

Ameliorating Dark silicon in Multi-core Systems by DVFS

M.Hariharan 16BCE1293 , J.S.Sandeep Kiran 16BCE1041



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computing Sciences and Engineering, VIT Chennai, Tamilnadu, India 600127

1. Abstract

In this paper, we propose a Dynamic Voltage and Frequency Scaling (DVFS) approach as an alternative to control the power consumption of any computer system, ranging from mobile phones to servers. The goal of this paper is to primarily run 2 different parallel benchmark applications, one the Stanford SingleSource benchmarks (lightweight parallel code) and the other, Stanford Parallel Applications for Shared-Memory (SPLASH-2) programs (heavy real-time data streaming parallel applications that run on multi-core clusters). We run them on the gem5 Full System simulator, booting a linaro based linux kernel.

To get an holistic big picture of the power and thermal properties of systems that run on DVFS, apart from the DVFS-control features in gem5, we also add a power-estimation framework which is required for evaluating the efficiency of various DVFS policies, the MCPAT and Hotspot. Finally, based on the run Stanford workloads, we judge our DVFS model in terms of memory footprint and on other critical processor performance metrics, like running time per core, bus latency, number of read, writes, access time, cache hit or miss, etc.

We add the notion of clock and voltage domains to the simulator, as well as the simulation structures that manage voltage and frequency scaling. We extend gem5 with full DVFS support. On the hardware side, we procure and run on a kernel 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

2. Introduction

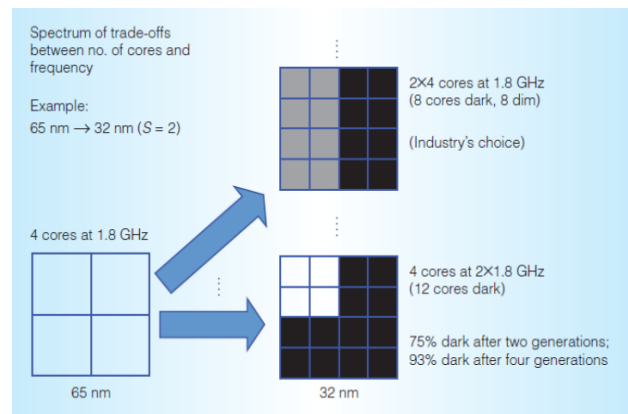


Figure 1. Multicore scaling leads to large amounts of dark silicon.³ Across two process generations, there is a spectrum of trade-offs between frequency and core count; these include increasing core count by 2 \times but leaving frequency constant (top), and increasing frequency by 2 \times but leaving core count constant (bottom). Any of these trade-off points will have large amounts of dark silicon.

In the early 1990s, there were hardly 30000 transistors per chip, even in the Microsoft's first x86 processors. Today, CPUs and GPUs have transistor counts measured in the billions. Adding a few thousand transistors to make a particular task faster doesn't noticeably increase the cost, but a new problem has arisen: power usage, which is closely related to heat dissipation. For every watt

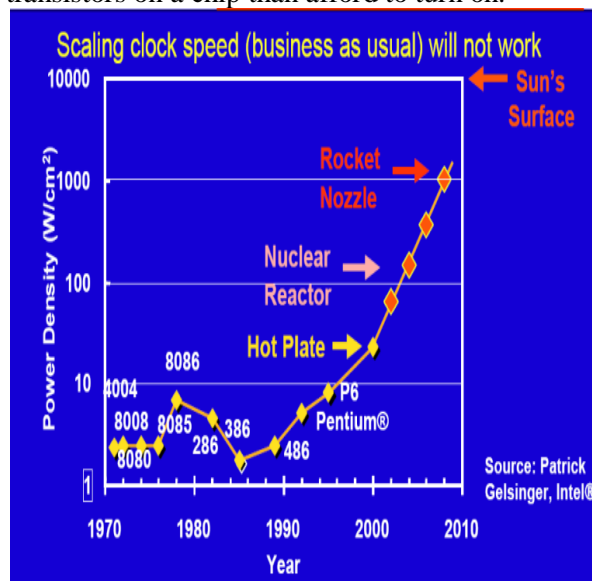
of power the CPU consumes, it must dissipate a watt of heat.

Unfortunately, while the number of transistors that can be put on a chip cheaply has increased, the power consumption per transistor hasn't dropped at a corresponding rate. The amount of power per transistor has dropped, but more slowly than the size of the transistor has shrunk. Eventually, you hit some real physical limits;

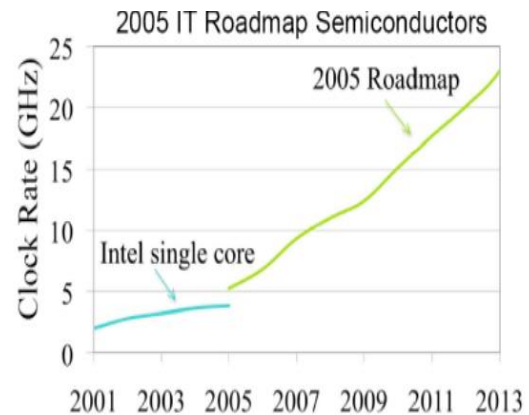
Intel Pentium 4, was the first to encounter this problem and scientists at Intel were just edging

into the regimes of a booming futuristic research domain and the goal of this paper is to help find new solutions to combat this hardware constraint.

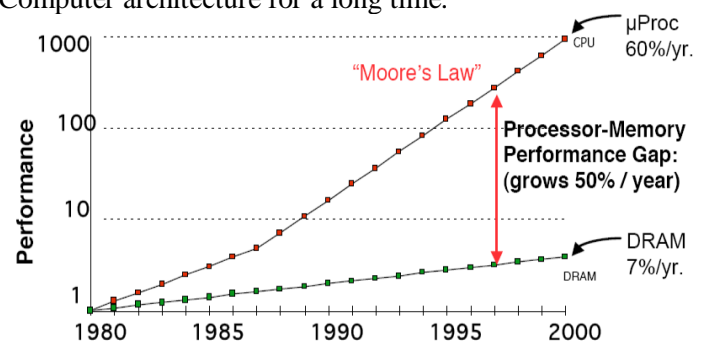
There's a practical limit to frequency scaling. As for why can't we put billions of transistors on a chip or operate the transistors the entire system under production power budget, comes from these 3 key hardware bottlenecks in the silicon industry. First, the power wall, that is you cant put more transistors on a chip than afford to turn on.



Second, the frequency wall, as the dynamic power in a chip is proportional to V^2fC , which means increasing 'f' will lead to power wall.



Third, the gap between processor and memory speeds. The continuous growing gap between CPU and memory speed is a problem in the Computer architecture for a long time.



Gem5

Gem5 is a full-system simulation infrastructure that attempts to merge the best aspects of the M5 and GEMS simulators. Its main selling point is its pervasive object oriented nature that allows for easy modification and addition of new components. The python scripting further helps to design a fully functional COTS system, right out of the bat, by integrating various hardware components from the gem5 software libraries. It has 2 basic modes of operation: SE and FS modes. FS (full-system) mode simulates all aspects of program execution, including the system calls to the operating system. SE (syscall emulation) mode has gem5 emulate the operating system calls.

MCPAT

McPAT (Multicore Power, Area, and Timing) is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. It models power, area, and timing simultaneously and consistently and supports comprehensive early stage design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers.

3.Related Study

A research paper presented by Dr.Lee and team on the notion ,”By increasing the number of transistors in a single chip”, coupled with breakdown of Dennardian scaling and increasing the on-chip power density, temperature and power management is a necessity in the current and future technologies In addition, different activity rate of functional blocks, non-uniform workload variation, and advanced static and dynamic power management capabilities in recent CMPs result in non-uniform power distribution on the substrate which leads to significant temperature gradient]. Large temperature variation across a chip decreases the reliability of the circuits and degrades their performance. Several research studies in the field of dynamic thermal management (DTM) aim at mitigating temperature and power violations at runtime in many-core systems. An efficient DTM technique necessities accurate on-chip thermal sensors in recent technologies to maximize the performance under a restricted chip temperature

DVFS is a generalized technique for dynamic power management. DVFS is used for reducing CPU power consumption, as CPU power consumption has a big portion of total power consumption in a processor. CPU is composed of

CMOS semiconductor and (2) is CMOS energy consumption:

$$P_{dyn} = N_{sw}C_LV_{dd}^2f$$

N_{sw} is the switching activity , C_L is CMOS circuit output load capacitance , V_{dd} is supply voltage and f is the operating frequency.

Power Consumption in CPU $E \propto V^2$ and f

Power Consumption is proportional to frequency and the square of supply voltage. Therefore, voltage and frequency scaling can influence on power consumption. A technique which is called DVFS. However, Disadvantage of DVFS is transition overhead can be generated when voltage and frequency are scaled.

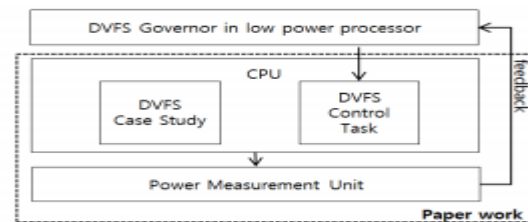


Fig. 2 DVFS System

Related Study in regard to Simulation frameworks

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 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 provided a unified power, area and timing estimation framework, and quickly developed as the most popular power extension for performance simulators. Several researchers have combined the gem5 simulator 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.

Related Study in regard to Dark Silicon and DVFS

The Original Moore's law stated that, usually cast on the Processor industry as, "the number of transistors per chip, at constant cost doubles every 18-24 months".

This has not been true for years. The improvement has been remarkable, but it is getting increasingly difficult to maintain this exponential improvement. Its not because of parallelism – for most of its life, Moore's Law and single processor performance correlated well

The reason is the size and speed are related – the smaller something is, the quicker it can be changed Thus, smaller transistors can switch at higher speeds

Dennardian Scaling and the Rise of the Dark Silicon problem



Why haven't clock speeds increased, even though transistors have continued to shrink? • Dennard (1974) observed that voltage and current should be proportional to the linear dimensions of a transistor ♦ Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor

$$\text{Power} = \alpha * CFV^2$$

Alpha – percent time switched , C = capacitance , F = frequency , V = voltage • Capacitance is related to area . So, as the size of the transistors shrunk, and the voltage was reduced, circuits could operate at higher frequencies at the same power .

The End of Dennard Scaling

Historically, the transistor power reduction afforded by Dennard scaling allowed manufacturers to drastically raise clock frequencies from one generation to the next without significantly increasing overall circuit power consumption.

Since around 2005–2007 Dennard scaling appears to have broken down. As of 2016, transistor counts in integrated circuits are still growing, but the resulting improvements in performance are more gradual than the speed-ups resulting from significant frequency increases. The primary reason cited for the breakdown is that at small sizes, current leakage poses greater challenges, and also causes the chip to heat up, which creates a threat of thermal runaway and therefore further increases energy costs.

Dennard scaling ignored the “leakage current” and “threshold voltage”, which establish a baseline of power per transistor. As transistors get smaller, power density increases because these don’t scale with size. These created a “Power Wall” that has limited practical processor frequency to around 4 GHz since 2006

The breakdown of Dennard scaling and resulting inability to increase clock frequencies significantly has caused most CPU manufacturers to focus on multicore processors as an alternative way to improve performance. An increased core count benefits many (though by no means all) workloads, but the increase in active switching elements from having multiple cores still results in increased overall power consumption and thus worsens CPU power dissipation issues.

The end result is that only some fraction of an integrated circuit can actually be active at any given point in time without violating power constraints. The remaining (inactive) area is referred to as dark silicon.

4.Our Work

The goal of this paper is to primarily run 2 different parallel benchmark applications, one the Stanford SingleSource benchmarks (lightweight parallel code) and the other, Stanford Parallel Applications for Shared-Memory (SPLASH-2) programs (heavy real-time data streaming parallel applications that run on multi-core clusters). We run them on the gem5 Full System simulator, booting a linaro based linux kernel. We also modify gem5 into a complete hardware-software framework suitable for full-system DVFS studies, by enabling DVFS Handlers to operate under a Voltage Domain to scale a set of CPU frequencies. We further setup the DVFS controller onto the Linux Device Tree, by loading a DVFS enabled vexpress bootloader.

