Introducing DVFS-Management in a Full-System Simulator

Vasileios Spiliopoulos*, Akash Bagdia†, Andreas Hansson†, Peter Aldworth†, Stefanos Kaxiras*
*Uppsala University, Department of Information Technology
{vasileios.spiliopoulos, stefanos.kaxiras}@it.uu.se
†ARM, Cambridge, United Kingdom
{akash.bagdia, andreas.hansson, peter.aldworth}@arm.com

***Abstract*—Dynamic Voltage and Frequency Scaling (DVFS) is an essential part of controlling the power consumption of any computer system, ranging from mobile phones to servers. DVFS efficiency relies on hardware-software co-optimization, thus using existing hardware cannot reveal the full optimization potential beyond the current implementation's characteristics.**

To explore the vast design space for DVFS efficiency, that straddles software and hardware, a simulation infrastructure must provide features that are not readily available today, for example: software controllable clock and voltage domains, support for the OS and the frequency scaling module of it, and an online power estimation methodology. As the main contribution, this work enables DVFS studies in a full-system simulator. We extend the gem5 simulator to support full-system DVFS modeling. By doing so, we enable energy-efficiency experiments to be performed in gem5 and we showcase such studies. Finally, we show that both existing and novel frequency governors for Linux and Android can be effortlessly integrated in the framework, and we evaluate the efficiency of different DVFS schemes.

I. Introduction

Dynamic Voltage and Frequency Scaling (DVFS) is one of the most important techniques for managing the dynamic power consumption of a system. From mobile devices to large data centers, scaling frequency down when it is not critical for performance also allows voltage to be scaled down. Due to the dependence between power consumption and voltage-frequency[22], this technique can provide significant energy savings, with limited or no performance loss. DVFS policies have focused on detecting idleness in the system [28, 30, 29, 19] to scale frequency down and maximize energy savings at a minimum performance cost. Even though DVFS is implemented in most modern processors, a significant amount of DVFS-related work has been taking place in simulators [23, 27, 24].

Simulation frameworks have always been a powerful tool in the hands of computer architects. Incorporating detailed performance models in simulators and using them to approximate the behavior of real hardware enables the computer designers to experiment with a variety of different configurations early in the design stage of a system and investigate interesting trade-offs before the prototype stage of the design process. Moreover, simulators can be utilized to gain insight about how real hardware works with respect to application behavior; executing applications in a controlled simulation environment allows the computer architect to investigate and better understand complex interplays between different aspects of the system, such as the memory hierarchy organization, pipeline design, interconnection between multiple cores in a multi-core system, etc. The freedom to monitor all these aspects gives the designer a better insight and can prove an extremely useful tool towards developing abstract models of computer systems.

Although a variety of different performance simulators exist, not as much effort has been spent towards enhancing these simulators with power modeling and power control features. Wattch [7] was one of the first frameworks to deliver power-related information, and was successfully used as an extension in several performance simulation frameworks (SimpleScalar, etc.) More recently, McPAT [25] provided a unified power, area and timing estimation framework, and quickly developed as the most popular power extension for performance simulators. Several researchers[9, 32, 21] have combined the gem5 simulator[5] with McPAT to get energy consumption estimation of a system. The way this typically works is to simulate a system, collect all the statistics required by McPAT and then feed them to the power models offline to get the final estimation. Although this approach works, it suffers from serious limitations:

- McPAT has no DVFS support, in the sense that voltage and frequency are design parameters that cannot be changed at runtime (changing frequency invokes the power optimizer to come up with a new system design)
- McPAT and gem5 interfaces are not always on par with each other and the assumptions that have to be made are hard to verify
- gem5 stats have to be mapped to McPAT stats, which is an error-prone process
- power estimation numbers are hard to feed into the performance simulator execution engine, which is prohibitive for simulator extensions dependent on online power-estimation

The above-mentioned issues introduce extra overhead for the user and narrow down the capability of extending the simulator with features that depend on power monitoring. Examples of such features are the integration of power sensors that are exposed to the software running on the simulator and system thermal management (e.g. throttling processor frequency when temperature gets critically high.) Since performance and power are so tightly coupled with each other, a simulator should

support easy integration of power estimation models.

Apart from estimating power consumption, a simulator should also model various power-saving techniques that are commonly used in computer systems. DVFS, clock gating and power gating are among the most popular techniques used for improving energy efficiency and have been extensively researched in both industry and academia, thus a simulator should be able to address these features from the hardware level till the software-control level.

The goal of this paper is to turn gem5 into a complete hardware-software framework suitable for full-system DVFS studies. To achieve that, apart from the DVFS-control features we also add a power-estimation framework which is required for evaluating the efficiency of various DVFS policies. More specifically, the main contributions of this paper are the following:

- we add the notion of clock and voltage domains to the simulator, as well as the simulation structures that manage voltage and frequency scaling (Section III)
- we extend gem5 with full DVFS support (Section IV.) On the hardware side, we implement a memory mapped DVFS controller which is flexible enough to control any clock/voltage domain topology and provides registers for software interface. On the software side, we implement the low-level Linux drivers that manage the DVFS controller in a way that existing higher level kernel modules (frequency governors) work without any modification
- we extend gem5 by adding a framework that allows easy power-model integration, and we tweak McPAT to provide the power coefficients required by gem5 (Section V)
- we show how these extensions can be used to perform full-system power efficiency studies (Section VI)

II. GEM5 BACKGROUND

The gem5 simulator is the outcome of merging two very popular tools, the M5 [6] and the GEMS [26] simulators. The M5 simulator provided features that many CPU-centric simulators lacked, such as full-system (including OS) simulation and I/O support. GEMS, on the other hand, featured more detailed memory models, and it was widely used in academia for memory-hierarchy related research. gem5 combines the pipeline models and full-system support of M5 with the memory model of GEMS, delivering a powerful tool that can cover a wide range of research topics.

