Non-intrusive DVFS emulation in gem5 with application to self-aware architectures

Haririan, P.
Inst. of Electrodynamics & Microelectron., Univ. of Bremen, Bremen, Germany
Garcia-Ortiz, A.

Abstract

The emergence of multi-processor systems-on-chip (MPSoC), the close interaction between the hardware architecture and the operating system, and the increasing requirements for energy efficiency are exacerbating the need for full-system architecture simulators such as gem5. A commonly-used technique for power optimizations is dynamic voltage and frequency scaling (DVFS). Native gem5 does not support DVFS. This work presents an efficient approach for prototyping and emulating DVFS in gem5. The approach is non-intrusive, easily modifiable and flexible. Several experimental results are provided to assess the efficiency of the method. Finally a test-case related to self-aware systems is presented to highlight the flexibility of the approach.

Published in:
Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014, 9th International Symposium on
Date of Conference: 26-28 May 2014
Page(s): 1 - 7
INSPEC Accession Number: 14468900
Conference Location: Montpellier
DOI: 10.1109/ReCoSoC.2014.6861356
Publisher: IEEE
Non-Intrusive DVFS Emulation in gem5 with Application to Self-Aware Architectures

Parham Haririan, Alberto Garcia-Ortiz
Institute of Electrodynamics and Microelectronics, University of Bremen
{haririan, agarcia}@item.uni-bremen.de

Abstract—The emergence of multi-processor systems-on-chip (MPSoC), the close interaction between the hardware architecture and the operating system, and the increasing requirements for energy efficiency are exacerbating the need for full-system architecture simulators such as gem5. A commonly-used technique for power optimizations is dynamic voltage and frequency scaling (DVFS). Native gem5 does not support DVFS. This work presents an efficient approach for prototyping and emulating DVFS in gem5. The approach is non-intrusive, easily modifiable and flexible. Several experimental results are provided to assess the efficiency of the method. Finally a test-case related to self-aware systems is presented to highlight the flexibility of the approach.

Keywords—DVFS; architecture; full-system; simulator; self-aware architecture; emulation; MPSoC; low-power

I. INTRODUCTION

Current technology trends advocate for the use of multi-processor architectures. Performance and specially power considerations motivate chip designers to integrate an increasing number of computing cores onto a single die [1]. The design and analysis of such multi-processor architectures at high levels of abstraction is a major challenge. On the one hand, the communication infrastructure and memory hierarchy can originate a severe bottleneck. This can be analyzed (and mitigated) only if a full system simulation with realistic test benches is used. On the other hand, the close interaction between the hardware architecture and the operating system needs to be properly addressed. Again, it requires full system simulation. The analysis gets even more intricate in novel energy-efficient architectures which use hardware-dedicated modules for the management; as for example, in hardware-dedicated self-configurable management approaches [2][3] and hardware operating systems [4].

Several architecture simulators have been proposed. Among them, gem5 [7] represents an excellent alternative; it allows the analysis of different micro-architectures ranging from Alpha to ARM, it provides a flexible set of models for the interconnects and memory hierarchy, and it allows full-system level simulation. Several groups have demonstrated empirically the high correlation between the performance metrics provided by gem5 and actual measurements in real systems [6].

One of the commonly-used techniques for reducing the energy consumption of multi-processor architectures is Dynamic Voltage and Frequency Scaling (DVFS). The idea is to exploit the trade-off between energy consumption and performance by dynamically adjusting the voltage and frequency of the processing cores to the demands of the application. Multi-processors require specialized algorithms to exploit the advantages of DVFS while meeting performance requirements and maximizing possible energy savings [5].

Beside its importance, gem5 lacks the support to incorporate DVFS natively. Recently, some efforts have been made to palliate this deficit [8]. However, the approach in [8] requires a sophisticated and dedicated extension of the gem5 infrastructure; moreover, it focuses on a DVFS mechanism using OS kernel frameworks.