To get an holistic big picture of the power and thermal properties of systems that run on DVFS, apart from the DVFS-control features in gem5, we also add a power-estimation framework which is required for evaluating the efficiency of various DVFS policies, the MCPAT and Hotspot. Finally,

based on the run Stanford workloads, we judge our DVFS model in terms of memory footprint and on other critical processor performance metrics, like running time per core, bus latency, number of read, writes, access time, cache hit or miss, etc.

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
- we extend gem5 with full DVFS support. On the hardware side, we procure and run on a kernel 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
- we show how these extensions can be used to perform full-system power efficiency studies by also including a visualization of a flamegraph model of our simulated gem5 system.

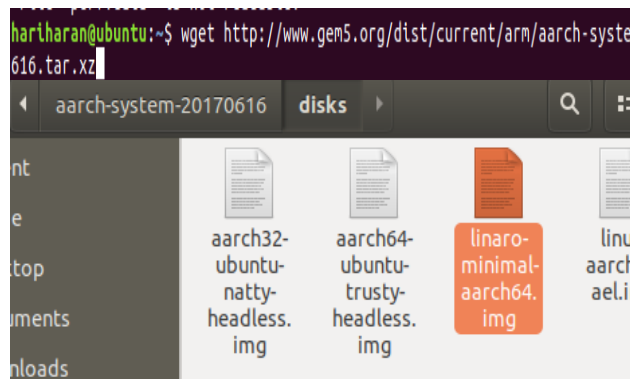
5.Implementation

Gem5 implementation details

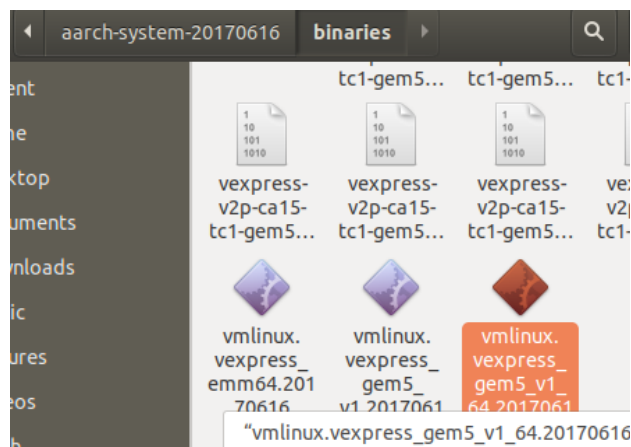
We use gem5 because it already supports full-system simulation, which is crucial for enabling DVFS. gem5 is capable of booting Linux, which already feature modules for DVFS management, thus we do not need to develop the whole software stack from scratch. Moreover, I/O devices are fully supported in gem5,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

use existing simulation structures to implement a memory-mapped DVFS controller.

In the FS mode, the complete system can be modeled in an OS-based simulation environment. In order to simulate using the FS mode, we have to take some extra steps by downloading the Arm Full-System Files from the gem5 Download page.

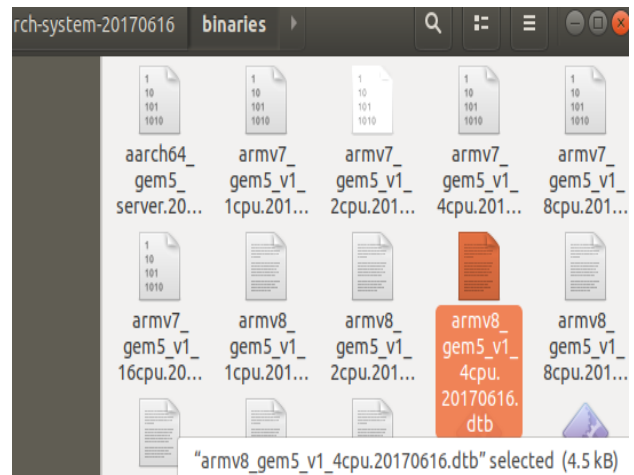


We use the Linaro's Linux kernel, which provides support for GNU Compiler Collection (GCC), power management, graphics and multimedia interfaces for the ARM family of instruction sets and implementations thereof as well as for the Heterogeneous System Architecture (HSA). In this research work, we focus on the use of the ARM instruction set in its version8 (64-bit) on the Cortex A9 core.



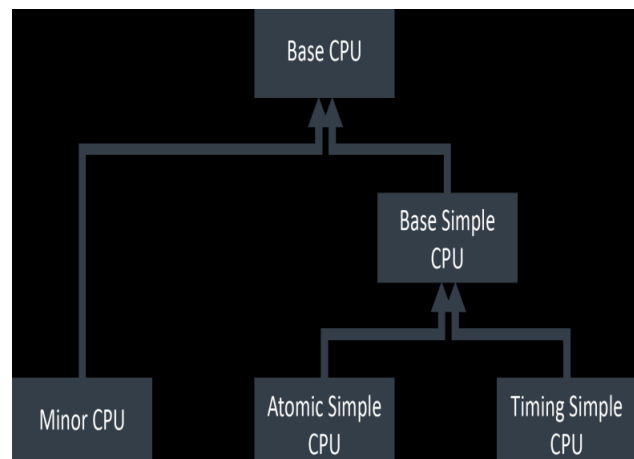
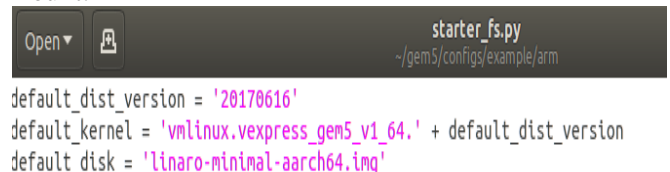
The linaro's linux kernel, vmlinux, is tailor fit to run on a gem5 simulator and we may want to recompile this from source to add our own syscall implementations.

A "dtb" file contains a Device Tree Blob (or Binary)(nice description [here](#)). It's the new(er) way to pass hardware information about the board to the Linux kernel, and is the default boot loader for our vmlinux OS. It can be loaded into memory and passed to the kernel during boot-up. We use the arm v8 bootloader DTB goes with the kernel.



We present a simplified example of how a gem5 configuration script is built to run in a Full system simulation mode.

First, we choose the kernel, DTB files to load into our simulated ARM, the file system disk image to mount.



CPU Types and Memory Footprints

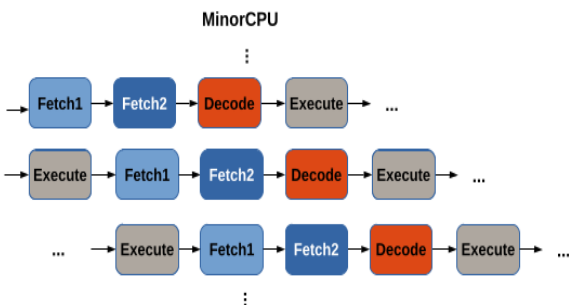
The 3 CPU models offered up by ARM vary on memory access and there are three types of access supported in gem5.

First, the atomic access is the fastest of the three, and completes a transaction in a single function call. Second, functional access is similar to atomic access in that it completes a transaction in a single function call and the access happens instantaneously, but additionally, functional accesses can coexist in the memory system with Atomic or Timing accesses. Third, Timing access is used for approximately-timed simulation, which considers the realistic timing, and models the queuing delay and resource contention

```
cpu_types = {
    "atomic" : ( AtomicSimpleCPU, None, None, None, None, None ),
    "minor" : ( MinorCPU,
                devices.L1I, devices.L1D,
                devices.WalkCache,
                devices.L2 ),
    "hpi" : ( HPI.HPI,
              HPI.HPI_ICache, HPI.HPI_DCache,
              HPI.HPI_WalkCache,
              HPI.HPI_L2 )
}
```

The AtomicCPU performs atomic memory access, while the TimingSimpleCPU adopted Timing memory access instead of the simple Atomic one. This means that it waits until memory access returns before proceeding, therefore it provides some level of timing.

MinorCPU, comprehensive and detailed CPU mode for emulating realtime systems uses the functional mode of memory access and implements a four-stage pipeline which includes fetching lines, decomposition into macro-ops, decomposition of macro-ops into micro-ops and execute.



The High Performance In-order processor in the current ARM v8, is an upgraded MinorCPU that extends to include a floating-point data unit.

Falling within the scope of this paper for measuring CPU and memory performance, we'll use the MinorCPU mode.

```
cpu_class = cpu_types[args.cpu][0]
mem_mode = cpu_class.memory_mode()
```

Figure: Choose the CPU type based on the cmd line args, and the compatible memory access mode.

```
system = devices.SimpleSystem(want_caches,
                              args.mem_size,
                              mem_mode=mem_mode,
                              dtb_filename=dtb_file,
                              kernel=SysPaths.binary(args.kernel),
                              readfile=args.script)
```

Figure: Create a gem5 system, from the args parser mem_size (256MB for RAM), DTB and kernel. Optionally, we may pass in a rCS boot script, that always during a linux boot.

```
for dev in system.pci_devices:
    system.attach_pci(dev)
```

Figure: Attach the PCI devices to the system. The helper method in the system assigns a unique PCI bus ID to each of the devices and connects them to the IO bus.

```
system.cpu_cluster = [
    devices.CpuCluster(system,
                       args.num_cores,
                       args.cpu_freq, "1.0v",
                       *cpu_types[args.cpu]),
    ]
```

Figure: Add CPU clusters to the system. This is a non-DVFS sys. So, simply set a constant supply voltage and the rest(freq, cores per chip, cpu type) user-friendly (init from cmd line args)

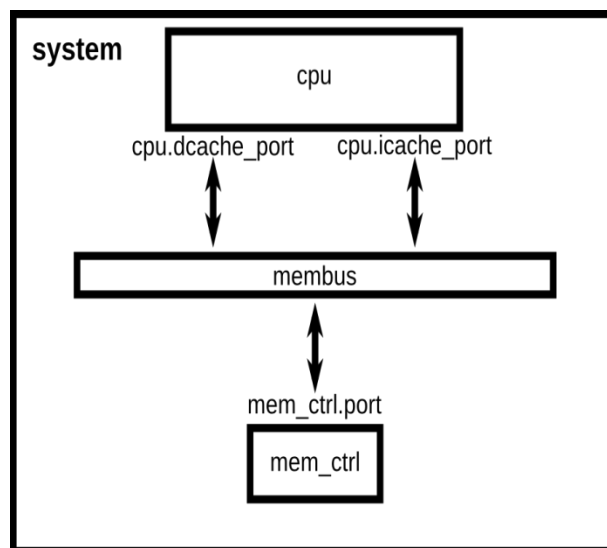
```
for cluster in system.cpu_cluster:
    system.addCaches(want_caches, last_cache_level=2)
```

Figure: Create a cache hierarchy for the cluster. We are assuming that clusters have core-private L1 caches and an L2 that's shared within the cluster on the chip. Based on our CPU type, we pick the right Cache levels and associativity, else leave it to defaults.

```
system.connect()
```

Figure: Wire up the system's memory system

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



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.

Running the simulation

Only based on our parser args in the python config script, we pass in custom arguments.

```
hartharan@ubuntu:~/gem5$ ./build/ARM/gem5.opt configs/example/arm/starter_fs.py
--num-cores=4 --mem-size=256MB
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.
```

Note that the system defaults to an EnergyController attached to the System bus,

rather than using a DVFSController.

```
info: Using bootloader at address 0x10
info: Using kernel entry physical address at 0x80000000
info: Loading DTB file: /home/hartharan/gem5/aarch-system-20170616/binaries/armv
8_gem5_v1_4cpu.20170616.dtb at address 0x80000000
warn: Existing EnergyCtrl, but no enabled DVFSHandler found.
info: Entering event queue @ 0. Starting simulation...
```

We can access the simulated system, after bootup, in the exposed loopback port using Telnet or m5term. We used m5term as it intercepts more of processor info than connections/network info.

```
hartharan@ubuntu:~$ m5term localhost 3456
==== m5 slave terminal: Terminal 0 ====
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Linux version 4.4.0+ (root@bbdeb8fab105) (gcc version 5.4.0 20160
609 (Ubuntu/Linaro 5.4.0-6ubuntu1-16.04.4) ) #1 SMP PREEMPT Fri Jun 16 09:13:26
UTC 2017
[ 0.000000] Boot CPU: AArch64 Processor [410fc0f0]
[ 0.000000] Memory limited to 256MB
[ 0.000000] cma: Reserved 16 MiB at 0x000000000f000000
[ 0.000000] PERCPU: Embedded 15 pages/cpu @fffffc00efb8000 s23320 r8192 d299
28 u61440
Last login: Mon Jan 27 08:00:03 UTC 2014 on tty1
root@genericarmv8:~# help
GNU bash, version 4.2.10(1)-release (aarch64-oe-linux-gnu)
These shell commands are defined internally. Type 'help' to see this list.
Type 'help name' to find out more about the function 'name'.
Use 'info bash' to find out more about the shell in general.
Use 'man -k' or 'info' to find out more about commands not in this list.
```

Figure: A final bash interface to the remotely simulated ARM, opens up on the m5term attached terminal to 3456.