We use gem5 because it already supports full-system simulation, which is crucial for enabling DVFS. gem5 is capable of booting Linux and Android, which already feature modules for DVFS management, thus we do not need to develop the whole software stack from scratch. Moreover, I/O devices are fully supported in gem5, thus we can use existing simulation structures to implement a memory-mapped DVFS controller.

gem5 is written in a combination of C++ and python. Most of the execution engine of the simulator is written in C++, using a flexible class hierarchy scheme. Base classes provide the main functionality aspects of commonly used components, such as CPUs, caches, buses etc. Classes derived from the base classes are used to define more specialized units, such as an out-of-order CPU, interconnect components and instruction/data caches. For C++ classes associated with major components, such as CPUs or buses, there are python classes that are used for configuring and connecting components with each other. Thus, on the user side, setting up and simulating a system in gem5 is nothing more that writing a python script that instantiates and connects various components with each other. This way, the topology of the system is not hard-coded in the simulator, and a variety of different systems can be created.

Listing 1 shows a simplified example of how a gem5 configuration script is built. In this example we skip many of the details and focus on the idea of how a system can be created and simulated in gem5. More complex configuration scripts, however, can be used to model systems ranging from smartphones/tablets to desktops and servers. Every object can be created in the python script by calling the class constructor and setting the parameters. This way, a CPU and instruction/data caches are added to the system. The LinuxSystem class sets up the whole simulation environment, including the hardware, kernel and disk images. In gem5, every object needs to be the child of another object, forming thus the simulation object-hierarchy. One object can have multiple children, but every object has only one parent object. In our example, the root object is the root of the hierarchy, and my_system is its child. my_system is the parent of the my_cpu object, which, in turn, has two children, the icache and dcache objects which are both of the type BaseCache. After the system has been configured, two special gem5 methods are used to i) instantiate all the objects in the hierarchy in the C++ part of the simulator and ii) start the simulation.

Listing 1. Default gem5 configuration script

```
from m5.objects import *

my_cpu=SimpleCPU(clock='2GHz', width=2)
my_cpu.icache=BaseCache(size='32KB', assoc=2)
my_cpu.dcache=BaseCache(size='64KB', assoc=2)
my_system=LinuxSystem(cpu=my_cpu)
root=Root(system=my_system)

m5.instantiate(root)
m5.simulate()
```

III. INTRODUCING CLOCK AND VOLTAGE DOMAINS

In this section we discuss what is the landscape of clock and voltage domains in real systems, how gem5 deals with them, and how we extend gem5 to realistically model clocks and voltages in the system.

A. Clock/Voltage Domains Landscape

Real systems comprise of different clock and voltage islands. Typically, a CPU core operates under a single clock and voltage domain, but several alternatives have been proposed for a multi-core chip: different cores can operate under the same (Intel) or different (AMD) clock domains, and similarly cores can be either on a single or multiple voltage domains. Apart from the CPU cores, the rest of the system in an SoC

(buses, shared caches, devices etc.) usually has different voltage-frequency requirements and thus separate domains are specified for them. Hence, it is clear that a simulator should support the flexibility for a variety of topologies to be expressed, allowing the exploration of a vast design space regarding DVFS for individual clock/voltage domains in the system.

B. Clocks in gem5

In gem5 (as of March 2013), every component (object) that has the notion of a clock is derived from the ClockedObject class. Every such object has a clock parameter, which is the clock frequency the component operates at. By default, objects inherit the clock of their parent, thus the user only needs to specify the clocks for high-level components. For example, since a cache is part of the CPU, and thus a child of the CPU object, it will operate under the same frequency, unless specified otherwise in the configuration script.

This infrastructure, although convenient for configuring statically the clock frequencies of individual components, is not appropriate for managing frequencies dynamically; in the existing gem5 framework, each object has a clock running independently from any other clock in the system. There is a connection between the clock values of objects linked with a parent-child relationship, but only in the sense that the initial values of these objects will be the same. Changing the clock period of the parent during the simulation will not affect the child's clock period. In other words, changing the clock frequency of a CPU object will not affect the cache's clock frequency, resulting in the CPU and the cache operating under different frequencies which is something undesirable.

C. Clock Domain Extension

Typically, we want to define simulation entities that have a single clock source. These entities may comprise of multiple clocked objects, thus whenever the clock frequency is scaled up/down, this change has to be communicated to each of these objects. Moreover, the way of declaring the top-level entities, i.e., the ones that correspond to the components that are clocked individually from any other component in the system, might change and should be configurable by the user; e.g., an Intel multi-core chip has all of the cores operating under the same clock frequency, AMD multi-cores, however, provide different clock signals to each of the cores in a multi-core[11], allowing each core to run at a different frequency. Thus, the introduction of a common clock source should not be hard-coded in a particular level of the class hierarchy, but should be kept generic enough for a variety of topologies to be supported.

We introduce a new class type to gem5, called ClockDomain. This class defines the clock period that, all objects belonging to that domain, will operate under. The ClockedObject class is modified so that, instead of declaring the clock period of the object, it now declares the clock domain the object belongs to. This is on par with the way real systems are implemented: a clock source is created in the system and the clock is fed as an input to each of the components that are supposed to operate under this clock.

Introducing this new class type makes handling the clocks of the system more flexible and convenient. Every clocked object, in order to simulate the timings at which specific events will occur, gets the clock period directly from the ClockDomain object it belongs to. Thus, changing the frequency of the clock domain immediately reflects on all of the objects under that domain. Objects linked with a parent-child relationship, belong by default to the same clock domain (the child inherits the clock domain parameter from its parent.) In our CPU example, this means that every child of the CPU object (caches, TLBs etc.) belongs to the same clock domain, unless the configuration script specifies different domains for some of them.

We provide two specialized classes that can be used for declaring *Source Clock Domains* and *Derived Clock Domains*. A *Source Clock Domain* is a clock domain that has a clock period independent of any other domain. On a real system[13], this corresponds to a tunable PLL which provides the clock signal to the components that are under this clock domain. On the other hand, *Derived Clock Domains* are clock domains that always run on some fraction of the frequency of another clock domain, and these are usually associated with sub-blocks of the design that do not need to operate on high frequency. Hence, the clock period of a *Derived Clock Domain* is determined using some hardware divider logic, and changes on par with the *Source Clock Domain* it is associated with.

D. Voltage Domain Extension

Similarly to grouping individual objects into clock domains, we also group them in voltage domains, i.e., objects that always operate under the same voltage. For this classification we make the following assumptions:

- objects that belong to the same clock domain always belong to the same voltage domain
- objects that belong to different clock domains can belong to a single or multiple voltage domains.

