integrated. The whole workflow shown in Figure 1 can be executed periodically and automatically.
- Closed-loop control: Due to the periodic and automatic execution of the workflow, the power consumptions respond to the change of frequencies and workloads immediately. There is almost the linear relationship between power and temperature. Then, the management algorithm can modify the frequencies and workloads according to the temperature information. Thus, the online closed-loop developing process has been established.
- Online monitoring: Three features, e.g., frequencies, power consumptions, and temperatures for all the cores are traced at the same time. Designers can observe the dynamics of these three features together after each simulation, so that they can evaluate the performance of different management algorithms.

2 Getting Started

2.1 Preparation

We recommend to compile Saturn based on Ubuntu 14.04 and stay connected to the Internet all the time in order to smoothen the setup process. Due to the dependencies, you also need to install the following packages: build-

essential, libbz2-dev, libboost-dev, libsqlite3-dev, gfortran, m4, xsltproc, libxi-dev, zlib1g-dev, libxmu-dev, gettext.

After preparing the system, please download all the following source files which are needed for Saturn:

- sniper-6.1
- pin-2.14-67254-gcc.4.4.7-linux.tar.gz
- sniper-benchmarks.tbz
- HotSpot-5.0.2.tar.gz
- Eigen 3.2.0
- Saturn-patch.tar.gz

Before compiling, please execute the following steps to make all source files ready:

Step 1: Create a folder (e.g., called "Saturn") , extract "sniper-6.1", "benchmarks", and "Saturn-patch" inside.

Step 2: Copy all files from the folder "Saturn-patch/configurations" to this directory.

Step 3: Apply the patch "sniper_base.patch" to "sniper-6.1", e.g.,

```
cd sniper-6.1
patch -p1 < ../sniper_base.patch
```

Step 4: Extract the archive "pin-2.14-67254-gcc.4.4.7-linux.tar.gz", rename it as "pin_kit", and copy it into "sniper-6.1".

Step 5: Create a folder named "HotSpot" under path: "sniper-6.1/common/scheduler/HotSpot". Copy all ***.c** , ***.h**, and **LICENSE** files from "HotSpot-5.0.2.tar.gz" into that folder and apply our patch "hotspot.patch".

Step 6: Copy the whole folder named "Eigen" from "Eigen 3.2.0" to the directory "sniper-6.1/common/scheduler/Eigen"