Writing ARM config scripts for handling DVFS

Custom Clock and Voltage Domain tuning

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.


```
# Create a CPU voltage domain
system.cpu_voltage_domain = VoltageDomain(voltage = args.cpu_voltage)

# Create a source clock for the CPUs and set the clock period
system.cpu_clk_domain = SrcClockDomain(clock = args.cpu_clock, voltage_d
system.cpu_voltage_domain, domain_id = 0)

system.dvfs_handler.domains = system.cpu_clk_domain
system.dvfs_handler.enable = 1
```

Script2 Figure: Create custom voltage and frequency domains from the args. And enable the DVFS handler in the kernel to make use of the values available in these domains.

```
parser.add_argument("--cpu-clock", type=list, default=['1 GHz', '750 MHz', '500 MHz'],
                    help = "the CPU freq domains to be used")

parser.add_argument("--cpu-voltage", type=list, default=['1V', '0.9V', '0.8V'],
                    help = "the CPU voltage domains to be used for DVFS")

args = parser.parse_args()
```

Figure: Expose these params on the parser's arguments. Unless its user set, it defaults.

The Clock Domain

In gem5, 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.

The ClockDomain class defines the clock period that, all objects belonging to that domain, will operate under. Any changes in the clock made by the DVFS handler is now communicated to each of the components operating in that domain.

Voltage Domain

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.

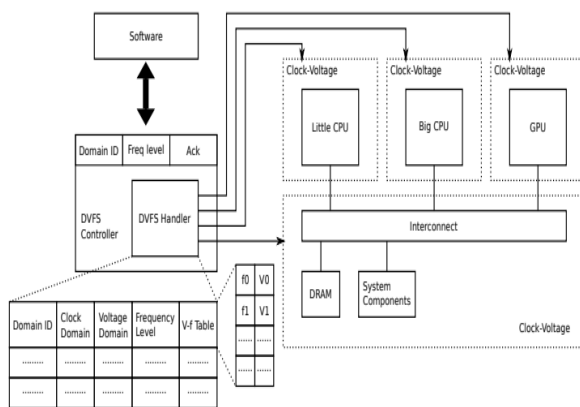
The Script2 shows how the example configuration script of Script1 can be updated to declare clock and voltage domains and assign components to them. In the example we first create a voltage domain and then a clock domain that is registered to that voltage domain. The CPU is assigned to that clock domain, 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

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.

DVFS in gem5

The 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, 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.)



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, because even between ARM v8 Cortex A9 and Cortex A8, their memory-maps are different.

So, the Controller is just an offline I/O device that performs runtime voltage-freq tuning and communicates with the OS by flagging its registers.

DVFS Handler

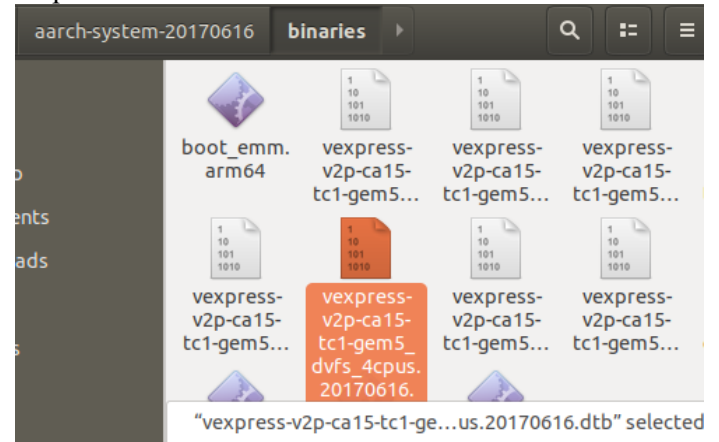
The DVFS Handler is a simulation object that handles all the clock and voltage domains in the system. Unlike the DVFS controller, which is an actual I/O device, the handler does not have a physical representation in the system; it is a simulation component which is part of the controller and maintains 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.

Linux Device Tree : The DTB file

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.

We use a different ARM DTB file within our config script to support this behavior, one that adopts DVFS across all 'n' cores.



```
if not args.dtb:
    dtb_file = SysPaths.binary("vexpress-v2p-ca15-tc1-gem5_dvfs_%scpus.%s.dtb" %
                                (args.num_cores, default_dist_version))
else:
    dtb_file = args.dtb
```

Figure: Boot loaders for extended device support

We may also use the dvfs per core Device Tree extension that sets up dedicated DVFS controllers for every core with multiple Handler instances.

Running the Full System Simulation on DVFS

```
hartharan@ubuntu:~/gem5$ ./build/ARM/gem5.opt configs/example/arm/dvfs_starter_fs_code.py --num-cores=4 --mem-size=256MB
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Oct  8 2018 02:49:38
gem5 started Nov 10 2018 21:28:29
gem5 executing on ubuntu, pid 36565
command line: ./build/ARM/gem5.opt configs/example/arm/dvfs_starter_fs_code.py -
-num-cores=4 --mem-size=256MB
```

Again we open up to a remote tunnel to the simulated ARM on m5term.

```
Stopping Bootlog daemon: bootlogd.
INIT: no more processes left in this runlevel
Last login: Mon Jan 27 08:00:03 UTC 2014 on tty1
root@genericarmv8:~# help
GNU bash, version 4.2.10(1)-release (aarch64-oe-linux-gnu)
```

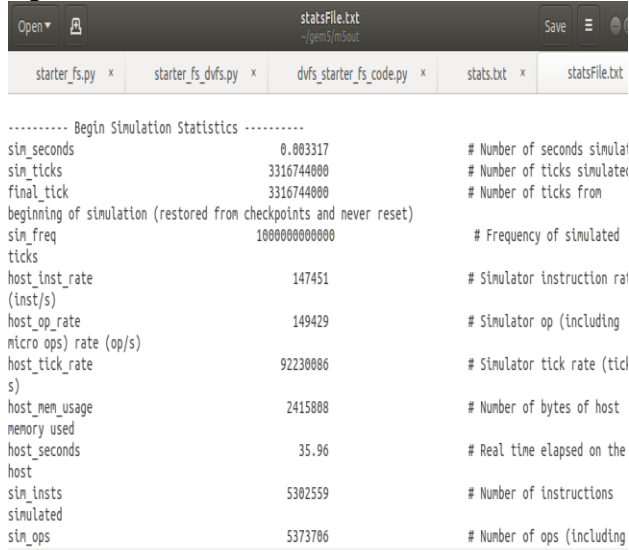
There's visually no big difference between the non-DVFS and DVFS run and remote terminal

access. But, the stats speak disparately for the 2 configs.

After running gem5, three files will be generated in the default output directory called m5out. This output directory can be changed by using the option -d.

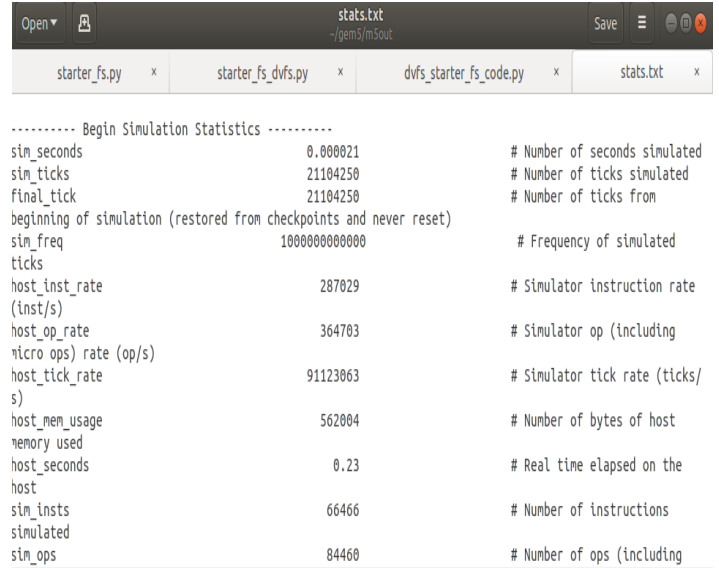
- Simulation parameters: the files named config.ini and config.json contain a list of each system component called SimObject as well as their parameters.
- Statistics: the file named stats.txt contains the statistics of each SimObject, which is detailed in the form of name, value, and description

Figure: Stats for DVFS



----- Begin Simulation Statistics -----		
sim_seconds	0.003317	# Number of seconds simulated
sim_ticks	3316744000	# Number of ticks simulated
final_tick	3316744000	# Number of ticks from
beginning of simulation (restored from checkpoints and never reset)		
sim_freq	1000000000000	# Frequency of simulated
ticks		
host_inst_rate	147451	# Simulator instruction rate
(inst/s)		
host_op_rate	149429	# Simulator op (including
micro ops) rate (op/s)		
host_tick_rate	92230006	# Simulator tick rate (ticks/
s)		
host_mem_usage	2415008	# Number of bytes of host
memory used		
host_seconds	35.96	# Real time elapsed on the
host		
sim_insts	5302559	# Number of instructions
simulated		
sim_ops	5373706	# Number of ops (including

Figure: Stats for non-DVFS



----- Begin Simulation Statistics -----		
sim_seconds	0.000021	# Number of seconds simulated
sim_ticks	21104250	# Number of ticks simulated
final_tick	21104250	# Number of ticks from
beginning of simulation (restored from checkpoints and never reset)		
sim_freq	1000000000000	# Frequency of simulated
ticks		
host_inst_rate	287029	# Simulator instruction rate
(inst/s)		
host_op_rate	364703	# Simulator op (including
micro ops) rate (op/s)		
host_tick_rate	91123063	# Simulator tick rate (ticks/
s)		
host_mem_usage	562004	# Number of bytes of host
memory used		
host_seconds	0.23	# Real time elapsed on the
host		
sim_insts	66466	# Number of instructions
simulated		
sim_ops	84460	# Number of ops (including

The first column is the name of the parameters, with their corresponding values in the second column and brief descriptions in the third column. For example, sim_seconds is the total simulated seconds, host_inst_rate is the instruction rate of running gem5 on the host, and sim_insts is the number of instructions simulated. After the basic simulation parameters, the statistics related to each module of the simulated system are printed. For example, system.clk_domain. clock is the system clock period in ticks.

This paper extensively discusses the stats in results section.

Running Benchmark Applications

We choose to run 2 set of benchmarks. One the Stanford SingleSource benchmarks (lighweight parallel code)and the Stanford Parallel Applications for Shared-Memory (SPLASH-2) programs (heavy real-time data streaming parallel applications that run on multi-core clusters)

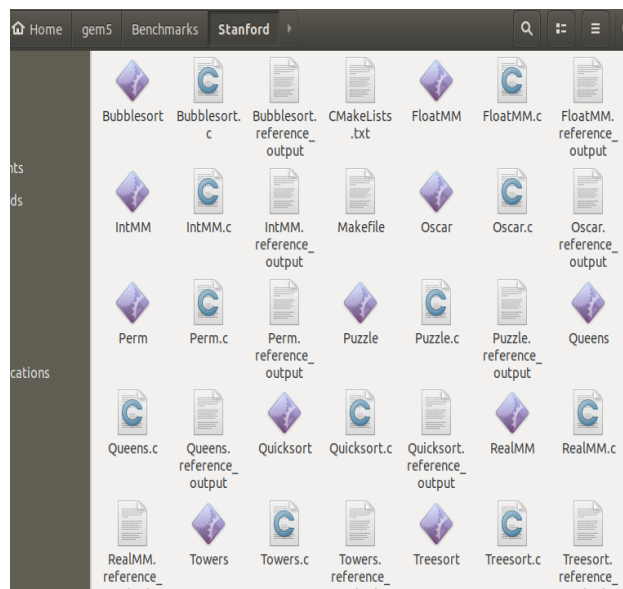
The Stanford SingleSource workloads from the LLVM test-suite for benchmarking in the FS mode.

