On Understanding the Energy Consumption of ARM-based Multicore Servers

Bogdan Marius Tudor and Yong Meng Teo
Department of Computer Science
National University of Singapore
13 Computing Drive, 117417
{bogdan,teoym}@comp.nus.edu.sg

ABSTRACT

There is growing interest to replace traditional servers with low-power multicore systems such as ARM Cortex-A9. However, such systems are typically provisioned for mobile applications that have lower memory and I/O requirements than server application. Thus, the impact and extent of the imbalance between application and system resources in exploiting energy efficient execution of server workloads is unclear. This paper proposes a trace-driven analytical model for understanding the energy performance of server workloads on ARM Cortex-A9 multicore systems. Key to our approach is the modeling of the degrees of CPU core, memory and I/O resource overlap, and in estimating the number of cores and clock frequency that optimizes energy performance without compromising execution time. Since energy usage is the product of utilized power and execution time, the model first estimates the execution time of a program. CPU time, which accounts for both cores and memory response time, is modeled as an M/G/1 queuing system. Workload characterization of high performance computing, web hosting and financial computing applications shows that bursty memory traffic fits a Pareto distribution, and non-bursty memory traffic is exponentially distributed. Our analysis using these server workloads reveals that not all server workloads might benefit from higher number of cores or clock frequencies. Applying our model, we predict the configurations that increase energy efficiency by 10% without turning off cores, and up to one third with shutting down unutilized cores. For memory-bounded programs, we show that the limited memory bandwidth might increase both execution time and energy usage, to the point where energy cost might be higher than on a typical x64 multicore system. Lastly, we show that increasing memory and I/O bandwidth can improve both the execution time and the energy usage of server workloads on ARM Cortex-A9 systems.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Performance Attributes*; C.5.5 [**Computer System Implementation**]: Servers

Keywords

Analytical Model, Performance, Energy, Low-Power, Multicore, Servers

SIGMETRICS'13, June 17-21, 2013, Pittsburgh, PA, USA.

1. INTRODUCTION

A by-product of the smartphone revolution is the sustained improvement in performance capabilities of low-power multicore systems. In the past three years, the clock frequencies and core counts of commodity ARM processors have pronouncedly increased, to the point that their performance is within the range of traditional x64 systems. Furthermore, ARM offers comparable core counts and clock frequencies at a fraction of the energy cost of traditional multicore systems. Due to their attractive power-efficiency, price and density, many companies and research projects have started to migrate towards servers based on low-power multicores. Barcelona Supercomputing Center is developing a supercomputer based on low-power ARM Cortex-A9 systems. Nvidia, Dell and HP [2] are developing systems based on quad-core Cortex-A9 while companies such as Calxeda are developing ARM-based systems which explicitly target the server landscape [1]. In particular, these multicore systems are very attractive for datacenters, as they usher in a new generation of green computing with much lower energy budgeted for cooling or lost during core idling, than what is possible using traditional Intel/AMD x64 or Intel Atom systems.

There has always been a trade-off between power-efficiency and performance of processing systems. Many currently-available ARM Cortex-A9 systems have been configured mostly for mobile computing devices such as phones or tablets, and thus their resources are sized for the balance between cores, memory and I/O required by mobile applications. As part of this balance, they are provisioned with lower *achievable memory bandwidth* than traditional x64 server systems. For example, the memory-level parallelism in Cortex-A9 chips is limited to two outstanding memory requests [33], as compared to ten in Intel chips [34]. Because mobile apps are typically not memory-bounded, the size of the caches range between 256 kB and 1 MB in the commodity Cortex-A9 systems shipped by vendors such as Samsung, Nvidia, Texas Instruments or ST-Ericsson. In contrast, the size of the cache memory in most x64 server systems exceeds 10 MB. Furthermore, the memory subsystem in many currently available ARM Cortex-A9 chips use low power memory, with 32-bit data bus width and operating at lower clock frequencies compared with traditional memory chips. As a result of these factors, low-power computing on ARM Cortex-A9 might suffer from a larger imbalance between arithmetic and memory performance than traditional x64 systems. Considering that many types of server workloads are I/O or memory-intensive, the large gap between core and memory performance might lead to unexpected results.