In this work we propose a non-intrusive approach for the emulation of DVFS mechanism in gem5. It exploits the runtime reconfiguration capabilities of gem5 to emulate the dynamic feature of DVFS. The contributions of this work are:

• An emulation methodology for DVFS in gem5
• A concrete implementation of the proposed methodology for the Alpha processor; it includes the modifications in gem5 to incorporate additional pseudo-instructions and the addition of Python source code to perform the frequency scaling.
• An example of the methodology in the context of hardware-dedicated self-configurable management systems using the application heartbeats approach [10]

II. METHODOLOGY AND IMPLEMENTATION DETAILS

In this section we explain our proposed emulation method to enable DVFS functionality in gem5. We start with a quick review of the simulation flow in gem5. We continue with description of the implementation details of the DVFS emulation approach. In our work, the configuration script of gem5 has the exclusive control of the DVFS mechanism. We explain the functions implemented in the configuration script in detail. Also we created new instructions to support the DVFS emulation flow. We explain how the new instructions were created.
DVFS policy algorithms manage the voltage and frequency scaling mechanisms for the efficient use of energy. A sample implementation of a DVFS policy algorithm is provided at the end of the paper.

A. Full-System Simulation Flow in gem5

The gem5 simulator provides full-system capability and supports booting OS on several ISAs. The simulation kernel in gem5 is in C++. The C++ classes are wrapped by Python classes. The Python object classes correspond to their peer C++ simulation object classes (SimObjects).

A collection of SimObjects builds the system for simulation. The configuration script instantiates the SimObjects and defines their relationships using the Python classes. Finally, gem5 loads the application benchmark, executes the configuration script and produces the simulation results (Figure 1).

The interaction between the configuration script and the simulation kernel (or in general between the user level and kernel level of gem5) is made possible using special instructions. The special instructions can be issued in full system (FS) mode where a complete system including the operating system is simulated. The issued instructions can trigger simulation-specific functionalities in gem5.

B. Proposed DVFS Emulation Mechanism

The gem5 simulator focuses on performance analysis. The simulation results of gem5 can be parsed and analyzed to obtain the performance values needed for estimating the power consumption using power models. To estimate the power consumption, the voltage values are also essential.

To emulate DVFS, only the performance aspects of DVFS need to be modeled. The key parameters to consider are the modification of the frequency of the processing cores, and the time required to switch from one frequency to another. All the contention, synchronization, and out-of-order execution issues affected by the frequency scaling are modeled by gem5. The execution statistics of gem5 and the voltage profile employed by DVFS manager can be used to obtain the energy estimates.

In our approach the configuration script controls DVFS based on the instructions provided by the simulation kernel. The configuration script reconfigures the system to scale frequency. For this purpose, we define several sets of CPUs with different clock frequencies. According to the DVFS policy algorithm, the configuration script directs the simulator to switch between the two sets at run time.

In this work, we extend the Alpha processor ISA and create new instructions needed for the proposed DVFS emulation flow. In the configuration side, we define a Python class with its corresponding methods to handle DVFS. The configuration script applies its DVFS policy to the system at run time based on the performance requirements information. Figure 2 summarizes our approach. As we do not modify the OS kernel, the OS has a transparent role in our approach. In the following sections, these topics are explained in detail.

1) DVFS Instructions: In our simulations, we use DEC Alpha processor model. We extend the Alpha ISA by creating new instructions to support our DVFS emulation flow. To create new instructions we use the reserved opcodes of the Alpha core. The new opcodes issue special instructions to trigger the DVFS functionality in the simulator; The opcodes form instructions to hand over the performance requirements of the processor(s) to the configuration script. The DVFS instructions bridge the performance requirements information and the configuration script. Based on the delivered information, the configuration script scales the frequency to save energy.