The test-suite contains benchmark and test programs. The programs come with reference outputs so that their correctness can be checked.

The suite comes with tools to collect metrics such as benchmark runtime, compilation time and code size. The SingleSource contains test programs that are only a single source file in size. A subdirectory may contain several programs. The suite of programs capture several architectural worst-case scenarios and maximise running times for most architectures.

The `svn` command below downloads the benchmarks to our local machine:

```
hariharan@ubuntu:~/gem5$ svn export https://llvm.org/svn/llvm-project/test-suite
/tags/RELEASE_390/final/SingleSource/Benchmarks/
```



The benchmarks range from different sorting techniques, to memory map to puzzles and board games.

Next, we need Install the Arm cross compiler toolchain, so that we can cross-compile

```
hariharan@ubuntu:~/gem5$ sudo apt-get
```

Next, we write a makefile to cross-compile all the benchmark files to ARM and generate binary executables that run on ARM.

```
SRCS = $(wildcard *.c)
PROGS = $(patsubst %.c,%, $(SRCS))
all: $(PROGS)
%: %.c
    arm-linux-gnueabi-gcc --static $<
clean:
    rm -f $(PROGS)
```

We mount the ARM linaro disk and copy these executables to the mounted file system.

```
hariharan@ubuntu:~$ sudo mount -o loop,offset=32256 linaro-minimal-aarch64.img /mnt
hariharan@ubuntu:~/gem5/aarch-system-20170616/disks$ cd /mnt
hariharan@ubuntu:/mnt$ ls
bin dev home lost+found mnt run sys usr
boot etc lib media proc sbin tmp var
hariharan@ubuntu:/mnt$ sudo cp /home/hariharan/gem5/Benchmarks/Stanford/ -r /mnt/home/root
```

Fig: Copy the files to the root dir of the to be simulated ARM.

We can call these benchmarks during runtime by writing a `exec()` to these files in the rCS bootscript residing within the `/etc/init.d` on the linaro disk.

```
Open *rcs [Read-Only]
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel

# Make sure proc is mounted
# [ -d "/proc/1" ] || mount /proc

# Source defaults.
# . /etc/default/rcs
# Trap CTRL-C &c only in this shell so we can interrupt subprocesses.
trap ":" INT QUIT TSTP

# Call all parts in order.
exec /etc/init.d/rc S

exec /home/root/Bubblesort & /home/root/Queens & /home/root/Treesort & /home/root/powr_stats
```

Or we may pass the rCS bootscript during runtime.

```

hartharan@ubuntu:~/gem5$ ./build/ARM/gem5.opt configs/example/arm/dvfs_starter_fs_code.py --num-cores=4 --mem-size=256MB --script=configs/boot/stan_bench.rcs
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Oct  8 2018 02:49:38
gem5 started Nov 10 2018 22:55:48
gem5 executing on ubuntu, pid 36812
command line: ./build/ARM/gem5.opt configs/example/arm/dvfs_starter_fs_code.py --num-cores=4 --mem-size=256MB --script=configs/boot/stan_bench.rcs

```

Either way, the simulation works.

The 4 Benchmarks Bubblesort, Queens, TreeSort and another powr_stats are run parallelly on the CPU cluster. The powr_stats is a simple piece of code written on our own, to estimate running power and energy consumed dynamically. Its affinity is set to run on the core 3 in DVFS mode, as from the experiments, core 3 in DVFS is seen to be utilized the least when it comes to running large parallel applications, like the SPLASH2.

```

int main(void) {
    cpu_set_t mask;

    print_affinity();
    printf("sched_getcpu = %d\n", sched_getcpu());
    CPU_ZERO(&mask);
    CPU_SET(0, &mask);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &mask) == -1) {
        perror("sched_setaffinity");
        assert(false);
    }
    print_affinity();
    /* TODO is it guaranteed to have taken effect already? Always worked on my
    printf("sched_getcpu = %d\n", sched_getcpu());
    system("powerstat");

```

Fig: A peek into a sample section of power stats.

```

Running for 280.0 seconds (28 samples at 10.0 second intervals).
Power measurements will start in 200 seconds time.

```

Time	User	Nice	Sys	Idle	IO	Run	Ctxt/s	IRQ/s	Watts
22:53:30	6.4	0.0	0.4	93.2	0.0	1	557	107	14.88
22:53:40	31.6	0.0	2.3	66.1	0.0	2	826	218	14.88
22:53:50	9.3	0.0	0.6	89.6	0.4	1	409	102	14.88
22:54:00	8.6	0.0	0.5	90.6	0.3	2	636	115	14.88
22:54:10	23.8	0.0	1.5	74.3	0.4	2	737	182	14.88
22:54:20	25.7	0.0	1.2	72.9	0.2	2	396	152	14.88
22:54:30	5.1	0.0	0.8	93.6	0.4	1	629	111	14.88
22:54:40	8.7	0.0	1.4	89.9	0.0	1	580	111	14.88
22:54:50	4.7	0.0	0.6	94.0	0.6	2	625	109	14.88
22:55:00	7.3	0.0	0.7	92.0	0.0	2	600	107	29.75
22:55:10	4.8	0.0	0.6	94.3	0.3	2	479	95	29.75
22:55:20	5.7	0.0	0.8	93.3	0.2	1	645	106	14.88
22:55:30	3.2	0.0	0.5	95.6	0.7	1	498	89	13.85
22:55:40	6.5	0.0	1.2	92.0	0.3	2	640	113	13.85
22:55:50	3.6	0.0	0.4	96.0	0.0	2	582	94	13.85
22:56:00	6.1	0.0	1.2	92.3	0.3	3	606	101	13.85
22:56:10	4.4	0.0	0.4	94.5	0.6	2	674	113	13.85
22:56:20	5.5	0.0	1.1	93.3	0.0	3	555	98	13.85
22:56:30	3.9	0.0	0.6	95.3	0.2	1	537	93	13.85
22:56:40	6.6	0.0	0.7	92.5	0.2	2	642	113	13.85
22:56:50	4.3	0.0	0.3	93.7	1.6	2	657	106	13.85
22:57:00	6.7	0.0	1.2	91.5	0.5	1	589	106	13.85
Time	User	Nice	Sys	Idle	IO	Run	Ctxt/s	IRQ/s	Watts
22:57:10	3.6	0.0	0.6	95.8	0.0	1	597	103	0.00
22:57:20	5.5	0.0	0.8	93.1	0.5	2	585	99	0.00
22:57:30	4.3	0.0	0.5	94.9	0.3	3	578	95	13.85

Average	8.2	0.0	1.0	90.1	0.8	1.8	607.6	116.4	13.87
GeoMean	6.6	0.0	0.8	89.7	0.0	1.6	598.3	112.8	0.00
StdDev	6.9	0.0	0.7	8.1	2.4	0.6	109.5	34.3	6.24
Minimum	3.2	0.0	0.3	66.1	0.0	1.0	395.5	89.4	0.00
Maximum	31.6	0.0	4.0	96.4	13.3	3.0	971.1	225.4	29.75
Summary:									
System: 13.87 Watts on average with standard deviation 6.24									
Note: Power calculated from battery capacity drain, may not be accurate.									

SPLASH2- Stanford Parallel Applications for Shared-Memory

The Splash are full applications and kernels. Each of the codes utilizes the Argonne National Laboratories(ANL) parmac macros for parallel constructs. Each of the codes in SPLASH assumes the use of a "lightweight threads" model in which child process share the same virtual address space as their parent process. In order for the codes to function correctly, the CREATE macro should call the proper Unix system routine (e.g. "sproc" in the Silicon Graphics IRIX operating system) The difference is that processes created with the In the threads model, child processes share the same virtual address space, and hence all global data. Some of the codes function correctly when the Unix "fork" command is used for child process creation as well.

We identify some of the programs as "core" programs that should be used in most studies for comparability. In the currently available set, these core programs include:

- (1)Ocean-Simulation
- (2)Hierarchical-Radiosity
- (3)Water Simulation with Spatial data structure
- (4)Barnes-Hut
- (5)FFT
- (6)Blocked Sparse Cholesky Factorization
- (7)Radix-Sort

More information on each of these real-time Big data scientific applications can be found on the official site. <https://github.com/staceyson/splash2>

Running benchmarks on gem5

Fig: Sample run of the Stanford Benchmark. On gem5 with DVFS enabled and without DVFS.


```

hariharan@ubuntu:~/gem5$ ./build/ARM/gem5.opt --outdir=dvfs_stanford config
mple/arm/dvfs_starter_fs_code.py --num-cores=4 --mem-size=256MB --script=c
/boot/stanford_benchmarks.rCS
hariharan@ubuntu:~/gem5$ ./build/ARM/gem5.opt --outdir=non_dvfs_stanford c
/example/arm/starter_fs.py --num-cores=4 --mem-size=256MB --script=configs
stanford_benchmarks.rCS
rpcbnd: cannot create socket for tcp6
rpcbnd: cannot get uid of '': Success
done.
creating NFS state directory: done
starting statd: done
Starting auto-serial-console:
Running Stanford benchmarks
sched_getaffinity = 1 1 1 1
sched_getcpu = 1
sched_getaffinity = 1 0 0 0
sched_getcpu = 0
-15239

```

```

hariharan@ubuntu: ~
File Edit View Search Terminal Help

Running Stanford benchmarks
sched_getaffinity = 1 1 1 1
sched_getcpu = 1
sched_getaffinity = 1 0 0 0
sched_getcpu = 0
1
2
-50000
-15239
3
4
-49783
5
-15986
6
7
-49692
-10769
8
9
-49678
10
2540

```

Note how during the rCS execution, the booting up output is rightly thrown to the console alongwith the power_stats code that specifically runs on Core 3 out of 4.

Similarly, we simulated the SPLASH set of benchmarks, one on enabling DVFSHandler and another at constant clocking.

```

mounting filesystems...
loading script...
Running FFT

FFT with Blocking Transpose
1024 Complex Doubles
1 Processors
65536 Cache lines
16 Byte line size
4096 Bytes per page

PROCESS STATISTICS
Proc      Computation Time      Transpose Time      Transpose Fraction
0         0              0              0              nan

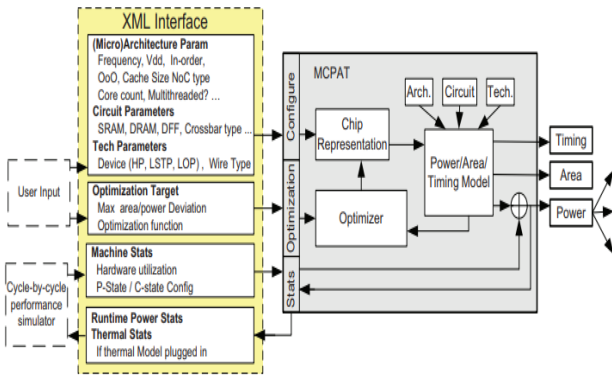
TIMING INFORMATION
Start time           : 1230768001818359
Initialization finish time : 1230768001819335
Overall finish time   : 1230768001819335
Total time with initialization : 976
Total time without initialization : 0
Overall transpose time : 0

```

McPAT

McPAT (Multicore Power, Area, and Timing) is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. It models power, area, and timing simultaneously and consistently and supports comprehensive early stage design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers. McPAT models timing, area, and dynamic, short-circuit, and leakage power for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors. McPAT has a flexible XML interface to facilitate its use with different performance simulators.

MCPAT Framework



The Above is block diagram of the McPAT framework. Rather than being hardwired to a particular simulator, McPAT uses an XML-based interface with the performance simulator. McPAT uses an XML parser developed by Berghen et.al to parse the large XML interface file. This interface allows both the specification of the static microarchitecture configuration parameters and the passing of dynamic activity statistics generated by the performance simulator.

McPAT can also send runtime power dissipation results back to the performance simulator through the XML-based interface, so that the performance simulator can react to power or even temperature data. This approach makes McPAT very flexible and easily ported to other performance simulators.

Since McPAT provides complete hierarchical models from the architecture to technology level, the XML interface also contains circuit implementation style and technology parameters that are specific to a particular target processor. Examples are array types, crossbar types, and CMOS technology generations with associated voltage and device types.