Previous studies on energy efficiency in server systems have addressed the impact of the number of active versus idle systems in a cluster of servers [14, 15, 31, 38]. A well known problem in datacenter systems is low system utilization which contributes to

significant energy wastage [4, 13, 15]. Additionally, the problem of selecting the optimal number of cores in a multicore system has previously been addressed from the perspective of selecting the optimal performance of area-equivalent cores that, when replicated across the entire die, offers the best system-wide throughput. Most of the previous work suggests that many cores with low individual performance (termed *wimpy cores*) may offer better system-wide throughput than the area-equivalent high performance cores (termed *brawny cores*) [12, 29], if two considerations are met. First, the workload has enough parallelism to sustain execution on many cores [17]. Second, the relative performance between wimpy to brawny nodes does not impact the overall cluster cost, programability and schedulability of the parallel tasks [18]. However, to the best of our knowledge, the problem of understanding the energy impact of the imbalance between the cores, and the memory and I/O resources has not been previously addressed. With the growth in core counts, as well as the large amount of power required just to keep the core idle, it is important to understand the extent to which the active number of cores influences the performance and power efficiency of server workloads. Furthermore, many contemporary systems have the ability to selectively power-off some of the CPU cores. However, these power states have long setup times, and thus, many users do not use them, due to the risk of not being able to power-on in time, when utilization increases. Such users would benefit from predicting exactly what is the impact of the number of cores on the performance of a particular job. If the user is satisfied that the job does not need all the available cores, it can leverage on the processor sleep states to achieve important energy savings [23].

In this paper, we propose a hybrid measurement-analytical approach to characterize the energy performance of server workloads on state-of-the-art low power multicores. Firstly, we introduce a general analytical model for predicting execution time and energy used in parallel applications, as a factor of number of active cores and core clock frequency. The key idea behind the model is to characterize the overlap between the response times of three key types of resources: processor cores, memory and network I/O resources. Using a simple queueing model, we account the overlap between CPU work cycles, CPU memory cycles and network I/O execution time, and identify the system bottleneck. Furthermore, we model the impact of changing the number of cores on each type of response time. To apply this general model, we perform a series of baseline runs of an application, during which we collect traces of the total number of cycles, total stall cycles, total last level cache misses, the time-distribution of the last level cache misses and I/O requests profile. Using a static power profile of the system and these collected metrics, we can predict the execution time and energy usage of an application for different number of cores and clock frequencies, and thus, we can select the configuration that maximizes performance without wasting energy. We validate the model against direct measurement of execution time and energy on a low-power server based on a quad-core Exynos 4412 ARM Cortex-A9 multicore processor. Validation on three types of parallel applications spanning high-performance computing, web-hosting and financial computing indicates that the relative error between predictions of our model averages 9%, with 70% of the experiments falling under 5% error.

The second contribution of our model is an analysis of execution performance and energy usage for a series of workloads covering high performance computing, web-hosting and financial applications. Firstly, using predictions of our model, we show that comparatively few types of workloads benefit from executing on large core counts. *Memcached*, a popular in-memory key-value store widely used by web giants such as Amazon, Facebook and Twitter, among others, achieves peak performance using just two cores and slightly more than half of the maximum core frequency. Keeping more than two cores powered-on, or allowing the clock to grow to maximum frequency, does not improve the performance even under continuous demand. High performance computing applications, which require large amounts of memory bandwidth, achieve even poorer results. For example, our prediction shows that pentadiagonal matrix solver *SP* from NASA Parallel Benchmarks suite, finishes in around four hours for a grid of size 162^3. The same program finishes in under seven minutes on a dual Intel Xeon X5650 processor. Due to this large difference in execution times, *SP* with input C may consume less energy on a traditional Intel multicore than on an ARM low-power multicore. Secondly, we show that our model can predict configurations of core counts and clock frequencies that may reduce the energy incurred by one third, without significantly extending the execution time of the applications. Thirdly, we show that that server workloads can achieve better energy proportionality on low-power multicores if the performance of the memory and I/O subsystems is improved, as expected in the ARM Cortex-A15 and the upcoming 64-bit ARM Cortex-A50 families.