The main new instruction is called m5_dvfs. To create the m5_dvfs instruction, we add a new function prototype and the functional implementation of the instruction to the Alpha ISA pseudo-instruction codes. In addition to that, we add the new function prototype to the gem5 utilities provided for the Alpha ISA. The gem5 utilities are incorporated in compilation of the applications which call the function(s) defined in the gem5 utilities.

The m5_dvfs instruction triggers the configuration script to react accordingly and increase or decrease the operating frequency at run time.

2) DVFS Class: In the configuration side, we implement a DVFS class. The class DVFS provides functionality to scale frequency up/down depending on the performance requirements of the processors. The class DVFS contains methods to handle the DVFS instructions. The configuration script exploits the DVFS class methods to control the frequency scaling mechanism.

The main method in the DVFS class is py_dvfs. The core component of the method py_dvfs is another method called py_change_freq. The method py_dvfs of the class DVFS exploits the method py_change_freq to scale frequency at run time due to the DVFS policy algorithm. The method py_dvfs
Fig. 2 gem5 with proposed DVFS mechanism. OS has a transparent role.

calls `py_change_freq` whenever the value of the operating frequency exceeds/recedes a defined range.

To emulate DVFS, we define two sets of CPUs with different clock frequencies. The configuration script directs the simulator to switch between the two sets using the method `py_change_freq`. The method `py_change_freq` exploits the `switchCpus` method of gem5 to switch from a set of CPUs to another set with a different clock frequency. The challenging point is that the `switchCpus` method allows Python to switch to a different CPU model.

The gem5 simulator supports several CPU models including AtomicSimpleCPU and O3CPU; The AtomicSimpleCPU is an in-order model with atomic access to the memory. The O3CPU - in contrast - is an out-of-order, timing-accurate and detailed CPU model.

As we need to keep simulation running with the same CPU model (likely the detailed model for power/performance optimizations), we need to handle temporary states in course of the switching process. This is achieved by a transient short simulation after switching from a detailed to atomic model. The shortest simulation time supported in gem5 is 1 tick that is equivalent to 1 ps. We set the transient simulation time to only 1 tick. Larger transient simulation time is possible to model the overhead of switching one frequency to the other.

The `py_change_freq` starts with a simulation of detailed core model. We emulate changing the operating frequency by switching to another detailed CPU model with a different frequency. To this end, the `py_change_freq` sets up a list of cores in pairs of (detailed, atomic) and runs a transient short simulation just after `switchCpus` switches the cores from the current detailed model to atomic. The simulation in this step runs with the atomic model only for 1 tick. The `py_change_freq` method sets up another list of cores at this point - this time in pairs of (atomic, detailed) - having new detailed cores with a different frequency. After this step, the simulation continues with CPU models with a new clock frequency. Figure 3 shows this process.

The proposed DVFS emulation mechanism is based on reconfiguration of the system under simulation at run time. The run-time reconfiguration of the system is due to a drain-resume process that occurs at the switching time. Method `switchCpus` of gem5 drains and resumes the simulation. First, it drains the simulation and switches out the old CPUs. It resumes the simulation just after the new CPUs with new clock frequency take over the job from the old ones.

The simplest DVFS policy algorithm is to call the `py_change_freq` method each time the configuration script is triggered with the `m5_dvfs` instruction. Flexibility of the proposed method makes it possible to easily modify and implement desired policy algorithms.

III. EVALUATION

To validate our approach, we investigate both functionality and flexibility of the proposed mechanism. As explained in section II-B-2, the `switchCpus` method of gem5 is employed to switch between two sets of CPU models and consequently, some transient states need to be handled to afford the job. Investigation is needed to check if this methodology introduces any malfunctions to the performance metrics of a target architecture.

The other parameter to investigate is the flexibility and extensibility of the proposed approach. To this end, we implement the basic structure of a hardware-based self-aware system. Then the DVFS manager scales the operating frequency by applying a simple DVFS policy algorithm. Parts A and B of this section correspond to the assessment of functionality and flexibility of the proposed DVFS emulation method, respectively.