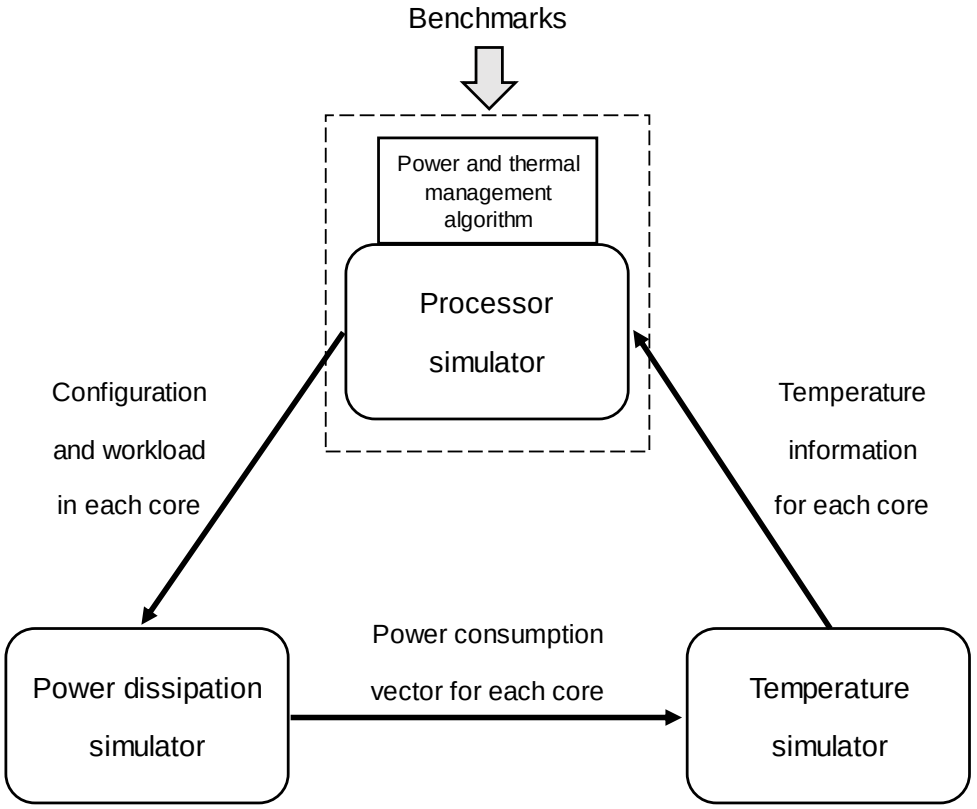

Figure 1: Workflow for simulation-based power and thermal management algorithm development.

2.2 Compiling Saturn

Saturn consists of two main components, namely, the integrated simulator and the benchmarks. Due to the dependencies, you have to compile them separately.

2.2.1 Compiling the Sniper

First, the parameters in the main configuration file should be set correctly. Please note that the location of the main configuration file is currently hard-coded in file "sniper/sniper-6.1/common/scheduler/configurationParameters.cc", "sniper-6.1/config/buildconfig.sh, buildconfig.makefile" as well as in "sniper/saturn.config". Please modify the paths and let them point to the right path (e.g., "/home/yourname/Saturn/saturn.config").

After that, compile the Sniper using:

```
cd sniper-6.1
make clean
make
```

2.2.2 Compiling the Benchmarks

At first, please apply our patch "benchmakrs_local.patch" under the path "sniper/benchmarks/local", which contains the demo scheduler of Saturn. Then, comment the cpu 2006 since it is payed, comment the parsec since it cannot be compiled under Ubuntu14.04 in the **Makefile**. Finally, execute:

```
export GRAPHITE_ROOT=// path / to / sniper -6.1
export BENCHMARKS_ROOT=$(pwd)
make
```

2.2.3 Compiling the Benchmark "parsec" Separately

If you need the benchmark parsec, we provide a way to compile it using Ubuntu-12.04.

Step 1: Boot the system by Ubuntu 12.04 using a live CD or bootable USB, click "Try Ubuntu".

Step 2: Mount the main disk where the source code is stored. "**sudo fdisk -l**" to show all disks in your system, and "**sudo mount -t ntfs-3g /dev/sdaX**" to mount the desired disk (X indicates the name of your desired disk).

Step 3: Update the system using "**sudo update**", then install all the dependencies in 2.1.

Step 4: Switch to the benchmarks folder on your main disk, (Usually, the disk is mounted under the path: "/media/.../home/yourname/.../benchmarks)". Modify the **Makefile**, since you only need to compile the benchmark parsec.

Step 5: Finally, execute the code like in the normal compiling process:

```
export GRAPHITE_ROOT=// path / to / sniper -6.1
export BENCHMARKS_ROOT=$(pwd)
make
```

3 Customizing Power and Thermal Management Algorithms in Saturn

In this section, we explain how to implement a new management algorithm in Saturn. To run a simulation with the new integration, you only need to execute:

```
./run-sniper --benchmarks=splash2-fft-small-2,splash2-fft-small-4\
-n 64 -c gainestown -c noc -senergystats\
-sstattrace:core.energy-dynamic::1000000 --sim-end=last
```

The only difference to native Sniper is that the following commands are added:

```
-senergystats -sstattrace:core.energy-dynamic::1000000
```

These commands make McPAT execute periodically whereat the number at the end of **energy-dynamic::number** should match that of **-samplingInterval** in the configuration file. Then, we use **- -sim-end=last** to get the execution time output trace of all applications one time (writte in file **-throughputOut**). When an application finishes, it may be re-executed by Sniper, depending on the value of the **- -sim-end** configuration variable. Particularly, in case **- -sim-end=first** or **- -sim-end=last**, applications are not re-executed. However, in case **- -sim-end=last-restart**, applications are re-executed automatically by Sniper when they finish. Either way, variable $m\_num\_apps\_nonfinish$ keeps track of the number of applications that have to finish at least one time. Therefore, the value of $m\_num\_apps\_nonfinish$ is used in combination with **- -sim-end** for the cases in which **- -sim-end=last** or **- -sim-end=last-restart**. In other words, when **- -sim-end=first**, Sniper finishes its execution when the first application finishes. However, when **- -sim-end=last** or **- -sim-end=last-restart**, Sniper finishes its execution when all applications have finished at least one time (in case –sim-end=last-restart, they might have finished more than once). Because of this, the value of $m\_num\_apps\_nonfinish$ is only decreased the first time each application finishes.