The rest of the paper is structured as follows. Section 2 introduces our model for energy performance. Section 3 discusses the model input parameters and the validation results against direct measurements. In Section 4, we analyze of execution performance of server workloads in low-power multicore. Related work is discussed in Section 5, and section 6 concludes the paper.

2. PROPOSED MODEL

This section describes our proposed analytical model for energy consumption. We introduce first the system assumptions and model parameters. Second, we derive our model for execution time and energy usage.

2.1 Assumptions and Model Parameters

We consider the system-wide energy used by a shared-memory parallel application. We assume the application is executing in isolation, with negligible interference from other programs or background operating system tasks. The system is composed of one low-power processor with n cores, operating at clock frequency f, where $f \in [f_{min}, f_{max}]$, based on dynamic voltage and frequency scaling. We model superscalar and out-of-order cores, where an integer operation, an floating point operation, and a memory requests can be issued during each clock cycle. The system has one memory controller connected by a memory bus, shared equally between all cores and one wired network interface. All I/O operations involve the network, thus we do not consider storage I/O. The network device is memory-mapped, and read/write operations are performed using interrupt-driven I/O using direct memory accesses (DMA) that incur negligible CPU cycles [9].

The system-wide energy used by the system is split into three categories: cores, memory and network device. During the program execution, the cores are considered to be active (i.e. in C-state 0), even when idling because of lack of workload. The cores can change frequency (i.e. change the P-state), and thus the power drawn depends on the core frequency. Furthermore, within a P-state, we consider that power drawn depends on the type of workload (i.e. integers, floating points or stalls) executed by the core. The memory is considered to have two power-states, depending if it is issuing memory requests versus just refreshing its contents. Similarly, the network device is considered active when it is sending or receiving data, and passive when idle.

The parallel applications studied are composed of t homogeneous worker threads. If the applications have threads that do not

significantly participate in the work, such as threads active only during short initialization and finalization period, we do not count them in the t worker threads. We consider that the system has enough capacity to run all worker threads concurrently, $t \leq n$. We do not consider scenarios with more threads than cores, because the overhead of context switching among threads is significant on ARM systems [11, 33], and thus, such scenarios are unlikely to occur in practice. Table 1 summarizes the model parameters.

Symbol	Description
General parameters	
n	Number of cores
f	Clock frequency
Workload parameters	
c	Total cycles incurred by program
w	Work cycles executed by program
m	Stall cycles due to contention
b	Stall cycles not due to contention
r	Last level cache misses
$\bar{r_j}$	Average size of one memory burst
λ_M	Arrival rate of memory requests
$\lambda_{I/O}$	Arrival rate of I/O requests
System parameters	
s_M	Service time of one memory request
$P_{CPU,idle}$	Idle CPU power consumption
P_{WORK}	Power of CPU work cycles
P_{STALL}	Power of CPU stall cycles
$P_{M,idle}$	Memory power when idle
$P_{M,active}$	Memory power under load
$P_{I/O,idle}$	I/O power when idle
$P_{I/O,active}$	I/O power under load
Time performance	
C	CPU response time
T_M	Response time of all memory requests
I	I/O response time
I_T	I/O service time
U_{CPU}	CPU utilization
T	Execution time of a program
Energy performance	
E_{CPU}	Energy used by processor
E_M	Energy used by memory
$E_{I/O}$	Energy used by I/O device
E	Total energy of a program

Table 1: Table of Notations

2.2 Model Overview

In this section we introduce our trace-driven model for optimal core-frequency configuration that reduces execution time without wasting energy. The approach is to derive the execution time and energy used by a system while running a parallel program, as a factor of number of cores, n, and core clock frequency, f. To achieve this, first we model the impact of three key resources on the execution time of the program: cores, memory and I/O device. Next, based on the service times and the utilization of each resource, we propose a model for the energy usage of the program.