A. Functionality

To assess the functionality of the proposed method, we analyze the execution statistics file of a multicore system that exploits the `py_change_freq` method of the DVFS class in the configuration script. The goal is to analyze the effect of the switching process - from a set of CPUs to another set with a different clock frequency - on the key parameters of the processor.

The experimental system configuration is the default gem5 configuration for DEC Alpha processor. We run the simulation for 4 out-of-order processing cores with L1 and shared L2 caches, all with the default values configured in the native gem5. We consider the same clock frequency for the processing cores.

```
00 simulate detailed model with freq_1
01 cpu_list_1 = [(detailed_1, atomic)] for n cores
02 switchCpus(cpu_list_1)
03 simulate atomic model for 1 tick
04 cpu_list_2 = [(atomic, detailed_2)] for n cores
05 switchCpus(cpu_list_2)
06 simulate detailed model with freq_2
```

Fig. 3 Pseudo-code for the `py_change_freq` method
As application workloads, we use some of the parallel applications of the PARSEC benchmark suite [11] with simmedium input sets.

The validation scenario consists of a batch of three simulations of the same experimental system. In the first two simulations, we set a fixed frequency of the processing cores and dump the performance results at two time instants. The time of the first dump is set inversely proportional to the clock frequency and dump the performance results at two time instants. The second dump is used as the baseline for the subsequent comparison.

In the third simulation, we change the frequency (and the clock frequency) per cycle) per core. As expected, the values of the first phase of the simulation 1 and 3 are equal, and the values of the second phase of the simulation 2 and 3 are very similar. The resulted value of the execution time in the simulation 3 is very close to the expected execution time for the simulation 3. The resultant value of the execution time in the simulation 3 is very close to the expected value (0.71073). The experimental results fit the estimations and confirm that no malfunction is introduced by the py_change_freq method. Similar results were observed for the experiments with 1.2 and 1.8 GHz.

To analyze the memory hierarchy more precisely, Table II reports the miss rate of L1 and L2 and the IPC (instructions per cycle) per core. As expected, the values of the first phase of the simulation 1 and 3 are equal and the values of the second phase of the simulation 2 and 3 are very similar. The analysis of the results shows very slight variations of IPC and the cache miss rate parameters.

The simulation results for the same parameters for three different PARSEC applications with simmedium input sets.

As application workloads, we use some of the parallel applications of the PARSEC benchmark suite [11] with simmedium input sets.

The validation scenario consists of a batch of three simulations of the same experimental system. In the first two simulations, we set a fixed frequency of the processing cores and dump the performance results at two time instants. The time of the first dump is set inversely proportional to the clock frequency so that the number of instructions executed by each system is (approximately) the same. The second dump is performed at the end of the Region-of-Interest (ROI) of the parallel application. These two simulations establish the baseline for the subsequent comparison.

In the third simulation, we change the frequency (and dump the performance results) at the time point that we used for the first simulation. We use the method explained in section II-B-2 to change the frequency in the third run.

We repeat the experiments in three different frequency ranges and analyze the results. Table I reports the number of committed instructions and execution time for the two phases of the three experiments for the case of scaling from 1 GHz to 1.5 GHz. In this experiment, the simulation 3 exercises the set of 1 GHz processors during the first phase and the set of 1.5 GHz processors during the second phase. In the second phase of the simulation 3 of the first CPU set (1 GHz), there is a single committed instruction because of the switching process explained in section II-B-2. Taking the execution time of the first two simulations as a reference, we can estimate the expected execution time for the simulation 3. The resulted value of the execution time in the simulation 3 is very close to the expected value (0.71073). The experimental results fit the estimations and confirm that no malfunction is introduced by the py_change_freq method. Similar results were observed for the experiments with 1.2 and 1.8 GHz.