These two assumptions are compatible with the way clock and voltage domains are specified in a real system. Components that are clocked under the same clock source, have the same voltage requirements and thus should belong to the same voltage domain. On the other hand, if two components have individual clock sources, they can still get powered by the same voltage source as long as the voltage is tuned so as to satisfy the component with the highest voltage requirement. This is because, for a given clock frequency, a minimum voltage is required for the component to operate properly, so any voltage greater than this minimum is also valid.

Regarding gem5, we create a VoltageDomain class and we assign every clock domain object to some voltage domain object in the configuration script. This way we ensure that every component in a single clock domain belongs to the same voltage domain. Moreover, since voltage is essentially an object and not just a parameter in the ClockDomain class, we can assign multiple clock domains to the same voltage domain. Finally, both clock period and voltage of clock/voltage domains

respectively are exported as statistics, and they are a crucial part of the power-estimation framework discussed in Section V.

Listing 2 shows how the example configuration script of Listing 1 can be updated to declare clock and voltage domains and assign components to them. In the example we first create a voltage domain and then a clock domain that is registered to that voltage domain. The CPU is assigned to that clock domain, and all the children of the CPU (icache and dcache objects), inherit the clock domain parameter from their parent, i.e., the my_cpu object. Thus, with the addition of clock and voltage domains, the simulator has the ability to express the notion of clock and voltage sources, providing support for all the connections to be configured in the configuration script.

Listing 2. gem5 configuration script augmented with clock and voltage domains

```
from m5.objects import *

v_dom=VoltageDomain(voltage='1V')
clk_dom=SrcClockDomain(clock='2GHz',
                       voltage_domain=v_dom)

my_cpu=SimpleCPU(clock_domain=clk_dom, width=2)
my_cpu.icache=BaseCache(size='32KB', assoc=2)
my_cpu.dcache=BaseCache(size='64KB', assoc=2)
my_system=LinuxSystem(cpu=my_cpu)
root=Root(system=my_system)

m5.instantiate(root)
m5.simulate()
```

IV. Introducing DVFS in gem5

In this section we explain how we extend gem5 with full DVFS support. In Section III we explained how the notion of clock and voltage domains is introduced to the simulator. This extension provides the flexibility to declare clock/voltage topologies for a system, but there is no functionality for managing clock frequencies and voltages dynamically. In real systems, such a functionality requires both hardware and software support. On the hardware side (Section IV-A), we need to implement a controller that i) manages the frequencies of the various clock domains in the system based on OS requirements, ii) manages the voltages of the voltage domains so that all the voltage requirements for the individual clock domains are satisfied and iii) provides a register interface so that software can issue DVFS commands for any clock domain of interest. On the software side (Sections IV-B, IV-C), we need to develop the drivers that control the modeled hardware and, if possible, integrate them on the current frequency scaling module of the Linux kernel. An important consideration in our design is to keep the implementation generic enough so that any configuration of interest can be applied.

A. DVFS Controller

In a real system, a DVFS controller is a component that is responsible for setting clock frequencies according to OS policies. It provides memory-mapped registers to communicate with software[17], and should be able to handle all the different clock domains available in a system. For every different clock domain, the controller needs to provide a register that can be used by the software to request for different frequency levels. OS can write some integer value in that register, which corresponds to some frequency level (in Linux, lower frequency level values correspond to higher clock frequencies, with 0 mapping to the highest available frequency of the system.) It is also possible to read the same register to retrieve the current frequency level of the domain. Another register can be provided for the same domain for handshaking purposes: the controller can set this register once the transition to the new frequency has been completed, and the register can be reset once it is read by the software.

In our implementation, similarly to a real SoC, we need to design a controller flexible enough to model an arbitrary number of clock domains. Dedicating one pair of registers for each clock domain however is rather restricting. The number of registers of the controller needs to vary depending on the simulated system, which means that the memory-map of the device is not constant across different system configurations. Instead of having a configurable number of registers for the device, we minimize the amount of registers required by multiplexing the clock domains.

Figure 1 shows a high level schematic of our controller along with an example SoC[2]. Depending on the actual SoC, the implementation of the controller differs. The controller shown in Figure 1 is an ad-hoc implementation specific to gem5, but it is generic enough to model any topology of interest. We assume that the example system consists of a Little CPU, a Big CPU and a GPU, each of them in a separate clock/voltage domain. One more clock/voltage domain is used to supply the rest of the system, i.e., interconnect, DRAM and other system components. In this example there is one-to-one mapping between clock and voltage domains, but as explained in Section III multiple clock domains can lie under the same voltage domain. Software (OS) can interact with the controller through 3 registers. The *Domain ID* is a read/write register, used for selecting the clock domain of interest. *Freq level* register is also read/write register and is used for setting or reading the frequency level of the domain that *Domain ID* register points to. Finally, *Ack* is a read-only register used for acknowledging that a DVFS transition has finished. In that case, the controller sets the *Ack* register to 1 and, once software reads the register, it resets it to 0.

A limitation of the current design is that only one outstanding DVFS request can exist at a time. This is in accordance with the way cpufreq drivers are developed: the part of the code that makes the request and waits for the DVFS transition to complete is in a critical section of the driver, thus only one request can exist at a time. However, our controller design can be easily extended to support multiple concurrent DVFS transitions: for each clock domain, there can be an internal pair of registers (*Freq level* and *Ack*) that maintains the state of that domain, and the 3 registers exposed to the memory address of the system would remain the same. This way, we can have multiple DVFS transitions, with one *Ack* register per domain, and the selection of which register is to be exposed on the memory map can be done using the *Domain ID* register.