In our model, a server application consists of workloads that are serviced by three types of resources: n CPU cores, memory and I/O device. However, the response times required at these resources can overlap in time, and thus, the response time of the program cannot be established by simply adding the service and waiting time on all the resources. Modern server systems, including ones based on low-power processors, have I/O devices that can send and receive data without intervention from the CPU cores. They do so because the device is memory mapped, and all data transferred between the device is marshalled by a special processor called a DMA controller. Therefore, I/O device response time can be overlapped with the CPU response time. Furthermore, CPU cores have deeply pipelined out-of-order executions that overlaps the execution of arithmetic operations with waiting for memory requests. Thus, in a multicore system, the arithmetic instructions executed by multiple cores in parallel are overlapped with waiting for outstanding memory requests and I/O response time on the I/O device. However, from a measurement point of view, not all active and idle times are independent. A CPU core is seen as active both during execution of arithmetic operations, and while waiting for memory requests [16]. As a result, the *CPU time* of a program effectively accounts for both memory response time and service time of arithmetic instructions. Based on the overlap between processor and

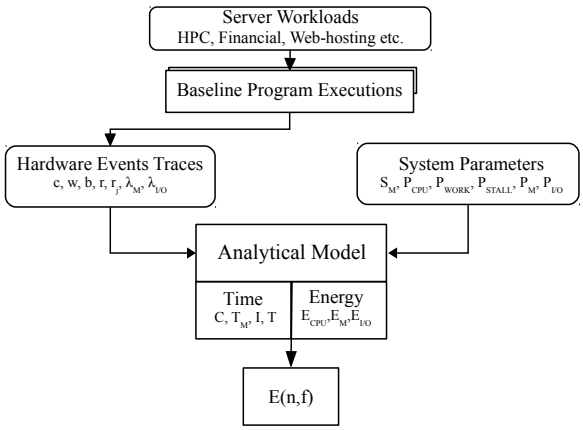

Figure 1: Approach for Applying Proposed Model

I/O resources, we define two response times in the system:

1. *CPU response time* (C) – total time during which a core is executing instructions or waiting for memory requests, for all cores;
2. *I/O response time* (I) – total time during which any core waits for the I/O device.

In this paper we target parallel server workloads that consist of multiple iterations of similar compute and I/O phases. Therefore, *either CPU response time or I/O response time dominates the execution time, and the other is completely overlapped*. First we use a bottleneck analysis technique to model the impact of CPU time and I/O time on the execution time of the entire program. Considering n cores, and that CPU time can be parallelized among these cores, while I/O time cannot be parallelized, the execution time of a program is:

$$T = max\Big(\frac{C}{n}, I\Big) \tag{1}$$

while the CPU utilization averaged over the entire execution of the program is:

$$U_{CPU} = \frac{C}{nT} \tag{2}$$

Figure 1 presents our trace-driven approach for applying the model to predict the energy performance of a program. We perform two baseline runs of the program, during which we collect traces the processor hardware events and I/O requests. During the first run, we execute the program on on core, alternating between the highest and lowest core frequency. Using the hardware events counters, we measure the total cycles, stall cycles, and last level cache miss profile. Using these values, we predict the CPU time as a factor of number of cores n and core clock frequency f. During a second run, we measure the I/O requests sizes and the measured inter-arrival time of the I/O requests, thus, predicting the I/O time I. Next, using a static power profile of the system and the service times for each resource, we predict the energy consumption for any configuration of number of cores and core clock frequency.

2.3 Execution Time Model

This section introduces our proposed model for deriving the CPU response time, C, and I/O response time, I.

2.3.1 CPU Response Time

The CPU time model captures the impact of the number of cores and core clock frequency on the CPU time of the program. Our model distinguishes the CPU cycles incurred due to memory contention versus clock cycles unrelated to memory contention. State-of-the art low-power processors such as ARM consist of several cores that share important off-chip resources, such as the memory bus and the main memory banks. Due to resource sharing, a well-known performance issue in multicore systems is the *memory contention among cores* [27]. When a thread suffers from memory contention among cores, each off-chip memory access takes a longer time to complete, compared to the uncontended case. When the contention is high, the thread might not have enough useful work to overlap over the memory request response time, and the thread incurs stall cycles.