In the configuration file, Saturn provides another option for the users named **-schedulerAppRunning**. If you only want to use the integrated simulator without any additional scheduler, you can set this option to "0". Nevertheless, if you want to apply a scheduler written at the application level, then you need to set the option to "1".

To enable the execution of the above, the scheduler application should be compiled in the benchmark's local folder. For example, a scheduler called **myScheduler** should be placed inside a folder in path "sniper/benchmarks/local/myScheduler". Furthermore, before compiling and running the scheduler, the "sniper/benchmarks/local/_init_.py" file, and the "sniper/benchmarks/local/Makefile" file should be modified. Particularly, several lines should be added to the Makefile, like:

```
include ../tools/hooks/buildconf.makefile
all:
        make -C pi
        make -C myScheduler
clean:
        make -C pi clean
        make -C myScheduler clean
```

You should add the new scheduler to the "sniper/benchmarks/local/_init_.py" file:

```
def allbenchmarks():
        return [ 'pi', 'myScheduler' ]
```

Then add the handling of the corresponding case, for example:

```
elif program == 'myScheduler':
        num_cores = nthreads
        return '%s/myScheduler/ myScheduler %s' % (HOME, num_cores)
```

Finally, the users should compile the benchmarks in the local folder, either running "make" on the sniper/benchmarks folder or on the sniper/benchmarks/local folder.

4 Additional Functions and Tracing Files

Besides the original APIs and trace mechanisms provided by Sniper, Saturn provides additional APIs and trace mechanisms, in order to implement customized power and thermal management algorithms, trace and evaluate their performances.

4.1 Additional Functions in Sniper's API

In order to implement a new performance, power, and thermal aware scheduling algorithm, Saturn provides several new useful interfaces. Through these interfaces, designers can not only obtain the information from the simulator, but also modify the configuration and change the workload of the cores in the simulator. All functions added to Sniper's API have been listed as follows:

- **SimGetTemperature(proc):** Returns the current temperature of the core selected by the specified ID. Note that the returned temperature is an integer that holds the value in microKelvin. Hence, the temperature value should be divided by 1000000.0 to convert it back to Kelvin.