The key components of McPAT are (1) the hierarchical power, area, and timing models, (2) the optimizer for determining circuit level implementations, and (3) the internal chip representation that drives the analysis of power, area, and timing. Most of the parameters in the internal chip representation, such as cache capacity and core issue width, are directly set by the input parameters. McPAT's hierarchical structure allows it to model structures at a low level including underlying device technology, and

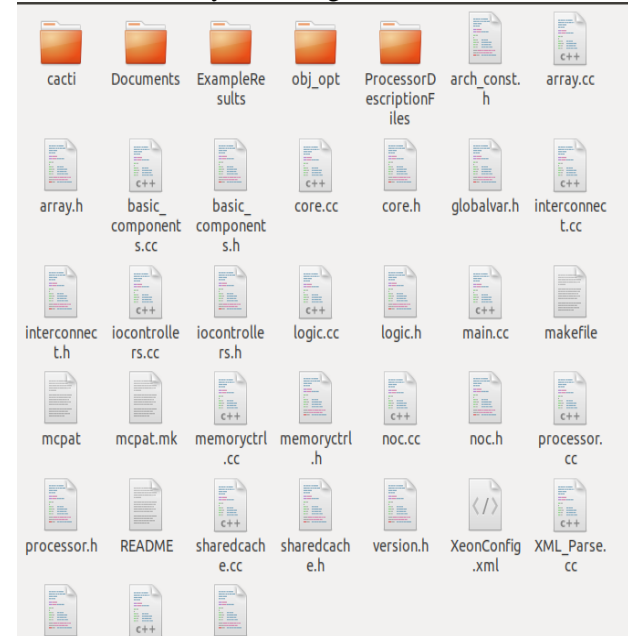
yet still allows an architect to focus on a high-level architectural configuration.

MCPAT Implementation Details

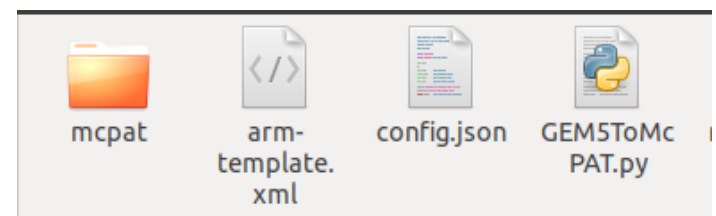
Step1: Install/Download MCPAT from <https://github.com/HewlettPackard/mcpat>

```
sandeepkiran@sandeepkiran-VirtualBox:~$ git clone https://github.com/HewlettPackard/mcpat
```

After successfully installing MCPAT,



Step2: After installing MCPAT, Set up a GEM5-MCPAT Parser, which has a XML template file that can read the stats.txt file generated by GEM5 and can produce a output which can be easily understood by the user



XML Template:

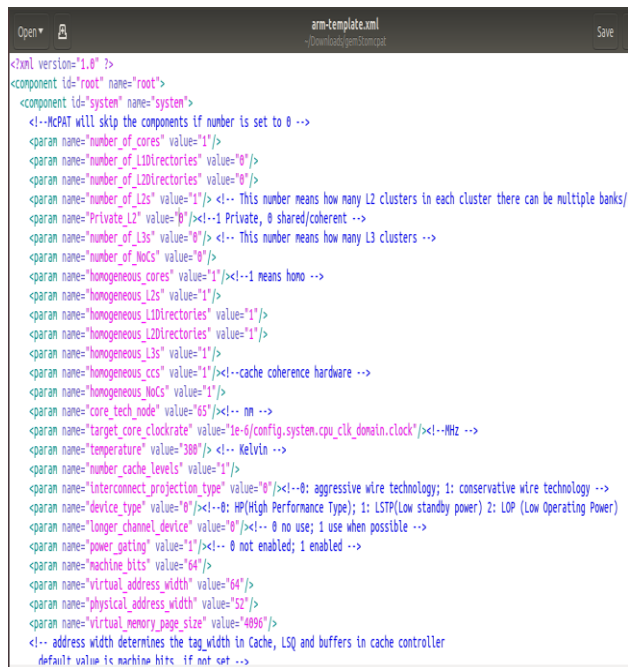
Set up the parameters

Parameters of target designs need to be set in the *.xml file for entries tagged as "param". McPAT have very detailed parameter settings.

Pass the statistics

There are two options to get the correct stats:

- The performance simulator can capture all the stats in detail and pass them to McPAT
- Performance simulator can only capture partial stats and pass them to McPAT, while McPAT can reason about the complete stats using the partial information and the configuration. Therefore, there are some overlap for the stats.



The above example uses a template file created by modifying ProcessorDescriptionFiles/Xeon.xml from McPAT. The parameters in arm-template.xml that should get their value from

GEM5 configuration (config.json) should be replaced with config.<parameter_path>.

For example

```
<param name="number_hardware_threads" value="2"/>
```

should be replaced by

```
<param name="number_hardware_threads" value="config.system.cpu.numThreads"/>
```

in the template file.

Any standard computation in python-supported format is allowed. For example, the following is acceptable:

```
<param name="target_core_clockrate" value="1e-6/config.system.cpu_clk_domain.clock"/>
```

Similarly, statistics that should use values from GEM5 statistics (stats.txt) should be replaced with stats.<stat_path>.

For example,

```
<stat name="total_cycles" value="100000"/>
```

should be replaced by

```
<stat name="total_cycles" value="stats.system.cpu.numCycles"/>
```

in the template file.

Usage of GEM5ToMcPAT.py:

Used for reading stats file of gem5, config file, template file and also for displaying the mcpat-out.xml file.

```
Open GEM5ToMcPAT.py
~/Downloads/gem5tomcpat

def readStatsFile(statsFile):
    global stats
    stats = {}
    if opts.verbose: print "Reading GEM5 stats from: %s" % statsFile
    F = open(statsFile)
    ignores = re.compile(r'^---[\^$]')
    statLine = re.compile(r'([a-zA-Z0-9_\.:-]+)\s+([+]?[0-9]+\.[0-9]+|[+]?[0-9]+|nan|ir)')
    count = 0
    for line in F:
        #ignore empty lines and lines starting with "---"
        if not ignores.match(line):
            count += 1
            statKind = statLine.match(line).group(1)
            statValue = statLine.match(line).group(2)
            if statValue == 'nan':
                print "\tWarning (stats): %s is nan. Setting it to 0" % statKind
                statValue = '0'
            stats[statKind] = statValue
    F.close()

def readConfigFile(configFile):
    global config
    if opts.verbose: print "Reading config from: %s" % configFile
    F = open(configFile)
    config = json.load(F)
    #print config
    #print config["system"]["membus"]
    #print config["system"]["cpu"][0]["clock"]
    F.close()

def readMcpatFile(templateFile):
```

Stats file for Non DVFS:

```
Open stats-dvfs.txt
~/Downloads/gem5tomcpat

----- Begin Simulation Statistics -----
sim_seconds          0.000029
sim_ticks            28858000
final_tick           28858000
checkpoints and never reset)
sim_freq              1000000000000
host_inst_rate        61093
host_op_rate          70859
host_tick_rate        87667380
host_mem_usage        2279580
host_seconds          0.33
sim_insts             20108
sim_ops               23324
system.voltage_domain.voltage 3.300000
system.clk_domain.clock 1000
system.mem_ctrls0.pwrStateResidencyTicks::UNDEFINED 28858000
system.mem_ctrls0.bytes_read::cpu_cluster.cpus0.inst 10560
system.mem_ctrls0.bytes_read::cpu_cluster.cpus0.data 4288
system.mem_ctrls0.bytes_read::cpu_cluster.cpus1.inst 10688
system.mem_ctrls0.bytes_read::cpu_cluster.cpus1.data 4288
system.mem_ctrls0.bytes_read::cpu_cluster.cpus2.inst 10688
system.mem_ctrls0.bytes_read::cpu_cluster.cpus2.data 4800
system.mem_ctrls0.bytes_read::cpu_cluster.cpus3.inst 9984
system.mem_ctrls0.bytes_read::cpu_cluster.cpus3.data 4608
system.mem_ctrls0.bytes_read::total 59904
system.mem_ctrls0.bytes_inst_read::cpu_cluster.cpus0.inst 16
memory
system.mem_ctrls0.bytes_inst_read::cpu_cluster.cpus1.inst 16
memory
```

Stats file for DVFS:

```
Open stats-dvfs.txt
~/Downloads/gem5tomcpat

----- Begin Simulation Statistics -----
sim_seconds          1.317973
sim_ticks            1317972609500
final_tick           1317972609500
checkpoints and never reset)
sim_freq              1000000000000
host_inst_rate        120369
host_op_rate          142348
host_tick_rate        150141942
host_mem_usage        2574912
host_seconds          8778.18
sim_insts             1056623743
sim_ops               1249556206
system.voltage_domain.voltage 1
system.clk_domain.clock 1000
system.bootmem.pwrStateResidencyTicks::UNDEFINED 1317972609500
system.bootmem.bytes_read::cpu_cluster.cpus0.inst 896
system.bootmem.bytes_read::cpu_cluster.cpus0.data 44
system.bootmem.bytes_read::cpu_cluster.cpus1.inst 704
system.bootmem.bytes_read::cpu_cluster.cpus1.data 16
system.bootmem.bytes_read::cpu_cluster.cpus2.inst 704
system.bootmem.bytes_read::cpu_cluster.cpus2.data 16
system.bootmem.bytes_read::cpu_cluster.cpus3.inst 704
system.bootmem.bytes_read::cpu_cluster.cpus3.data 16
system.bootmem.bytes_read::total 3100
system.bootmem.bytes_inst_read::cpu_cluster.cpus0.inst 896
system.bootmem.bytes_inst_read::cpu_cluster.cpus1.inst 704
system.bootmem.bytes_inst_read::cpu_cluster.cpus2.inst 704
system.bootmem.bytes_inst_read::cpu_cluster.cpus3.inst 704
```

Running GEM5-MCPAT PARSER:

GEM5 generated stats file, a processor configuration file produced by GEM5 and a McPAT template file. To run GEM5ToMcPAT.py use:

GEM5ToMcPAT.py stats.txt config.json template-xeon.xml

The above command will produce mcpat-out.xml file. mcpat-out.xml can be used in the usual way with McPAT.

<mcpat-bin> -infile mcpat-out.xml

```
sandeepkiran@sandeepkiran-VirtualBox:~/Downloads/gem5tomcpat$ ./GEM5ToMcPAT.py stats-nondvfs.txt
Reading GEM5 stats from: stats-nondvfs.txt
Warning (stats): system.mem_ctrls0.writeRowHitRate is nan. Setting it to 0
Warning (stats): system.mem_ctrls1.writeRowHitRate is nan. Setting it to 0
```