Because measurement tools report CPU time as cycles, our model for CPU time uses cycles as the unit of time. Thus, all response times of core and memory requests are accounted in the CPU time based on the clock frequency and the number of cycles. If $C(n, f)$ is the CPU time of a program, and $c(n, f)$ is the total number of cycles incurred across all cores, then:

$$C(n, f) = \frac{c(n, f)}{f} \tag{3}$$

We consider the cycles incurred by a program as belonging to three categories:

- *Work cycles* (w) – the CPU cycles during which at least one integer operation, one floating point operation or one memory request is issued.
- *Stall cycles not due to memory contention* (b) – stalls caused by the latency of cache hits, pipeline bubbles or contention between the memory requests issued by the *same* core.
- *Stall cycles caused by memory contention* (m) – the stall cycles incurred because more than one core are competing for off-chip memory bandwidth.

Let $w+b$ denote the total work of the program that is independent on memory contention. In general w can be considered the parallel useful work of the program, while m as the overhead caused by memory contention. Notice that we do not consider all stall cycles as being overhead, since the focus of this model is to analyze the overhead caused by changing the number of cores on the CPU time incurred by a program. Thus, stall cycles that are independent on the memory contention among cores are treated by our model as a useful work.

Next we model the total number of cycles incurred by a program, $c(n, f)$. ARM Cortex-A9 processors have pipelined cores that execute out-of-order instructions. Thus, the core can overlap part of the time required to retrieve the data from memory, with execution of instructions for which the data is available. Because of the out-of-order execution, the CPU time incurred by a thread can be divided into a series of *instruction windows* (i.e. discrete compute time epoch). During each window, the core executes the instructions for which the operands are available, and at the same time issues memory requests for the data of the next instruction window. The next instruction window commences as soon as the data begins to arrive. If the data needs to be fetched from the main memory, the core might stall for many cycles. When several cores are issuing memory requests at the same time, the memory requests' response time experienced by a core increases due to the service time of memory requests performed concurrently by other cores. Due to this overlap between computation and memory requests, the execution time of any compute epoch in a thread depends on two possible bottlenecks. For low core frequencies, or compute episodes with small number of main memory accesses, the core is the bottleneck and the execution time is dominated by the time required to execute the useful work $w + b$. In contrast, when the compute episode requires many main memory accesses, the execution time of the episode is dominated by the memory response time.

Let $w_j + b_j$ denote useful work cycles and m_j denote the memory contention cycles executed during one instruction window. The response time of the entire episode is:

$$c_j(n, f) = \begin{cases} w_j + b_j & \text{if core is bottleneck} \\ m_j & \text{if memory is bottleneck} \end{cases} \tag{4}$$

While the processor core is waiting for the data to be fetched from memory, it is incurring stalled cycles. Let $T_{M,j}$ be the response time for the memory requests issued by a core during instruction window j, then the stall cycles incurred by a core while waiting for the data to arrive from main memory is:

$$m_j(n, f) = T_{M,j}(n) \cdot f \tag{5}$$

The number of cycles incurred by the entire program execution is:

$$c(n, f) = \sum_j c_j(n, f) \tag{6}$$

Since the bottleneck device dominates the response time:

$$c_j(n, f) = max(w_j + b_j, f \cdot T_{M,j}) \tag{7}$$

$$c(n, f) \approx max(\sum_j w_j + b_j, f \sum_j T_{M,j}) \tag{8}$$

To infer the CPU time of a program, we model w_j+b_j and $T_{M,j}(n)$.