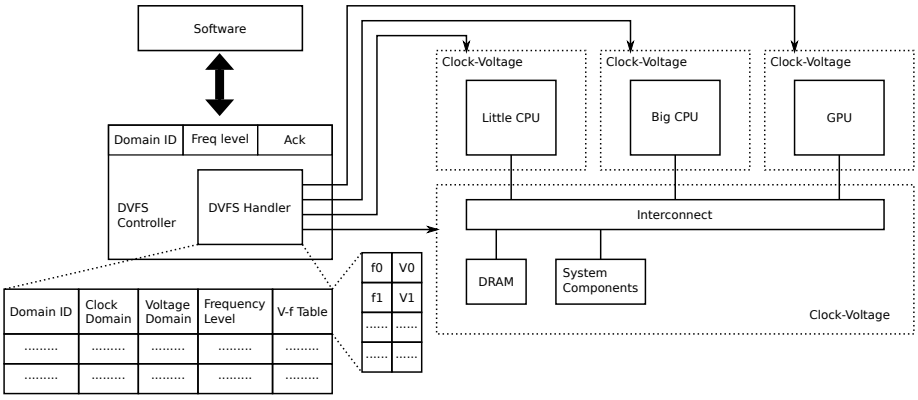

Figure 1. DVFS controller and example system to control. The controller features 3 different registers to control the available clock domains of the system. These registers are used to specify the domain of interest, set/get the performance level and acknowledge that the transition has completed. The controller is aware of the exact topology of the system and the V-f parameters of each of the domains.

The *DVFS Handler* is a simulation object that handles all the clock and voltage domains in the system. Unlike the DVFS controller, which is an actual I/O device, the handler does not have a physical representation in the system; it is a simulation component which is part of the controller and maintains all the required information for each clock domain. Moreover, it provides methods for setting/getting clock frequency and voltage values, and, in the case of multiple clock domains lying under the same voltage domain, it makes sure that the voltage requirements of all of the clock domains are satisfied. For each clock domain in the system, the handler maintains an entry in an array which includes the ID of the domain, pointers to the corresponding ClockDomain and VoltageDomain C++ objects, the current frequency level of the domain and, finally, all the available V-f pairs the domain can operate under.

Listing 3. gem5 configuration script with DVFS controller declaration

```
from m5.objects import *

v_dom=VoltageDomain(voltage='1V')
clk_dom=SrcClockDomain(clock='2GHz',
                       voltage_domain=v_dom)

domains_config = []
clk_dom_config = DomainConfig(
  clk_domain = clk_dom,
  freq_op_points = ['1.7GHz', '1.2GHz,
    '0.7GHz', '0.2GHz'],
  voltage_op_points = ['1.28V', '1.08V',
    '0.96V', '0.93V'],
  domain_id = 0,
  transition_latency = '100us')

domains_config.append(clk_dom_config)
my_dvfs_controller.dvfs_handler =
  DVFSHandler(domains = domains_config)

my_cpu=SimpleCPU(clock_domain=clk_dom, width=2)
my_cpu.icache=BaseCache(size='32KB', assoc=2)
my_cpu.dcache=BaseCache(size='64KB', assoc=2)
my_system=LinuxSystem(cpu=my_cpu,
  dvfs_controller=my_dvfs_controller)
root=Root(system=my_system)

m5.instantiate(root)
m5.simulate()
```

Listing 3 completes the configuration script shown in previous sections with *DVFS Handler* and *DVFS Controller* declarations. For the clock domain we have declared, we specify the configuration parameters (domain ID, DVFS transition latency, and voltage-frequency operating points), pass them to the handler, and assign the handler to the DVFS controller.

B. Linux Cpufreq Driver

With the hardware extensions in place, it is still left to the software to make use of it and control all the clock and voltage domains in the system. Figure 2 shows the software stack for the cpufreq component in Linux. This can be divided into three parts. At the lowest level, the machine-specific driver initializes the power control unit and provides the basic I/O access functions to the device. At the next level, kernel modules use the functions provided by the underlying layer to implement more high-level functionalities, such as detecting the system domain-topology, translating requests for frequency transitions to appropriate frequency levels and call the low-level DVFS routines provided by the lower-level driver. These modules, called cpufreq drivers, are machine-specific and only one of them can be registered in a system at a time. Registering the cpufreq driver is part of the kernel-boot process, and every cpufreq driver should comply with the interface specified by the cpufreq module. As Figure 2 shows, it is only the low-level modules that are gem5 specific. The remaining of the cpufreq module can be used as it is, without any modifications.

In our implementation, the lowest level driver simply reads and writes to the DVFS controller registers. A **set_performance** function writes first the domain ID and then the required performance level to the corresponding registers. Then, it keeps polling the *Ack* register until it gets set, meaning that the DVFS transition has completed. Similarly, for a **get_performance** function, the driver writes the *Domain ID* register and then reads the *Freq level* register to acquire the current frequency of the specified domain.

Listing 4 shows the convention for developing a cpufreq driver. The cpufreq_driver struct consists of a set of function pointers. Once a driver is registered as the system's cpufreq driver, the high-level cpufreq modules call these functions to get

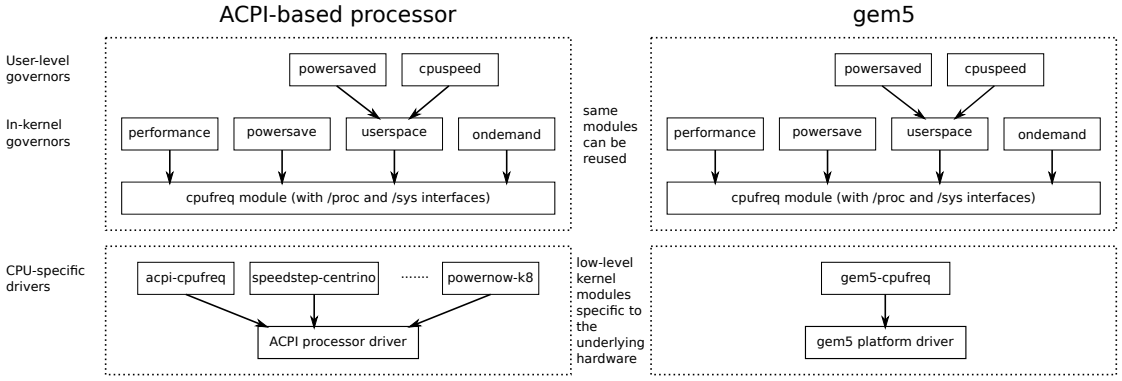

Figure 2. Software stack of the Linux cpufreq module in an ACPI-based[28] system and the modifications required for a gem5-based system. Note that only the low-level modules are gem5-specific, whereas all the high-level kernel and userspace modules can be reused as they are.

the low-level services. Implementing the proper function is the only task that the driver developer needs to do for everything to work seamlessly between the different levels of the software stack. Hence, as Figure 2 shows, we only need to develop proper configurable low-level modules, and all the standard high-level modules can be reused as they are.