Now, displaying the XML File

```
Writing input to McPAT in: mcpat-out.xml
sandeepkiran@sandeepkiran-VirtualBox:~/Downloads/gen5toMcpat$ mcpat/mcpat -infile mcpat-out.xml
MCPAT (version 1.3 of Feb, 2015) is computing the target processor...

MCPAT (version 1.3 of Feb, 2015) results (current print level is 5)
*****
Technology 65 nm
Interconnect metal projection= aggressive interconnect technology projection
Core clock Rate(MHz) 2000

*****
Processor:
Area = 80.1683 mm^2
Peak Power = 67.734 W
Total Leakage = 20.8743 W
Peak Dynamic = 46.8597 W
Subthreshold Leakage = 20.0157 W
Subthreshold Leakage with power gating = 8.43875 W
Gate Leakage = 0.858594 W
Runtime Dynamic = inf W

Total Cores: 1 cores
Device Type= ITRS high performance device type
Area = 62.3363 mm^2
Peak Dynamic = 44.3457 W
Subthreshold Leakage = 17.6565 W
Subthreshold Leakage with power gating = 7.09542 W
Gate Leakage = 0.839711 W
Runtime Dynamic = inf W
```

```
sandeepkiran@sandeepkiran-VirtualBox:~/Downloads/gen5toMcpat$ mcpat/mcpat -infile mcpat-out.xml
MCPAT (version 1.3 of Feb, 2015) is computing the target processor...

MCPAT (version 1.3 of Feb, 2015) results (current print level is 5)
*****
Technology 65 nm
Interconnect metal projection= aggressive interconnect technology projection
Core clock Rate(MHz) 2000

*****
Processor:
Area = 80.1683 mm^2
Peak Power = 67.734 W
Total Leakage = 20.8743 W
Peak Dynamic = 46.8597 W
Subthreshold Leakage = 20.0157 W
Subthreshold Leakage with power gating = 8.43875 W
Gate Leakage = 0.858594 W
Runtime Dynamic = inf W

Total Cores: 1 cores
Device Type= ITRS high performance device type
Area = 62.3363 mm^2
Peak Dynamic = 44.3457 W
Subthreshold Leakage = 17.6565 W
Subthreshold Leakage with power gating = 7.09542 W
Gate Leakage = 0.839711 W
Runtime Dynamic = inf W

*****
Core:
Area = 62.3363 mm^2
Peak Dynamic = 44.3457 W
Subthreshold Leakage = 17.6565 W
Subthreshold Leakage with power gating = 7.09542 W
Gate Leakage = 0.839711 W
Runtime Dynamic = inf W

Instruction Fetch Unit:
Area = 12.8869 mm^2
Peak Dynamic = 6.36862 W
Subthreshold Leakage = 2.14624 W
Subthreshold Leakage with power gating = 0.885425 W
Gate Leakage = 0.0938309 W
Runtime Dynamic = -nan W

Instruction Cache:
Area = 3.41323 mm^2
Peak Dynamic = 1.50829 W
Subthreshold Leakage = 0.653302 W
Subthreshold Leakage with power gating = 0.273648 W
Gate Leakage = 0.026836 W
Runtime Dynamic = -nan W

Branch Target Buffer:
Area = 0.946863 mm^2
Peak Dynamic = 0.0843991 W
Subthreshold Leakage = 0.0901849 W
Subthreshold Leakage with power gating = 0.0485736 W
Gate Leakage = 0.00212441 W
Runtime Dynamic = -nan W

Branch Predictor:
Area = 0.279233 mm^2
Peak Dynamic = 0.0740148 W
Subthreshold Leakage = 0.0552507 W
Subthreshold Leakage with power gating = 0.0291987 W
Gate Leakage = 0.00144552 W
Runtime Dynamic = -nan W

Global Predictor:
Area = 0.0919946 mm^2
Peak Dynamic = 0.0191883 W
Subthreshold Leakage = 0.0197055 W
```



```

Local Predictor:
L1_Local Predictor:
  Area = 0.0545159 mm^2
  Peak Dynamic = 0.0163803 W
  Subthreshold Leakage = 0.0100013 W
  Subthreshold Leakage with power gating = 0.0100013 W
  Gate Leakage = 0.000280458 W
  Runtime Dynamic = -nan W

L2_Local Predictor:
  Area = 0.0279555 mm^2
  Peak Dynamic = 0.00834983 W
  Subthreshold Leakage = 0.0050708 W
  Subthreshold Leakage with power gating = 0.0050708 W
  Gate Leakage = 0.000147753 W
  Runtime Dynamic = -nan W

Chooser:
  Area = 0.0919946 mm^2
  Peak Dynamic = 0.0191883 W
  Subthreshold Leakage = 0.0197055 W
  Subthreshold Leakage with power gating = 0.0197055 W
  Gate Leakage = 0.000495847 W
  Runtime Dynamic = -nan W

RAS:
  Area = 0.0127724 mm^2
  Peak Dynamic = 0.0109081 W
  Subthreshold Leakage = 0.000767647 W
  Subthreshold Leakage with power gating = 0.000767647 W
  Gate Leakage = 2.56146e-05 W
  Runtime Dynamic = -nan W

Complex ALUs (Mul/Div) (Count: 1 ):
  Area = 0.336336 mm^2
  Peak Dynamic = 0.327493 W
  Subthreshold Leakage = 0.508016 W
  Subthreshold Leakage with power gating = 0.508016 W
  Gate Leakage = 0.0227379 W
  Runtime Dynamic = -nan W

Results Broadcast Bus:
  Area Overhead = 0.0986178 mm^2
  Peak Dynamic = 3.37793 W
  Subthreshold Leakage = 0.24116 W
  Subthreshold Leakage with power gating = 0.24116 W
  Gate Leakage = 0.0107939 W
  Runtime Dynamic = -nan W

*****
L2
  Area = 17.8319 mm^2
  Peak Dynamic = 2.51399 W
  Subthreshold Leakage = 2.35924 W
  Subthreshold Leakage with power gating = 1.99999 W
  Gate Leakage = 0.018883 W
  Runtime Dynamic = inf W
*****

```

After interpreting results of DVFS and Without DVFS , we plot graphs to see the performance of DVFS in ARM Architecture

For, each and every core if we run in McPAT , we can find the power consumption per core

Hotspot

Hotspot is a KDAB R&D project to create a standalone GUI for performance data. It is a replacement for perf report. Hotspot's GUI takes a perf.data file, parses and evaluates its contents and then displays the result in a graphical way.

Hotspot's initial goal was to provide a UI like KCachegrind around Linux perf. Its open source and the official [source code is available on GitHub page](#).

Implementation

First we use the perf profiler to record data. The recording is based on attaching to an existing process. The process ID of our gem5 simulator 31603. We use that and specify the events: cycles + instructions to profile. Then, perf records cycle-by-cycle hotspot symbols (the main running components within our process, that run on different threads, identified by their namespace), on the basis of their CPU time. Then, we use the Hotspot AppImage, a readily available container image of the hotspot perf to analyse the obtained samples. It uses the perf Callgraph to find out points of forking, joining and serial execution. There's also a support utility, FlameGraph, to estimate

The most important current features are listed below:

The summary view of hotspot gives a quick overview of the analyzed perf.data file and the system it was recorded on.

Top Hotspots	
Symbol	Binary
CoherentXBar::recvAtomic(Packet*, short)	gem5.opt
AtomicSimpleCPU::tick()	gem5.opt
ArmiISA::TLB::getResult(ArmiISA::TlbEntry**, std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType)	gem5.opt
ArmiISA::TLB::translateFs(std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType, bool&, bool, ArmiISA::Stats::StatStor::inc(double))	gem5.opt

System Information	
Host Name:	ubuntu
Linux Kernel Version:	4.15.0-20-generic
Perf Version:	4.15.17
CPU Description:	Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz
CPU ID:	GenuineIntel.6.61.4
CPU Architecture:	x86_64
CPU Online:	2
CPU Available:	2
CPU Sibling Cores:	[0], [1]

The perf.data file is now broken down into 5 symbols. Each, run by the gem5.opt simulator software- The AtomicCPU, the TLB lookup, Packet send and recv processes to and from the L1, L2 caches and a background Stats thread. The CPU-clock is the aggregation of sample costs attributed to that symbol which includes various function calls made by that process + self cost.

The bottom-up view displays the aggregated sample costs from where the cost is actually spent. This shows the hot functions in your code base.

perf.data	
File Settings Help	
Summary	Bottom Up
Search	
Symbol	
<ul style="list-style-type: none"> CoherentXBar::recvAtomic(Packet*, short) AtomicSimpleCPU::tick() ArmiISA::TLB::getResult(ArmiISA::TlbEntry**, std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType) ArmiISA::TLB::translateFs(std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType, bool&, bool, ArmiISA::Stats::StatStor::inc(double)) std::shared_ptr<FaultBase, (_gnu_cxx::Lock_policy)2>::shared_ptr(std::shared_ptr<FaultBase, (_gnu_cxx::Lock_policy)2> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType) ArmiISA::TLB::lookup(unsigned long, unsigned short, unsigned char, bool, bool, bool, bool) ArmiISA::TLB::getTE(ArmiISA::TlbEntry**, std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mode, BaseTLB::TranslationType) ArmiISA::TLB::updateMiscReg(ThreadContext*, ArmiISA::TLB::ArmTranslationType) ?? lib_malloc BaseSimpleCPU::preExecute() std::Hashtable<unsigned long, std::pair<unsigned long const, SnoopFilter::Snoopitem>*> ArmiISA::PCState::PCState(ArmiISA::PCState const&) AbstractMemory::access(Packet*) 	
Time Line	Filter
Thread	Events
gem5.opt (#...	

Here we see that, apart from the top5 symbols, a lot of running components are

being discovered as well. And each of them consume a very modest CPU time. Notably, the CPU, Cache coherent mechanism, the TLB lookups make it up for more than 70% of the entire processing lifetime of a gem5 simulator.

The top-down view of hotspot allows you to drill-down the call graph to find the most costly functions in your code base.

perf.data - Hotspot	
File Settings Help	
Summary	Bottom Up
Search	
Symbol	
<ul style="list-style-type: none"> ?? lib_malloc ArmiISAInst::LDRX64_IMM::~LDRX64_IMM() AtomicSimpleCPU::tick() GenericISA::BasicDecodeCache::decode(ArmiISA::Decoder*, BitfieldBackend::BitUnionOperators<ArmiISA::BitfieldBackend::BitUnionOperators>) BaseSimpleCPU::checkPcEventQueue() ArmiISAInst::ISA::readMiscReg(int, ThreadContext*) cfree AtomicSimpleCPU::readMem(unsigned long, unsigned char*, unsigned int, Flags<unsigned long>) Packet::Packet(std::shared_ptr<Request> const&, MemCmd) AtomicSimpleCPU::writeMem(unsigned char*, unsigned int, unsigned long, Flags<unsigned long>, unsigned long) ArmiISAInst::MicroStrImmUp::~MicroStrImmUp() operator new(unsigned long) Packet::~Packet() ArmiISAInst::MicroLdPairUp::~MicroLdPairUp() AtomicSimpleCPU::tryCompleteDrain() ArmiISA::ISA::readMiscRegNoEffect(int) const ProxThreadContext<SimpleThread>::ncState() 	
Time Line	Filter
Thread	Events
gem5.opt (#...	

The most costly function is the memory allocation task. It loads the kernel into the simulator's container instance.

Hotspot comes with a built-in flamegraph visualization. It is interactive and allows searching, zooming and more.

perf.data	
File Settings Help	
Summary	Bottom Up
Search	
cpu-clock:HG	Bottom-Up View
Collapse Recursion	Cost Threshold: 0.10%
1.77595e+10 aggregated sample costs in total	
1.77595e+10 aggregated sample costs in total	
Time Line	Filter
Thread	Events
gem5.opt (#...	

The colors on the flamegraph come from the CPU-clock HG costs, which represent the overall callgraph structure. The base of this city of towers, is the simulator instance that take 86.5% of CPU, on top of which 12-15 tasks branch out and run parallelly, on which, again run a few tasks, and this builds on uptil the top of the tower, with a single task. There are multiple peaks for every tower, branching out at different heights and the width of each bar and the color is mapped to the costs. It is used estimate the overall share of CPU time acquired by each of the processes.

Hotspot's Caller / Callee view aggregates the data into a main table view. This is useful to find unexpected costs of frames in the middle of the call stacks.

Summary Bottom Up Top Down Flame Graph Caller / Callee				
Search				
Symbol				
CoherentXBar::recvAtomic(Packet*, short)				
AtomicSimpleCPU::tick()				
ArmlISA::TLB::getResultTe(ArmlISA::TlbEntry**, std::shared_ptr<Request> const&, ThreadCo				
ArmlISA::TLB::translateFs(std::shared_ptr<Request> const&, ThreadContext*, BaseTLB::Mo				
Stats::StatStor::inc(double)				
std::shared_ptr<FaultBase, (_gnu_cxx::Lock_policy)2>::shared_ptr(std::shared_ptr<				
ArmlISA::TLB::checkPermissions64(ArmlISA::TlbEntry*, std::shared_ptr<Request> const&, B				
Stats::DistStor::sample(double, int)				
RefCountingPtr<StaticInst>::copy(StaticInst*)				
Caller	Binary	cpu-clock:HG	▲	Callee
??		10.9%		Clocked::clockPeriod(... gem!
CoherentXBar::recvAt...	gem5.opt	3.41%		CoherentXBar::recvAt... gem!
Stats::StatStor::inc(d...	gem5.opt	0.78%		SnoopFilter::updateR... gem!
Stats::Vector2dBase...	gem5.opt	0.287%		Stats::ScalarProxy<S... gem!
Stats::Vector2dBase...	gem5.opt	0.282%		Packet::isEviction() c... gem!
Stats::VectorProxy<S...	gem5.opt	0.117%		Stats::Vector2dBase... gem!
Stats::VectorBase<St...	gem5.opt	0.0887%		void Stats::ScalarPro... gem!
AddrRange::AddrRan...	gem5.opt	0.0887%		Stats::ScalarProxy<S... gem!
MemCmd::testCmdAt...	gem5.opt	0.0816%		RangeSize(unsigned l... gem!
Time Line Filter_ Search				
Thread Events				
gem5.opt (#...				