To analyze the memory hierarchy more precisely, Table II reports the miss rate of L1 and L2 and the IPC (instructions per cycle) per core. As expected, the values of the first phase of the simulation 1 and 3 are equal, and the values of the second phase of the simulation 2 and 3 are very similar. The analysis of the results shows very slight variations of IPC and the cache miss rate parameters.

The simulation results for the same parameters for three different PARSEC applications with simmedium input sets.

### Table I. Number of committed instructions and execution time for the three simulations of the validation test. The results correspond to the scaling from 1 GHz to 1.5 GHz and the blackscholes application with simmedium input sets.

<table>
<thead>
<tr>
<th>Freq. (GHz)</th>
<th>CPU set 1 (1 GHz)</th>
<th>CPU set 2 (1.5 GHz)</th>
<th>Execution (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim. 1</td>
<td>1.0</td>
<td>1.0</td>
<td>0.150000</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.911651</td>
<td></td>
</tr>
<tr>
<td>Sim. 2</td>
<td>1.5</td>
<td>1.5</td>
<td>0.100000</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>0.727724</td>
<td></td>
</tr>
<tr>
<td>Sim. 3</td>
<td>1.0</td>
<td>1.0</td>
<td>0.710761</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>0.704380</td>
<td></td>
</tr>
</tbody>
</table>

### Table II. Total cache misses and IPC for the three simulations of the validation test. The results correspond to the scaling from 1 GHz to 1.5 GHz and the blackscholes application with simmedium input sets.

<table>
<thead>
<tr>
<th>Freq. (GHz)</th>
<th>Mean L1 miss rate</th>
<th>Shared L2 miss rate</th>
<th>IPC per core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim. 1</td>
<td>0.00797019</td>
<td>0.771593</td>
<td>0.772042</td>
</tr>
<tr>
<td>Sim. 2</td>
<td>0.00787051</td>
<td>0.772084</td>
<td>0.772488</td>
</tr>
<tr>
<td>Sim. 3</td>
<td>0.00792737</td>
<td>0.771593</td>
<td>0.772042</td>
</tr>
</tbody>
</table>

### Table III. Total cache misses and IPC corresponding to last phase of simulations 2 and 3 of the validation test. The results correspond to the case of scaling from 1 GHz to 1.5 GHz and three different PARSEC applications with simmedium input sets.

<table>
<thead>
<tr>
<th>Mean L1 miss rate</th>
<th>Shared L2 miss rate</th>
<th>IPC per core</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Sim. 2</td>
<td>0.660100</td>
</tr>
<tr>
<td></td>
<td>Sim. 3</td>
<td>0.660100</td>
</tr>
<tr>
<td>canneal</td>
<td>Sim. 2</td>
<td>0.660100</td>
</tr>
<tr>
<td></td>
<td>Sim. 3</td>
<td>0.660100</td>
</tr>
<tr>
<td>swaptions</td>
<td>Sim. 2</td>
<td>0.660100</td>
</tr>
<tr>
<td></td>
<td>Sim. 3</td>
<td>0.660100</td>
</tr>
</tbody>
</table>
section. We aim to demonstrate the flexibility and extensibility of the proposed DVFS emulation approach in this step.

To demonstrate the flexibility of the DVFS emulation mechanism, we use a DVFS policy algorithm in the configuration script. The configuration script scales the frequency at run-time based on the status of a test application. We use an adapted form of the application heartbeats approach [10] to inform the configuration script about the application status. The heartbeats of the test application determines the right time and direction of scaling of the frequency.

DVFS API is an adapted version of the application heartbeat API to cope with our experiments. We re-implemented some of the API functions of the application heartbeats approach to provide the functionality needed in our case study. In addition to \texttt{m5\_dvfs}, we created two extra instructions to support DVFS. A list of the new instructions is presented in Table IV. The \texttt{m5\_initdvfs} instruction passes initial values such as maximum and minimum threshold values to the simulator. The instruction \texttt{m5\_dvfsstop} stops the DVFS process as the name suggests.