Listing 4. Linux convention for a cpufreq driver

```
static struct cpufreq_driver gem5_cpufreq_driver =
{
        .flags  = CPUFREQ_STICKY,
        .verify = gem5_cpufreq_verify_policy,
        .target = gem5_cpufreq_set_target,
        .get    = gem5_cpufreq_get,
        .init   = gem5_cpufreq_init,
        .name   = "cpufreq_gem5",
        .attr   = gem5_cpufreq_attr,
};
```

We refer the interested reader to the Linux kernel documentation[4] for more details about each of these fields. For the **gem5_cpufreq_init** function, it is worth mentioning that the driver parses the device tree (see Section IV-C) to discover the topology of clock domains and then registers and populates the frequency tables for every clock domain in the system. Hence, for different system configurations, we only need to provide a valid device tree instead of modifying the driver source code.

C. Linux Device Tree

The device tree is a structure that contains all the necessary information to setup and initialize all the devices in a system. During the boot process, the device driver parses the corresponding entry of the device tree to obtain information such as the memory map for the device, the number of registers required and various configuration parameters. For our kernel extensions, we need to extend the device tree by adding the frequencies available for each clock domain, and also create a node to register our DVFS controller as a system device.

Listing 5 shows how a device tree can be extended to support DVFS on gem5. First we create a "domains" node, and under that node we declare all the clock domains along with the available frequencies for each domain. Then, we add a device entry for our DVFS controller, and we specify the address range and a compatible driver for our device. To pick the address range, we need to examine the memory map of the system and detect an area of 12 bytes (3 registers, 4 bytes each) not used by other devices. We should mention that currently device tree and configuration script are written separately, so special care has to be taken for the configuration parameters to match each other. A device tree auto-generated from the configuration script is left for future work.

Listing 5. Linux Device Tree extensions

```
domains {
 cpu_clock_domain@0 {
  freqs = <1700000000 1200000000 700000000 200000000>;
 };
 cpu_clock_domain@1 {
  freqs = <1300000000 1100000000 400000000 200000000>;
 };
};

gem5_dvfs_controller@1c080000 {
 compatible = "arm,gem5_dvfs_controller";
 reg = <0x1c080000 0xc>;
};
```

V. Power Model Integration

So far we have discussed the DVFS extensions in gem5 that enable us to provide run-time, software-managed control of clocks and voltages of the different parts of the design. To perform interesting experiments and evaluate the benefits of various DVFS schemes we need two additional features currently lacking in gem5: i) the power models of the various blocks (simulation objects) of interest and ii) a well-defined methodology for run-time power estimation.

For power models we still rely on McPAT, although any power-modeling methodology can be used to provide the input for the power-modeling framework of gem5. In order to seamlessly integrate estimation in the simulator and not to rely on external tools, we designed a power-estimation framework within the simulator's existing framework that enables us to port power models in a gem5-friendly format. To achieve this, we use the existing statistics infrastructure within gem5. Using the statistics of gem5 is indispensable for power estimation, since power models rely on the activity of various components, as well as voltage and frequency of the various domains which are

also reported as statistics (section III.) For our studies we used this infrastructure for estimating total power consumption of the system, however our infrastructure allows more sophisticated studies to be performed, such as power control based on run-time power usage and temperature profile of the design.

The run-time power consumption of a configurable IP block in the design depends on a large set of static and dynamic parameters, such as the implementation technology node, design parameters (size, number of gates, flops, SRAMs etc.), switching activity of the component, voltage-frequency operating points, temperature variations, power state (clock-gated/power-gated) etc. With our proposed methodology we intend to address all this information and provide the necessary infrastructure to ensure that gem5 users can express the above intent with their power model at relative ease.

The power models are specified for each simulation object of interest. Every power model is expressed as a set of i) configuration parameters which are used to determine whether a given object qualifies for an available power model in the database, and ii) power model coefficients that are used to estimate power consumption based on statistics collected dynamically by the simulator. Different power models can co-exist for various configurations, and gem5 picks a proper model (if it exists) based on the simulation configuration script provided by the user.

We assume that the power models are only expressed for objects that are clocked objects which have a notion of clock and voltage associated with the modeled simulation object. This assumption comes because of the fact that power/energy consumption of a component being simulated varies with its operating frequency and voltage. Coefficients in the power/energy models are therefore expressed as clock/voltage variant or invariant. Every gem5 component that has a physical representation in the system, is a clocked object at some point in the hierarchy. For example, a TLB might not be a clocked object, but it is a part of a CPU which is a clocked object. Hence, a power model can be created for every component in the system.

The energy consumption equation can be described with in principle (but not limited to) three different types of coefficients:

- frequency-independent, voltage-dependent Energy Coefficients: such coefficients are used to estimate the event-based energy consumption of the system. Each event, such as a cache access, a cache miss or an ALU access, is associated with a frequency-independent, voltage-dependent coefficient, and the total energy consumed is simply the total number of the occurrences of the event multiplied by the coefficient.
- frequency-dependent, voltage-dependent Power Coefficients: such coefficients are used to estimate the power consumption due to activity that is not associated with some event count, e.g., power consumed in clock tree.
- frequency-independent, voltage-dependent Power Coefficients: this type of coefficient is used to take into account leakage power consumption, which depends on voltage but not on frequency.

We modified McPAT to retrieve a set of coefficients that describes our gem5 modeled system, and we feed these coefficients in our power-estimation framework. For each component, different equations can be defined reflecting the different power state it can be. Thus our power-estimation framework can be used for idle-power management studies. Apart from estimating power consumption in idle state, such studies require hardware and software support. As future work, we plan to extend the simulator with idle-power management support, as well as thermal management.

VI. Using gem5 for Energy-Efficiency Studies

In this section we show how the infrastructure we developed can be used for studying the efficiency of different DVFS schemes. To showcase the usefulness of our framework, we focus on experiments that are impossible or hard to run without the use of a simulator.