The useful work of the program consists of cycles executing the arithmetic instructions w_j and stall cycles unrelated to memory contention b_j. For fixed-sized workloads, these values do not change when the number of cores change [37]. The useful work of the program does not depend on the number of cores. The work cycles w_j depends only on the availability of the instruction operands. If the number of cache misses does not change when changing the number of cores, then the total cycles required to execute the episode does not change. Similarly, since b_j model back-end pipeline stalls due to register contention, branch mispredictions and latency of cache hits, it does not matter how many cores split the stalls, as the total number remains constant:

$$w + b \approx \sum_j w_j + b_j \tag{9}$$

We use a queueing model to predict the response time for the memory requests $T_{M,j}(n)$, a factor of number of cores performing concurrent memory requests, n. ARM Cortex-A9 memory subsystem is a uniform memory access (UMA) architecture. All n cores equally share the bus between the memory controller and the processor. For simplicity, we consider the entire memory subsystem, consisting of a memory bus, memory controller and memory banks as a single server. The number of memory requests in the system is bounded by the memory-level parallelism of the cores, and as such, we consider the entire system a closed system model.

The memory requests that are send to the memory server are filtered by two levels of cache, and since the programs analyzed in this paper consist of server workloads, we assume that there is sufficient time between memory requests arrivals to satisfy a memorylessness property of the memory requests inter-arrival time. However, we do not make any assumptions on the size of the memory

requests, and thus, on the distribution of service times required by the memory requests. $T_{M,j}$ is composed of service time $S_{M,j}$ and waiting time $Z_{M,j}$ of the memory requests.

$$T_{M,j} = S_{M,j} + Z_{M,j} \tag{10}$$

Let r be the total number of last level cache misses, $\bar{r_j}$ the average number of last level cache misses requested during one instruction window, and $Var(r_j)$ the variance of last level cache misses requested during one instruction window. The total number of instruction windows throughout the execution of the program is j:

$$j = \frac{r}{\bar{r_j}} \tag{11}$$

When s_M is the service time required by one memory request, then the average and variance of the service time required by all r requests are:

$$\begin{aligned} \bar{S_{M,j}} &= s_M \cdot \bar{r} \\ Var(S_{M,j}) &= s_M^2 \cdot Var(r) \end{aligned}$$

Let λ_M be the arrival rate of memory request from a single core. If there are n active cores that are issuing memory request, the total arrival rate of memory requests is:

$$\lambda = n \cdot \lambda_M \tag{12}$$

The response time of the r memory requests is modeled using a M/G/1 queueing system. From Pollaczek-Khinchin formula [36]:

$$S_{M,j}(n) = \bar{r_j} s_M \tag{13}$$

$$Z_{M,j}(n) = \bar{S_{M,j}}^2 \lambda \frac{1 + Var(S_{M,j})}{2(1 - \bar{S_{M,j}}\lambda)} \tag{14}$$

From equations 11 to 14:

$$T_{M,j}(n) = \bar{r} s_M + \bar{r_j}^2 s_M^2 n \lambda_M \frac{1 + s_M^2 Var(r_j)}{2(1 - \bar{r_j} s_M n \lambda_M)} \tag{15}$$

$$T_M = j \cdot T_{M,j} = r s_M \Big(1 + s_M \bar{r_j} n \lambda_M \frac{1 + s_M^2 Var(r_j)}{2(1 - s_M \bar{r_j} n \lambda_M)}\Big) \tag{16}$$

Equation 16 shows that the response time of the memory requests degrades significantly with an increase in n. Furthermore, the increase in memory response time also depends on the burstiness of memory traffic. Equation 16 also describes how the workload interacts with the machine: λ_M and r depend on the workload, while s_M is a system parameter, which depends on the bandwidth of the memory system.

To apply the model, we determine the response time of one memory request as the ratio of cache line size and the effective memory bandwidth. The average memory burst size r_j is determined based on the probability profile of the burst size. In the model parameterization section, we show that server workloads fall under two categories, bursty memory traffic and non-bursty memory traffic. For bursty memory traffic we use a Pareto distribution to model $\bar{r_j}$ and $Var(r_j)$, based on inputs collected during the baseline runs. For non-bursty memory traffic, we use an exponential distribution of burst size, and reduce the M/D/1 model to an M/M/1 model. Finally, r is determined from the trace of the hardware events counters.

2.3.2 I/O Response Time

The server applications targetted involve network I/O requests operating based on a *request-reply* pattern. In general, many types of web-hosting workloads are governed by this pattern [36]. For example, a webserver receives an HTTP request on the I/O interface, forms a reply by performing some computations and then sends the reply back to the sender.