The DVFS API provides functions to send the application heartbeats to the configuration script via the simulator. Using the DVFS API, the application communicates with the configuration script via the simulation kernel. The simulation kernel passes the provided information to the configuration script. In the primary implementation, we implemented three functions for the DVFS API (Table IV). The DVFS API functions deal with the DVFS instructions we developed in the simulation kernel in the same way explained in the ISA extension part in section II-B-1. The application status information originates from the application in form of the heartbeats and is sent to the configuration script via the DVFS instructions defined and added to the simulation kernel.

We added peer methods to the class DVFS (section II-B-2) to interact with the DVFS API functions in the application side. The DVFS API functions are summarized in Table IV and the DVFS class methods are listed in Table V.

The test application, uses the DVFS API to initialize the mechanism with maximum and minimum threshold values of the frequency. The configuration script uses the provided maximum and minimum threshold values to set the scaling range of the frequency for applying the DVFS policy algorithm. The application sends the initial values by the \texttt{m5\_initdvfs} instruction used in the \texttt{hb\_initdvfs} API function. Then it continuously sends the \texttt{m5\_dvfs} instructions as the heartbeats of the test application and eventually the stop instruction (\texttt{m5\_stopdvfs}).

The configuration script receives the DVFS instructions as the heartbeats of the test application and reacts accordingly by means of its peer methods. The method \texttt{py\_initdvfs} of the DVFS class sets the initial values in the configuration script. After initialization, the configuration script calculates the interval of the last received \texttt{m5\_dvfs} heartbeat. It scales the frequency accordingly at run time, if the calculated interval of the last heartbeat is not in the range defined at the initialization time. Finally, the method \texttt{py\_stopdvfs} finishes the work.

The test application sends heartbeats containing the DVFS instructions to the configuration script. As the test application is sequential, the generated heartbeat pattern is monotonic. To show the flexibility of the proposed DVFS infrastructure, we run some parallel applications together with the test application simultaneously. Although several programs run simultaneously in this step, the only source of the DVFS heartbeats is the non-parallel test application. The parallel applications do not contain DVFS heartbeat API but they affect the heartbeat interval pattern of the test application. The outcome is a pattern of heartbeats with various intervals. Figure 6(a) shows this effect.

We apply the sample DVFS policy algorithm presented in Figure 5 to this architecture; several applications run on a multiprocessor system. One of the applications emits heartbeats. The DVFS configuration script receives the DVFS heartbeats in various intervals. According to the sample DVFS policy algorithm, the configuration script checks if the calculated interval of the received heartbeats is in the defined range. In case the heartbeat interval exceeds (or recedes) an specific threshold value, the frequency is scaled up/down at run-time. Figure 6(b) depicts the scaled heartbeat pattern. The frequency value has been scaled according to the sample DVFS policy algorithm presented in Figure 4 and 5.

It is necessary to mention that the flexibility of the

**TABLE IV. DVFS API functions provided for the test application**

<table>
<thead>
<tr>
<th>DVFS API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hb_initdvfs</td>
<td>uses \texttt{m5_initdvfs} instruction</td>
</tr>
<tr>
<td>hb_dvfs</td>
<td>uses \texttt{m5_dvfs} instruction</td>
</tr>
<tr>
<td>hb_stopdvfs</td>
<td>uses \texttt{m5_stopdvfs} instruction</td>
</tr>
</tbody>
</table>

**TABLE V. Methods of the class DVFS in the configuration script**

<table>
<thead>
<tr>
<th>DVFS Class Method</th>
<th>Peer DVFS API Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>py_change_freq</td>
<td>-</td>
</tr>
<tr>
<td>py_initdvfs</td>
<td>hb_initdvfs</td>
</tr>
<tr>
<td>py_dvfs</td>
<td>hb_dvfs</td>
</tr>
<tr>
<td>py_stopdvfs</td>
<td>hb_stopdvfs</td>
</tr>
</tbody>
</table>
proposed method makes it possible to easily modify and implement desired policy algorithms. For instance, the controller function of the configuration script, may calculate the average value of the intervals of the received heartbeats in a window instead of only considering the interval of the last heartbeat. The light-weight user-level configuration script with its exclusive control on the DVFS emulation is easily modifiable to fit the performance requirements of the target architecture and apply various energy saving policies in a flexible way.