The main contribution of this paper is to enable DVFS studies to be performed in gem5. In the remainder of this section, we provide examples of experiments that can be carried out using the extended version of gem5. As future work, we target on more advanced studies that will investigate in more depth the design space of DVFS energy-efficiency (beyond the standard DVFS schemes of this paper) using the proposed framework.

Table I
GEM5 ARM MODEL DEFAULT SYSTEM PARAMETERS

Parameter	Value
CPU type	out-of-order
number of cores	1
dtlb	64 entries
itlb	64 entries
dcache	64KB-2way-64Byte block
icache	32KB-2way-64Byte block
L2 cache	2MB-8way-64Byte block
decode width	3
issue width	8
Reorder buffer	40 entries

Table II
VOLTAGE-FREQUENCY PAIRS FOR OUR EXPERIMENTS (RETRIEVED FROM SAMSUNG EXYNOS 5250)

Frequency(GHz)	Voltage(V)	Frequency(GHz)	Voltage(V)
1.7	1.278	0.9	1.001
1.6	1.228	0.8	0.989
1.5	1.177	0.7	0.963
1.4	1.140	0.6	0.951
1.3	1.115	0.5	0.939
1.2	1.077	0.4	0.926
1.1	1.046	0.3	0.926
1.0	1.027	0.2	0.926

First, we validate that our gem5 extension works as it is supposed to, by comparing the frequency transitions observed in gem5 and a real ARM-based development board when running an adaptive frequency governor. Then, we investigate what is the impact of DVFS transition latency in the system's efficiency, and whether existing frequency governors can take advantage of reduced DVFS latency to adapt faster to the workload

behavior and thus improve efficiency. Finally, we explore how the number of different V-f pairs affects efficiency, and how a computer architect can possibly reduce design complexity without sacrificing efficiency by picking the most important V-f points for specific workloads. For our evaluation, we use the default gem5 configuration for an ARM processor, the most important parameters of which are shown in Table I.

An important parameter for our experiments is the available frequencies the core can run at and the corresponding voltages. We get the available V-f pairs from an existing ARM implementation, the Samsung Exynos 5250 chip[3] featuring a dual-core ARM A15 processor. We get accesses to the V-f pairs of this processor by measuring the core voltage of an Arndale development board[1]. The voltage pin is exposed on the PCB of the development board and it is fairly easy to locate using the board schematics. By using the userspace governor to control the frequency and an oscilloscope to measure the voltage, we come up with the V-f pairs shown in Table II. We run a browser benchmark called BBench [15] on top of Android Ice Cream Sandwich operating system, and we evaluate against existing DVFS Governors (Interactive, Ondemand and Performance governors.) Note that, apart from modifying the decision interval of adaptive governors to make them take advantage of the reduced DVFS transition latency, no further modification was required to run the governors in gem5.

A. Validation

To validate that DVFS in gem5 works as it is supposed to, we run BBench in both gem5 and our Arndale board and profile the frequency migrations driven by the Interactive governor. Figure 3 shows that both the Arndale and gem5 runs follow similar frequency-migration patterns. The reasons for the imperfect alignment are i) the Arndale run is only sparsely sampled (5 samples/sec) so that the frequency-monitoring process does not affect the governor behavior, ii) noise from system processes running simultaneously with BBench and iii) a certain amount of discrepancies is explained by the fact that gem5 does not model precisely the Arndale board. gem5 is a generic simulator that allows high-level parameters (such as cache sizes, number of cores etc.) to be configured, but it cannot capture the details of the processor pipeline design. Therefore, in the figure we can see similar behavior between the two runs (e.g., intervals A, B, C and D), even though the durations of idle/active intervals might differ between the real system and the simulator.

B. Running BBench with different governors

We run BBench with 3 different governors: Performance governor, which sets the frequency always at maximum value, and Ondemand and Interactive governors, which set the frequency based on CPU-utilization. For these experiments we assume that the DVFS latency is 100usec (as in Exynos chip) and we use the voltage-frequency pairs of Table II. Figure 4 shows execution time and energy consumption for BBench normalized on the case of Performance governor. As the figure shows, Performance governor is the one that achieves the

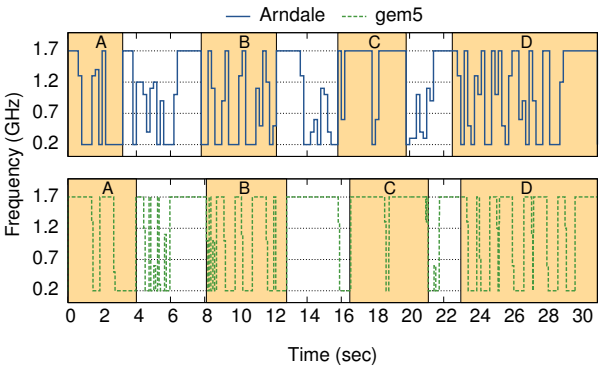

Figure 3. Running BBench with Interactive governor on Arndale board (top) and gem5 (bottom). The two cases, even though not identical, follow similar frequency migration patterns. In the figure, we have highlighted four example intervals that, although shifted in time, behave similarly in terms of frequency migrations.

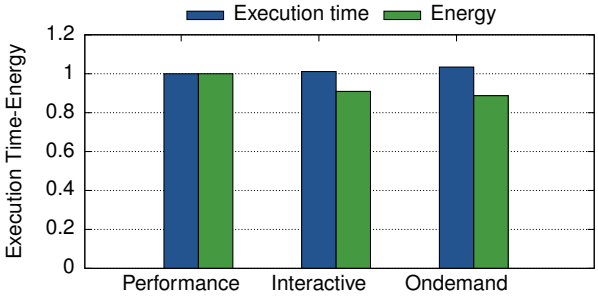

Figure 4. Execution time and energy consumed for Performance, Interactive and Ondemand governors, normalized to the case of Performance governor. Performance governor is slightly faster, but utilization-based governors can save a significant amount of energy by scaling frequency down during idle periods.

lowest execution time, but at the same time consumes the most energy. This is because Performance governor always picks the highest frequency, even during the periods that CPU is idle, hence losing any opportunity to lower power consumption during the idle periods. Interactive and Ondemand, on the other hand, are adaptive governors that scale CPU frequency based on CPU-utilization. By lowering frequency during idle periods they achieve low static power consumption, while by raising frequency during non-idle periods they try to maximize performance, thus both Interactive and Ondemand achieve slightly worse performance but significant energy savings compared to Performance governor. If we compare Interactive and Ondemand, Interactive is faster to adapt to the workload changes and thus achieves better performance at about the same energy consumption.