- **SimGetOwnTemperature():** Returns the current temperature of the core from where the function is called.
- **SimGetPower(proc):** Returns the current power consumption of the core specified by the respective ID.
- **SimGetOwnPower():** Returns the current power consumption of the core from where the function is called.
- **SimSetThreadAffinity(thread, affinity):** Sets the affinity of a thread ID to a certain core ID. Sniper will then migrate the thread to the specified core in the next appropriate time interval.
- **SimSetOwnAffinity(affinity):** Sets the affinity of the thread that calls this function to a certain core ID. Sniper will then migrate the thread to the specified core in the next appropriate time interval. This is usually used when the scheduler/power manager wants to be executed in a certain core.
- **SimGetThreadAppId(thread):** Returns the application ID to which the specified thread ID belongs to.
- **SimIsThreadFinished(thread):** Returns whether a certain thread (identified by the given ID) has finished execution. Note that Sniper does not delete thread IDs from its thread list (similar to process IDs in Linux).
- **SimIsApplicationFinished(app):** Returns whether a certain application (identified by the given ID) has finished execution. Note that Sniper does not delete application IDs from its application list (similar to process IDs in Linux).
- **SimGetIps(proc):** Returns the current Instructions per Second (IPS) of the specified core identified by a given ID.
- **SimGetAccumulatedInstructions(proc):** Returns the total number of executed instructions (accumulated since the beginning of the simulation) of the specified core identified by the given ID.
- **SimGetOwnAccumulatedInstructions():** Returns the total number of executed instructions (accumulated since the beginning of the simulation) of the core from where the function is called.
- **SimSetIslandBoosting(island, boostingIsland):** Sets the status of Turbo Boost (similar to Intel's Turbo Boost) for the specified voltage island.
- **SimActivateIslandBoosting(island):** Activates Turbo Boost (similar to Intel's Turbo Boost) for the specified voltage island.
- **SimDeactivateIslandBoosting(island):** Deactivates Turbo Boost (similar to Intel's Turbo Boost) for the specified voltage island.
- **SimReserveCoreForApplication(proc,app):** Reserves a core for a certain application for newly created threads. That is, threads that were already spawned before this command was issued should be migrated with SimSetThreadAffinity(). However, when one or more cores are reserved for a certain application, newly created threads will have their affinities automatically set to one of the reserved cores, which can save an additional thread migration. In case no cores are reserved for a certain application, then the default affinity selection of Sniper is used.
- **SimGetAverageTemperatureLastWindow(unit):** Returns the average temperature of a core (or floorplan unit if the index corresponds to a block in the floorplan that is not a core) during the last (fixed-time and hard-coded) time window. The size of the window is specified by WINDOW_SIZE and the duration of the window corresponds to (WINDOW_SIZE * samplingInterval).
- **SimGetAverageIpcLastWindow(proc):** Returns the average Instructions per Cycle (IPC) of the specified core during the last (fixed-time and hardcoded) time window.

4.2 Tracing Files

Saturn also builds its own trace mechanism with the purpose of monitoring the whole process of the power and thermal management algorithm. Such trace files can help us observe the dynamics of the performance, power consumption as well as the temperature of different cores in the system. All tracing files managed by **saturn.config** are listed in the following:

- **matexConfig:** Location of the HotSpot/Matex configuration.

- **floorplan:** Location of the floorplan file. Please keep in mind that currently the blocks in the floorplan should be ordered in a certain way. Particuarly, all cores should come first, arranged in the same order that Sniper would use as core IDs, followed by the remaining of the blocks in the floorplan like caches, routers, etc.
- **scheduleOut:** Schedule trace file, where we write the schedule traces of every core, that is, the application ID and thread ID executed on every core at a given time . If no file is given, then the output trace file is not generated.
- **frequencyOut:** Frequency trace file where we write the frequency traces of every core. If no file is given, the output trace file is not generated.
- **powerOut:** Power trace file where we write the power consumption traces on every block.
- **energyOut:** Energy trace file where we write the energy consumption traces on every block.
- **accEnergyOut:** Energy trace file where we write the accumulated energy consumption traces on every block.
- **ipsOut:** IPS trace file where we write the instantaneous (between time intervals) Instructions per Second (IPS) traces on every core .
- **ipcOut:** IPC trace file where we write the instantaneous (between time intervals) Instructions per Cycle (IPC) traces on every core .
- **throughputOut:** Application execution time trace: File containing the total execution time of every application
- **instOut:** Instruction trace file where we write the instructions traces of every core.
- **accInstOut:** Accumulated instruction trace file where we write the accumulated instruction traces of every core.
- **tempOut:** Temperature trace file where we write the temperature traces of every block.
- **maxTempOut:** Maximum temperature trace file where we write the traces with the maximum temperatures among all blocks .
- **schedulerAppRunning:** Indicates if an application executes as scheduler (value: 1) or not (value: 0). If yes, then this application should not be counted to terminate Sniper since it does not stop any way.
- **schedulingAlgorithm:** Not used right now
- **coresPerIsland:** Not used right now. Read directly from Sniper's configuration.
- **dtmEnabled:** Not used right now.
- **t_amb:** Ambient temperature in °C.
- **t_dtm:** Temperature for triggering Dynamic Thermal Management (DTM) or Turbo Boost in °C.
- **samplingInterval:** Sampling interval for the output traces (in ns)

5 Case Studies

The same as what we has shown in our paper (we can prepare them later).

References

[1] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[2] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st annual Design Automation Conference*, pages 878–883. ACM, 2004.

[3] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[4] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 2–13. IEEE, 2003.