The typical mechanism employed by a program to receive data from a I/O device involves performing a system call on a network socket. To service the requests, a thread performs a system call (on Linux, typically `read` or `recvmsg`) instructing the operating system to read the content of the request from the device. If the system does not receive any data on the device, the system call blocks the calling thread until the request can be completed. When the I/O device receives the request data, the operating system copies the data to the main memory using direct memory access (DMA), and then unblocks the thread from the system call. After the reply is formed, the thread performs another system call (typically `write` or `sendmsg`) that instructs the operating system to send a reply data to the I/O device. Thus, response time of an I/O operation can be divided into:

1. I/O blocking time (I_B) – total time between the thread blocking on a read system call and the time moment when the data arrives from the sender to the I/O device, for all read system calls.

2. I/O transfer time (I_T) – total time required to transfer the data between the I/O device and the main memory.

In contrast to the read operation, the write requests do not incur blocking time until the data arrives to the destination, because this aspect of the communication protocol is controlled independently by the operating system, according to the underlying transport protocol. Figure 2 shows a typical sequence of I/O system calls and the

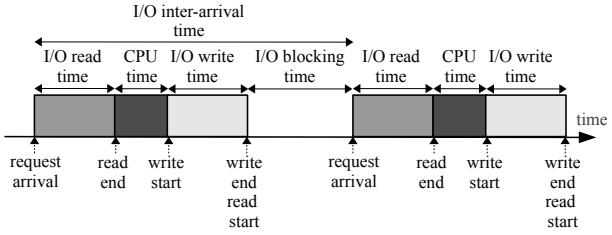

Figure 2: Overlapping of I/O Times

I/O blocking times and transfer times. I_T is the sum of the I/O read and write times. Let $\lambda_{I/O}$ be the inter-arrival time of I/O requests:

$$I_B = \frac{1}{\lambda_{I/O}} - I_T - C \tag{17}$$

If we consider $\lambda_{I/O}$ independent of the I/O sequence response time, the blocking time and I/O transfer time of a thread can be overlapped. Thus, for a thread, the response time of the I/O incurred during the request-reply episode is:

$$I \approx max(I_T + C, \frac{1}{\lambda_{I/O}}) \tag{18}$$

and the I/O idle time, $D_{I/O}$, is the difference between inter-arrival time and the sum of CPU time and I/O transfer time.

However, in server workloads such as *Apache* or *memcached* a thread multiplexes multiple network sockets, such that the CPU time incurred by one request-reply can overlap with the transfer time incurred by another request-reply [36]. Thus, the I/O time of a program is

$$I = max(I_T, \frac{1}{\lambda_{I/O}}) \tag{19}$$

where I_T is determined as the ratio of transferred data to the network bandwidth, and the I/O blocking time is derived from the arrival rate of I/O requests.

2.4 Energy Model

The energy model predicts the energy used (E) used as a factor of number of cores, n and core clock frequency f. The approach is to divide the energy used based on the three types of resources: cores, memory and I/O device. The total energy of the system, E is:

$$E = E_{CPU} + E_M + E_{I/O} \tag{20}$$

The energy used by the cores depends on how the number of active cores and the type of activity effected by the cores. Let $P_{CPU}(n, f)$ be the power drawn by the processor when n core are active and operating at frequency f. By convention we denote $P_{CPU,idle} = P_{CPU}(0, f)$. The energy consumed by the cores throughout the execution of a program is:

$$E_{CPU} = \sum_{k=0}^{\#cores} T_n P_{CPU}(n, f) \tag{21}$$

where T_n is the total wall clock time when n cores are active. Because the workloads are fully parallelizable, the modeling of T_n can be simplified. Assuming that the program uses n threads, we split the entire execution time of the program, T, into a period during which n cores are active and periods during which no cores are active[1]:

$$\begin{aligned} E_{CPU} &= T \cdot U_{CPU} \cdot P_{CPU}(n, f) + \\ &\quad T(1 - U_{CPU}) P_{CPU,idle} \end{aligned} \tag{22}$$

The power and energy usage of the rest of the system (i.e. video, storage, peripheral devices, voltage stabilizers etc.) is considered fixed and independent of the workload, and accounted in $P_{CPU,idle}$.