C. Sensitivity to DVFS Transition Latency

In this section we take the two adaptive governors explored in VI-B and investigate whether a faster DVFS transition latency has an impact on the governor's efficiency. We try 4 different transition latencies: 100usec, 50usec, 10usec and 1usec. For the governor to take advantage of the reduced latency, we scale the interval that the governor takes a decision proportionally to the DVFS transition latency. For example, going from 100us to 50us transition latency, we also reduce the duration of decision interval by a factor of 2. Figure 5 illustrates the results of our

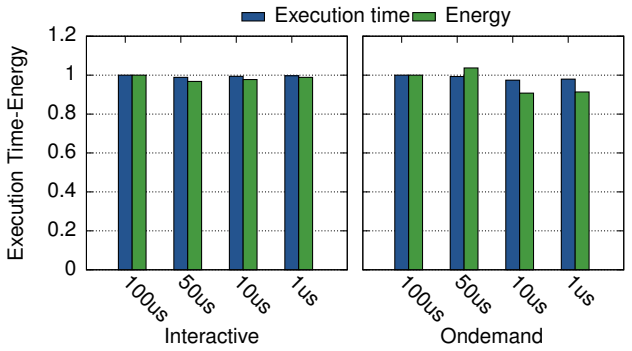

Figure 5. Impact of DVFS transition latency in execution time and energy consumption of Interactive and Ondemand governors running BBench. For each governor, execution time and energy are normalized to the case of 100usec of transition latency.

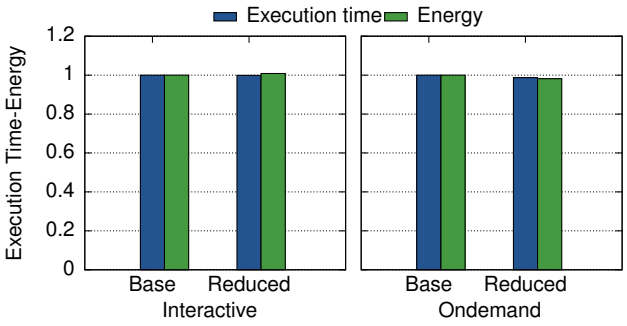

Figure 6. Execution time and energy consumed for Interactive and Ondemand governors, when the Base and Reduced voltage-frequency sets are used. Reduced cases are normalized to the corresponding Base cases. The binary behavior of BBench results in the number of available frequencies only marginally impacting the efficiency of the system.

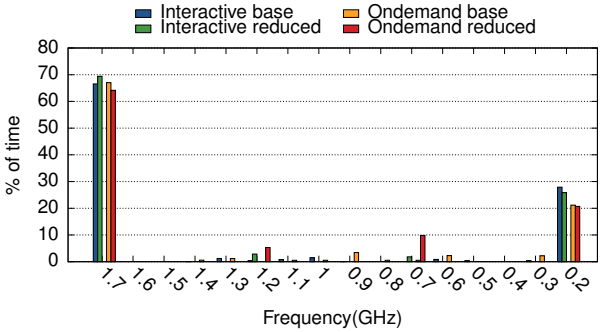

Figure 7. Distribution of execution time in different frequencies for Interactive and Ondemand governors. A high concentration in maximum and minimum frequency indicates that the number of intermediate frequency steps does not highly impact governors' behavior.

experiments for Interactive and Ondemand governors. Both governors can take advantage of the reduced latency to i) scale frequency down faster during idle periods, resulting in better energy savings, and ii) scale frequency up faster when the idle period is over and the CPU is active again executing BBench code, improving this way the execution time.

Since Interactive governor was already quite fast in adapting to program's behavior, Ondemand experiences a larger benefit from the DVFS-latency reduction. At 50usec, however, Ondemand governor shows an increase in energy consumption compared to the base case. This can be explained by the fact that at lower latencies, the governor gets invoked more frequently, more instructions are executed and an energy overhead is introduced. As the DVFS latency gets further reduced, however, the benefits in execution time and energy compensate for the extra instructions that are executed. Interactive governor, on the other hand, does not suffer from similar effects, since it enters a "wait" state during idle periods and is not invoked until the processor wakes up again.

At the lowest DVFS latency point (1usec), we observe only marginal differences with the 10usec case. At this point, the governor invocation frequency has dropped below the Linux scheduling quantum, which is 10msec by default in Android systems, hence the governors will be invoked with a 10msec interval and thus they cannot take advantage of the reduced DVFS latency. To overcome this limitation, there are two alternatives: lowering the scheduling quantum and using high-resolution timers. Even though both of them would allow faster invocation of the DVFS governors, modifying timing aspects of the system should be done very carefully. A parameter that one should take into account is that in real systems, individual cores can be shut down after some period of idleness to save energy. Although this technique comes with great potential for energy savings, the cost of switching the state between active/idle is rather high. Hence, enabling DVFS switching at a pace faster than the default OS schedule interval should be explored in accordance with the overhead it would cause in the switching between "on" and "off" states of the system. Such an investigation requires idle-power management capabilities which are not currently available in the simulator, so we leave this kind of analysis for future work.

D. Sensitivity to number of available V-f pairs

In this section we explore how the amount of V-f pairs impacts execution time and energy of BBench. We compare two cases, the Base case with the V-f pairs shown in Table II, and the Reduced case, using only 200MHz, 700MHz, 1.2GHz and 1.7GHz and the corresponding voltages from Table II.

Figure 6 shows that changing the number of available frequencies has almost no effect in execution time and energy consumed, for both Interactive and Ondemand governors. A close inspection of how much time each governor spends on each frequency (Figure 7) sheds more light on this behavior. BBench has a rather binary behavior, spending most of the time in either maximum or minimum frequency for both Interactive and Ondemand governors. Thus, for this workload, the most important parameter is the range for frequency scaling instead of the number of intermediate steps. By exploring the behavior of a wider set of workloads, and given that this shows similar trends, one could argue that we can reduce the design complexity by reducing the number of V-f pairs without significantly affecting performance and energy efficiency of the system. If, on the other hand, this analysis reveals that there are workloads highly affected by the frequency steps, one can use the extended version of gem5 to decide upon the optimal set of voltage-frequency pairs for the workloads of interest.