IV. RELATED WORK

The system-level power optimization efforts for multiprocessor architectures can be categorized into two broad directories; architectural and functional approaches. While in the architectural power optimization techniques, the focus is on the ease of supplying instruction and data to the components, the functional approaches rely on tuning the functional metrics of the energy, mainly voltage and frequency. DVFS belongs to the second group of the optimization techniques.

With growing design complexity of multiprocessor systems, architectural simulation for power optimization has become crucial. The architecture simulators support both architectural and functional optimizations for power and performance. Among several architecture simulators, gem5 provides full-system simulation capability and supports detailed CPU models with different ISAs. Since the native gem5 does not support DVFS, researchers combine the gem5 performance simulation results with power simulation tools in offline mode. The off-line combination of gem5 + McPAT [9] is a common example. To the best of our knowledge there is only one published work related to the process of enabling DVFS in gem5. The work presented in [8] extends gem5 for DVFS support. It requires a sophisticated extension of the gem5 infrastructure in kernel level. In addition to that, it focuses on the DVFS approaches using OS kernel frameworks. It adds a kernel-level DVFS management unit to gem5 and provides corresponding OS kernel drivers.

The proposed approach in this paper is different than [8] in that it provides a user-level solution; we emulate the DVFS management via the configuration script in the user side of gem5. The configuration script has exclusive control on the DVFS mechanism. This provides a flexible control over the DVFS policies applied to the target architecture without need to re-compilation of the gem5 source code. Moreover, it does not modify the OS kernel. The whole DVFS mechanism is managed in the user level by means of the configuration script based on the performance requirements of the processing cores.

V. CONCLUSION AND FURTHER WORK

This work presents a lightweight yet powerful approach to incorporate DVFS management in a full system simulator, gem5. Our non-intrusive DVFS emulation approach uses new gem5 pseudo instructions to allow an easy interaction between the emulated platform and the simulation engine. The DVFS management runs in the host processor using Python as the programming language. It is OS agnostic and does not rely on a particular OS kernel. The main features and advantages of the proposed DVFS approach are as follows;

- Non-intrusive: it does not need additional tasks for implementing the DVFS policies in the emulated cores. Thus, it is suitable to analyze hardware-dedicated management solutions.
- OS-independent: since it does not incorporate any OS kernel utilities and the OS has a transparent role in the approach, it is applicable to any full system using any OS even if no CPU frequency scaling infrastructure (such as CPUfreq in Linux) exists.
- Easily-modifiable and extensible: since the core component of DVFS management is implemented in Python (and in the host processor), it is very straightforward to modify and extend the DVFS policies. Moreover, the infrastructure can be easily used for further purposes.

We presented experimental results to demonstrate the functionality of the approach. As expected, the interaction between the processor, caches, and interconnect architecture cannot be modeled by straightforward performance scaling. Our experimental results show very small variations of the IPC and cache miss rates in the simple target model (Table II and III). Further on, we evaluated the flexibility and extensibility
of the approach implementing a simple hardware-dedicated self-aware multiprocessor system using an adapted version of the application heartbeats API. With a very compact Python model we were able to emulate a self-aware controller. It can be easily modified and extended to analyze the trade-off between hardware complexity and control optimality in self-aware systems.

As the further work, we plan to employ the DVFS infrastructure to characterize the performance profile of an in-house developed multiprocessor system.

REFERENCES