Again, clicking on a given symbol, leads to finding out all the possible caller and callees on that symbol. The top 5 symbols against branch out to most of the calls made in gem5.

An event time line which allows you to filter the results by time, thread or process

A new record page to configure and run perf graphically from within hotspot. It supports both, launching a new application as well as attaching to running processes

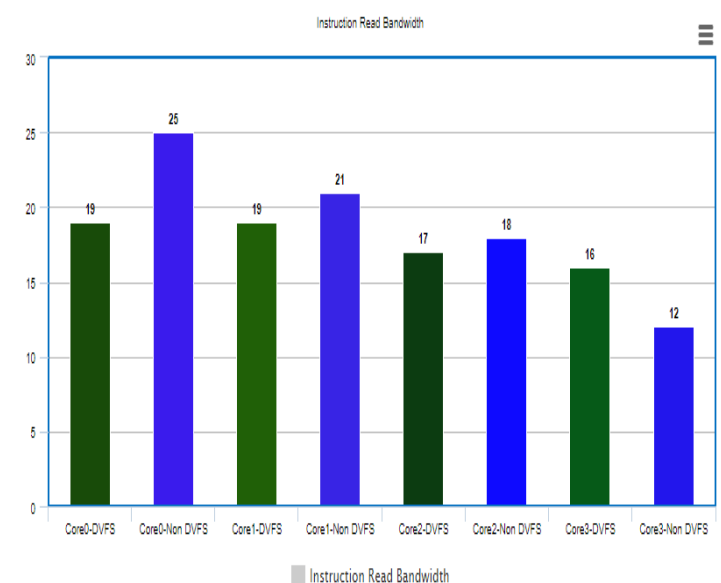
The analysis presented is from the hotspot AppImage from KDAB, Germany and not the Hotspot tool, from Virginia Tech that measures Thermal efficiency of processors from a Thermal floorplan. So, it was more of a profiler, except for the flame graph. But it is still based on the functions and code segments that make up for most CPU time. The expensiveness is purely CPU time. And it wasn't possible to run this tool within the simulated ARM, because, the ARM generic v8 linux is strictly shielded from any installation and the native packages are from the 2009 ARM archives.

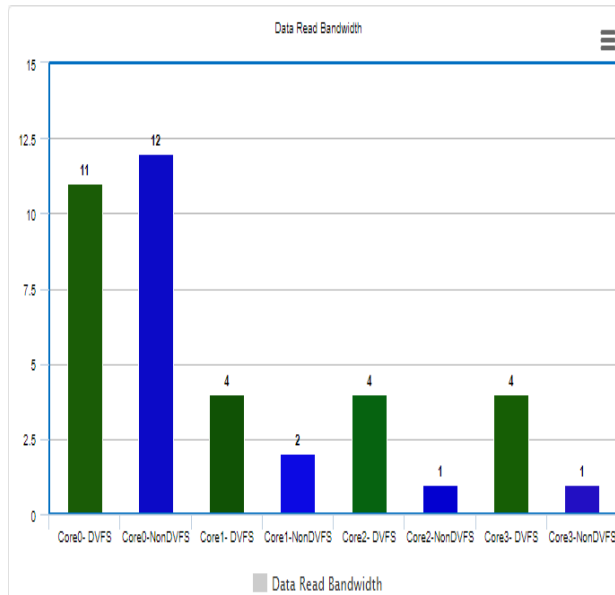
6.Results and Inferences

Stanford SingleSource Benchmarks

Holistic Comparison Between DVFS and NON-DVFS

Instruction and Data Read Bandwidth:

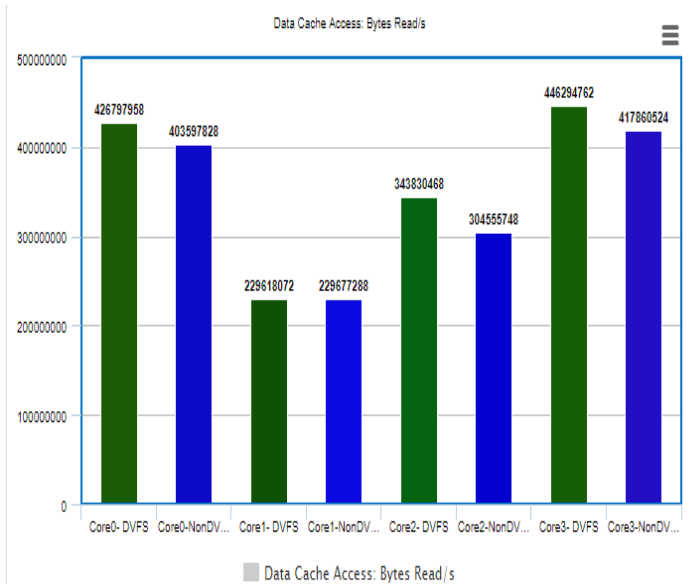
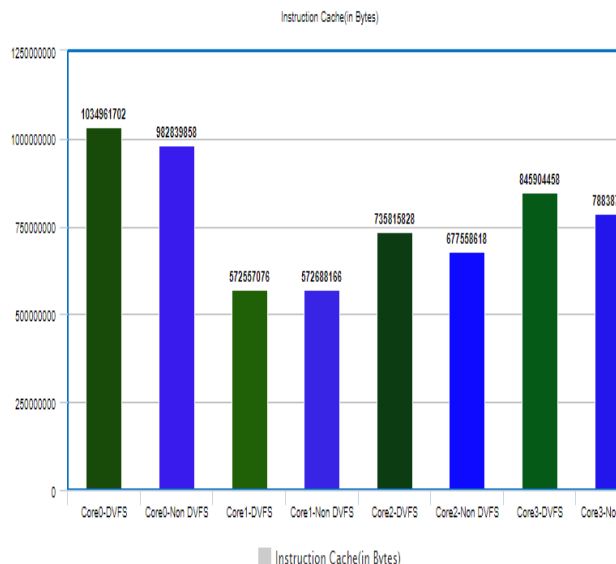




It's evident that DVFS has almost an uniform instruction read bandwidth for all the cores, so there is better utilization of the cores as the hypothesis is supposed to be. DVFS ameliorates dark silicon. This plot is testimonial in asserting that.

But in the case of Non-DVFS, there is a staggering difference in the instruction read bandwidth varying from 12 to 25 between the cores, which reveals the presence of huge amount of dark silicon, which obviously affects the performance of the system.

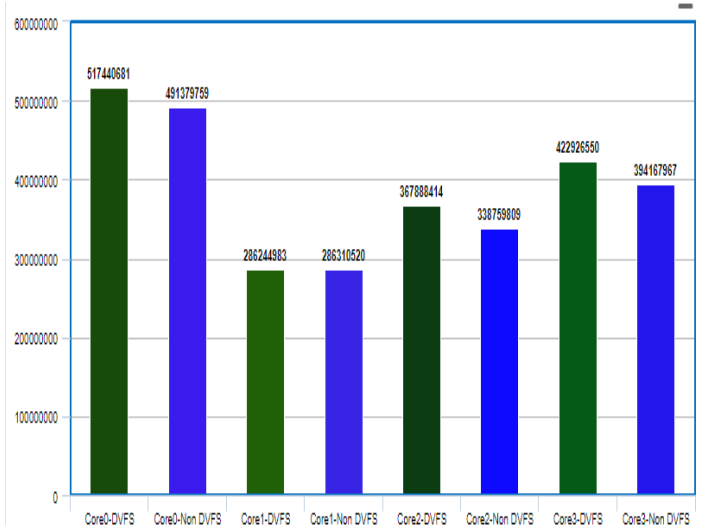
Instruction Cache and Data Cache Access: Bytes Read/s



For all the cores, DVFS Instruction Cache access is utilized to the maximum which accounts for the better bandwidth of the system as we come to see better latency and more hits and reduced misses the instructions are fetched from memory.

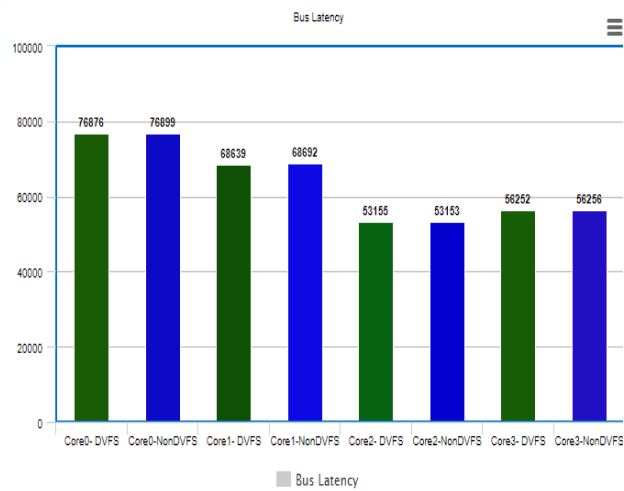
As for the Data Cache Access, peak number of bytes read per second is maximum in DVFS enabled ARM. Even the average and overall rate of reading data bytes from memory is higher in DVFS, for each core. This sheer advantage in reading and processing data quickly, at core level, forms the distinguishing feature of DVFS based systems. They process greater workload in short time.

Data Hits:



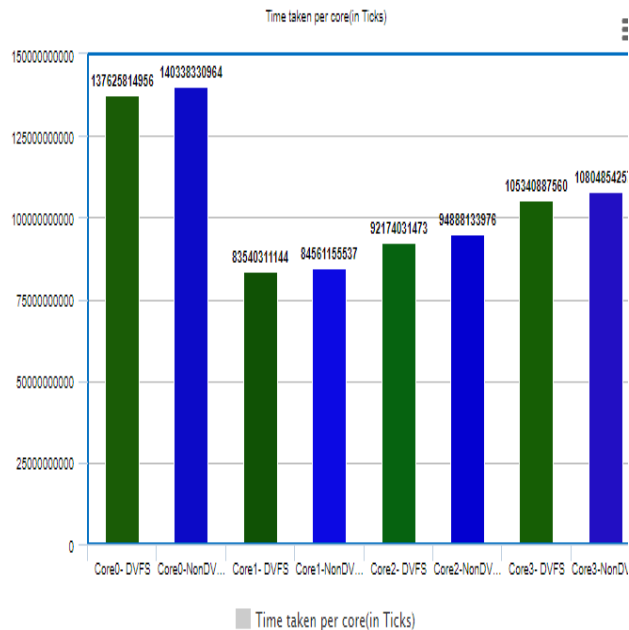
From the plot, DVFS has a better data hit ratio, since DVFS uses data cache to the maximum and this is in perfect sync with the plot of “Data Cache Access(in Bytes/s)”. The better hit ratio is indicative of the DVFS strategy overcoming the dark silicon problem.

Bus Latency



Latency is low in case of dvfs and because of that bandwidth, processing rate is high and this is consistent across all the cores

Time Taken per core(in Ticks):



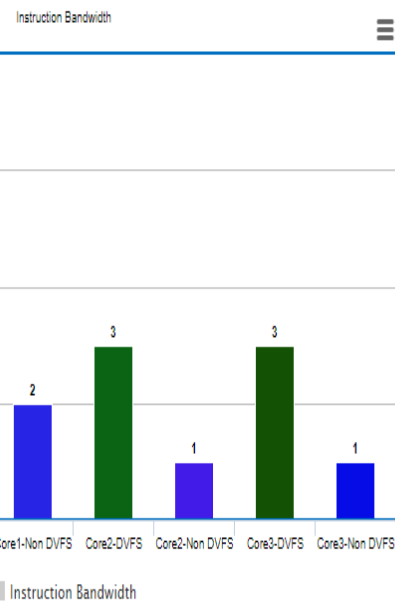
Processing the same workload at every core with DVFS, results in slightly reduced times than systems without it. This slight but similar processing times (at each core), is again an useful performance upgrade that is purely achieved by dynamically scaling operating frequency on every core.