VII. Related Work

Here we summarize work related to power-modeling and DVFS-modeling. Table III shows how different frameworks compare against each other in terms of the features they offer.

Brooks, Tiwari and Martonosi developed Wattch, one of the first frameworks to estimate power consumption at the architectural level [7]. It used power models to estimate energy consumption of different structures (array structures, combinational logic, clock distribution circuit, wires, etc.) in order to project how architecture-level decisions impact the power consumption of a system. Wattch was developed as an extension to SimpleScalar, and for several years the fusion of the two was the most commonly used power-performance simulator. Nowadays, however, this framework is somewhat outdated, since it does not offer any full-system and DVFS support, and, regarding the power-modeling part, it does not follow the latest technology advances.

More recently, McPAT makes use of updated power models and projections about technology nodes to deliver a tool that was validated against contemporary processors [25]. McPAT has served as an extension to performance simulators, such as gem5 and Sniper [8, 16], however, McPAT's XML interface does not support coefficients for many different voltage-frequency operating points, thus these tools cannot be readily used for DVFS explorations even in the cases that the performance-modeling part of the framework supports frequency scaling (e.g., SNIPER.) Comparing Sniper and gem5, the main difference is that gem5 is a cycle-accurate, full-system simulator, whereas Sniper is a fast simulator based on interval models for the core and does not support full-system simulation. Thus, Sniper cannot run unmodified DVFS governors. On the other hand, gem5 can model a whole SoC booting Linux/Android, and with our extensions it now fully supports DVFS.

Various researchers have characterized existing hardware using statistical models that correlate power consumption with performance counter events [20, 10, 18, 14, 31]. All of these power-modeling techniques are orthogonal to our work for integrating power models in gem5, since any technique can be used to derive the coefficients to be used in the simulator.

Table III
Comparing different simulation frameworks

Simulator	Cycle-accurate	Full System	Full DVFS Support	up-to-date power models
SimpleScalar+ Wattch	yes	no	no	no
gem5+McPAT	yes	yes	no	yes
Sniper+McPAT	no	no	no	yes
gem5 standalone	yes	yes	yes	yes

Regarding DVFS-Management, the Linux kernel offers a series of standard cpufreq governors for the user to pick from [4, 28] and are mainly based on detecting periods of low CPU load to scale frequency down and save energy. Other methods rely on empirical [19] or analytical [23, 30, 12] techniques to detect phases on an application's execution that memory is the bottleneck. These methods exploit that fact by scaling down core frequency without significantly harming performance and, at the same time, achieving high energy savings.

VIII. Conclusions and Future Work

In this paper we demonstrate how a full-system simulator can be extended with DVFS support. We enhance gem5 with features that are not available in most simulators: clock and voltage domain declaration, online power-estimation, a DVFS controller, and kernel drivers for full-DVFS support. The most important contribution of our work is that it is not a "hack": instead of taking shortcuts to come up with some implementation that "sort of works", we develop a framework that comes with robust specifications and complies with all the hardware and software conventions, so that it is highly configurable and can be used along with existing cpufreq governors. We intend to release the software as an open-source tool, so that in the future it will be integrated with the mainline gem5 source code.

To showcase the usefulness of our framework, we perform a series of DVFS experiments looking on both software and hardware aspects. We experiment with different governors and DVFS transition latencies/frequency operating points, and we reason about the energy efficiency of different DVFS schemes.

As future work, we plan to enhance gem5 with even more power-control features, such as idle-power management, thermal monitoring and power sensors, so that it can serve as a tool for exploring numerous energy-efficiency optimizations.

IX. Acknowledgements

We would like to thank the reviewers for their valuable feedback. Also, we would like to thank HiPEAC for funding Vasileios' internship at ARM, during which this work was performed. Finally, we appreciate all the help we received from ARM R&D Systems Group regarding gem5-related issues.

References

[1] Arndale board. URL `http://www.arndaleboard.org`.

[2] Big.little processing with arm cortex-a15 & cortex-a7. URL `http://www.arm.com/files/downloads/`.

[3] Samsung exynos. URL `http://www.samsung.com`.

[4] Linux kernel. URL `https://www.kernel.org/`.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, 2000.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, 2011.

[9] M. Catanzaro and D. Kudithipudi. Reconfigurable rram for lut logic mapping: A case study for reliability enhancement. In *SOC Conference (SOCC), 2012 IEEE International*, pages 94–99, 2012.

[10] G. Conrteras and M. Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. In *Int. Symposium on Low Power Electronics and Design*, 2005.

[11] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 102–103, 2007.

[12] S. Eyerman and L. Eeckhout. A counter architecture for online dvfs profitability estimation. *Computers, IEEE Transactions on*, 2010.

[13] E. G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, 2001.

[14] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Green Computing Conference, 2010 International*, pages 135–146. IEEE, 2010.

[15] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 81–90. IEEE, 2011.

[16] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 3–12, 2012.

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, volume 3b: system programming guide edition, September 2010.

[18] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Int. Symposium on Microarchitecture*, 2003.

[19] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Int. Symposium on Microarchitecture*, 2006.

[20] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Int. Symposium on Low Power Electronics and Design*, 2001.

[21] A. B. Kahng, S. Kang, T. Rosing, and R. Strong. Tap: token-based adaptive power gating. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ISLPED '12, pages 203–208, 2012.

[22] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008. ISBN 0-471-05669-3.

[23] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval based models for run-time dvfs orchestration in superscalar processors. In *Int. Conf. on Computing Frontiers*, 2010.

[24] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134. IEEE, 2008.

[25] 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.

[26] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[27] R. Miftakhutdinov, E. Ebrahimi, and Y. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012.

[28] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[29] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9. IEEE, 2007.

[30] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, 2011.

[31] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 241–250. IEEE, 2012.

[32] E. Tomusk and M. O'Boyle. Weak heterogeneity as a way of adapting multicores to real workloads. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '13, pages 5:1–5:3, 2013.