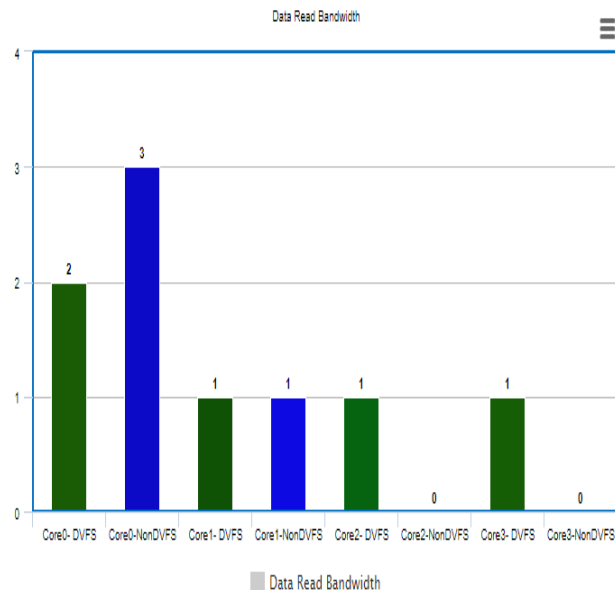
Now, these stats were recorded from running a subset of the light-weight Stanford SingleSource benchmarks + our custom superlight power_stats code for ARM. Next up, we present the comparison of a large parallel real-time data streaming app, Splash, that in contrast to a lot of small parallel tasks, is one single massively parallel task which forks out and caters to many CPUs.

The SPLASH Benchmark

Holistic Comparison Between DVFS and NON-DVFS

Instruction and Data Read Bandwidth:

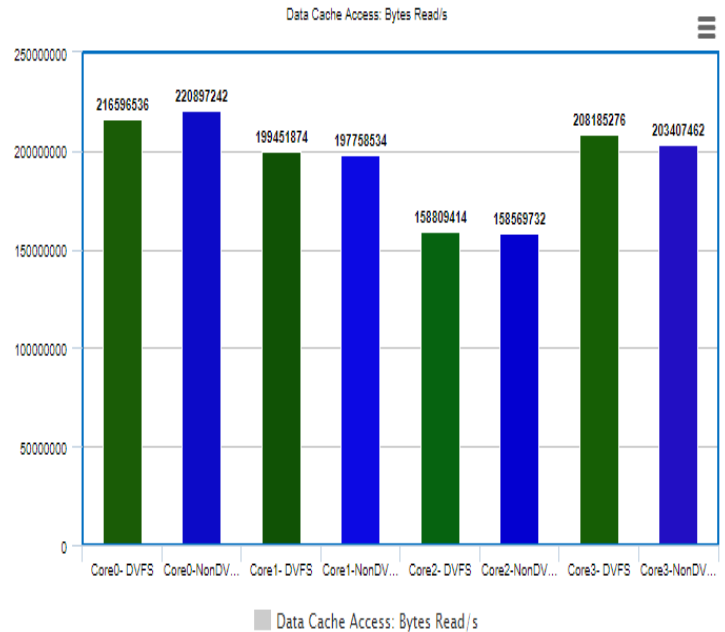
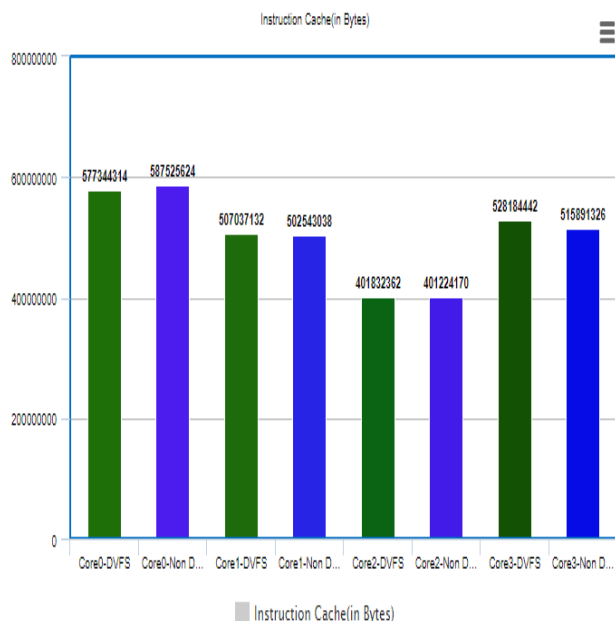




Again, the DVFS has almost a uniform instruction read bandwidth for all the cores which shows signs of better utilization. But in the case of Non-DVFS, the values fluctuate between 1 to 7 for I-cache and 0 to 3 for d-cache.

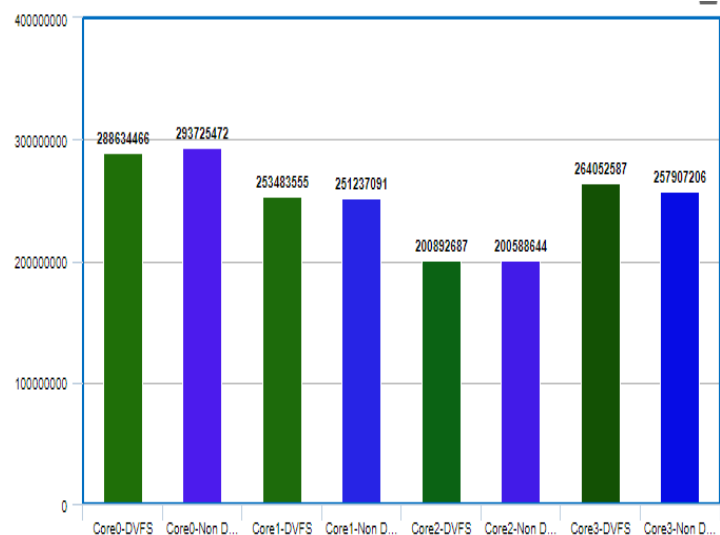
The last 2 cores, were almost non-functional, severely affected by Dark Silicon problem, whereas the DVFS puts these dark core transistors to better use. The comparison strikes coherence, as data is recorded between the 2 system architectures running the same workloads.

Instruction Cache and Data Cache Access:



Except for Core0, all the cores in DVFS use Instruction and Data Cache well, and the output just speaks for itself from the bandwidth results. Better the bandwidth, greater data you read at a time, greater you process and greater the throughput, lesser the latency.

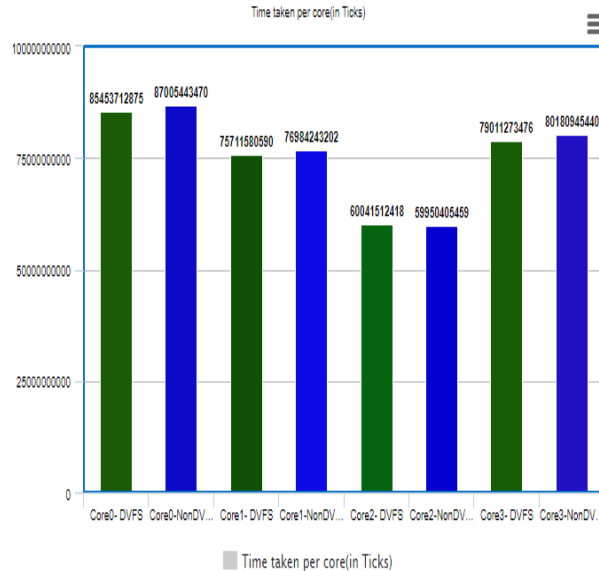
Data Hits:



DVFS has better data hits and processing speed is partly due to this fact as well, though this isn't supposed to be a consequence of frequency scaling. The access times and processing speeds

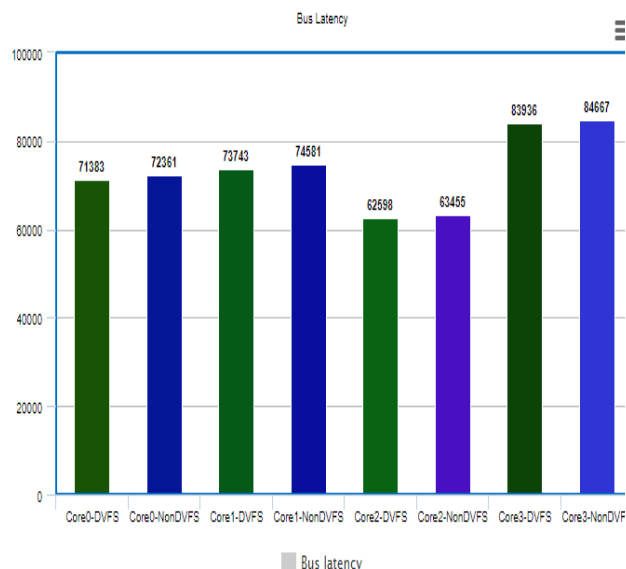
though, is a direct implication of scaling frequency.

Time taken per core (in Ticks):



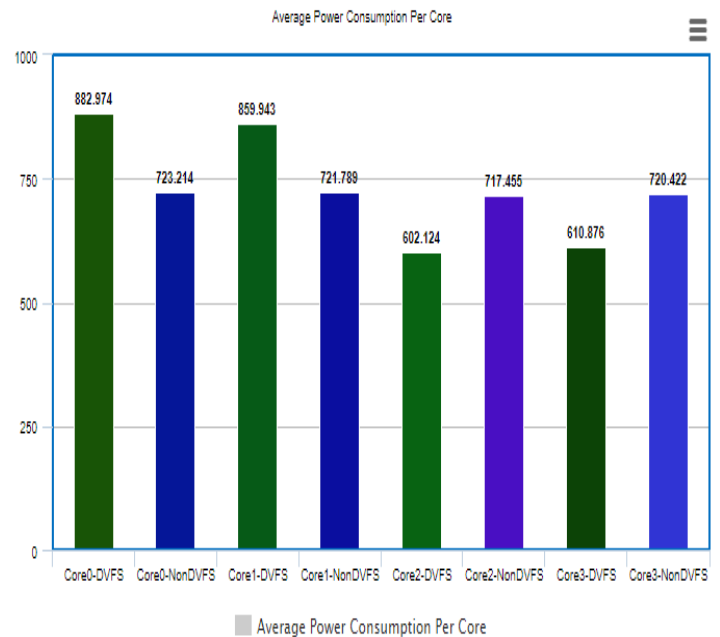
Again, the effect of power scaling has result in reduced processing times, as work now is intelligently managed, though the times are just a slight crossover.

Bus Latency:



Latency is low in case of dvfs and because of that bandwidth, processing rate is high and this is consistent across all the cores

Average Power Consumption



The DVFS has an overall uneven spread in power distribution across the cores. This is very much expected off a an offline processing entity that adjusts power in response to changes in workload, as the varying power consumption is coherent with the fundamental concepts discussed in the initial sections.

There is atleast a difference of 150 W for the first 2 cores and 100W for the next 2 cores, when compared to its non-DVFS counterpart.

Power Constraint and operating within the power budget is an important constraint at optimizing dark silicon . The power consumption per core is now suitably managed to avoid overheating. Thanks to the controller attached to the system bus. All the cores run at a lowered frequency, still aid to overall throughput. What follows is, that more tasks are being processed in less time and there are less idle cores .

7. Conclusion and Future Work

In this work, we proposed and evaluated memory voltage/ frequency scaling in order to reduce memory power and increase energy efficiency. Beginning with a note on Moore's law and its implications, followed by the shortcomings of Dennard scaling, we discussed the possibility of Dynamic Voltage Scaling as an alternative to Static Active Power Management in multi-core. Then the argument was advanced on theoretical front by presenting concrete facts about how DVFS might actually overcome dark silicon. Finally after the Full System simulation in Gem5, power modelling in MCPAT and thermal flamegraph in Hotspot, we observe that, DVFS in fact responds to changes in workloads. It means that the voltage is automatically scaled to match the needs of parallelism at a point in time. This was empirically verified with the plots from data gathered out of McPAT.

Atlast, we rate the DVFS model on various terms, including a comprehensive per core analysis for each metric considered (as this happens to be a heterogeneous hardware design with the DVFS controller in place), the key being the memory footprint. This forms the scope for future work. That is, to tweak the DVFS Handler source code implemented by gem5.opt to leverage the knowledge of this research work undertaking, by utilizing the performance analysis.

By observing the memory read, write bandwidth utilization, the bus latency, data access, hits, our proposed model for overcoming dark silicon, is proven to react for both when the workload is a large set of light-weight parallel codes, and for a single large parallel code.

We conclude that DVFS can be an effective energy efficiency technique to exploit unused silicon, especially when the need to generate better throughput.

8. References

<http://learning.gem5.org/book/>

<http://www.gem5.org/Tutorials>

<http://users.ece.utexas.edu/~ljohn/teaching/382m-15/lectures/lec16.pdf>

<https://github.com/KDAB/hotspot>