The power drawn by a core depends on the type of activity effected by the core. Let $P_{WORK}(n, f)$ be the power consumed by n cores when executing work cycles, and $P_{STALL}(n, f)$ be the power consumed when executing stall cycles. Because threads are considered homogeneous, all cores execute an equal mix of instructions and the power drawn by n cores is the average of P_{WORK} and P_{STALL}, weighted with the ratio of work to stall cycles:

$$P_{CPU} = \frac{w \cdot P_{WORK} + \big(c - w\big) P_{STALL}}{c} \tag{23}$$

Both P_{WORK} and P_{STALL} are system characteristics that depend on n and f, while w and $c(n, f)$ are workload characteristics. The energy incurred by the memory is divided into energy incurred when there are no memory requests, $E_{M,idle}$ and energy incurred when the memory is serving memory requests, $E_{M,active}$:

$$E_M = E_{M,active} + E_{M,idle} \tag{24}$$

The total time where requests are serviced by the memory is memory service time, S_M, while the time when the memory does not service requests is $T - S_M$. Because $S_M = r \cdot s_M$:

$$\begin{aligned} E_{M,active} &= r \cdot s_M \cdot P_{M,active} \\ E_{M,idle} &= (T - r \cdot s_M) P_{M,idle} \end{aligned} \tag{25}$$

Similarly, for the I/O requests, the time when the I/O device is busy transferring data is I_T, while the idle time is $D_{I/O} = T - I_T$:

$$\begin{aligned} E_{I/O} &= E_{I/O,active} + E_{I/O,idle} \\ E_{I/O,active} &= I_T \cdot P_{I/O,active} \\ E_{I/O,idle} &= (T - I_T) \cdot P_{I/O,idle} \end{aligned} \tag{26}$$

The model separates the impact of the system parameters from the workload parameters. P_{WORK}, P_{STALL}, $P_{CPU,idle}$, P_M, and $P_{I/O}$ are independent of workloads. Thus, they can be measured once and then used as constants in the model. In contrast, w and I_T depend only on the workload, while c, T, U_{CPU} depend both on workload and on the system, as described in the previous section.

[1]Our workloads vary less widely than online data-intensive services. For more realistic workloads this assumption results in an underestimation of the power usage, because cores become active at discrete intervals, rather than as a cohort. See Meisner et al. [32] for a detailed power characterization of such workloads.

2.5 Summary of Model

The objective of the model is to derive the energy performance of server workloads on low-power multicore systems. The model is based on the modeling the overlap of CPU cores, memory and I/O execution. Since the energy incurred is the product of power utilization and execution time, the first step is to model the execution time, based on a bottleneck analysis technique:

$$T = max\big(\frac{C}{n}, I\big) \tag{27}$$

The model for program execution time (T) is divided into CPU response time (C) and I/O response time (I), as follows:

$$C(n, f) = max\big(\frac{w + b}{f}, T_M\big) \tag{28}$$

$$I = max\big(I_T, \frac{1}{\lambda_{I/O}}\big) \tag{29}$$

In modeling C, we consider two main cases. When the cores are the bottleneck, execution time is the sum of work cycles and stall cycles not due to memory contention. But when memory is the bottleneck, memory request response time is modeled using an M/G/1 queuing system and we consider two main types of memory contention. Our workload characterization shows that bursty memory traffic fits the Pareto distribution, and non-bursty memory traffic is exponentially distributed. The second step is to model the energy used by CPU, memory and the I/O device, considering the response times of these resources and the power consumption for each resource:

$$\begin{aligned} E &= T \cdot U_{CPU} \cdot \frac{w \cdot P_{WORK} + \big(c - w\big) P_{STALL}}{c} \\ &+ T(1 - U_{CPU}) P_{CPU,idle} \\ &+ r \cdot s_M \cdot P_{M,active} \\ &+ (T - r \cdot s_M) P_{M,idle} \\ &+ I_T \cdot P_{I/O,active} \\ &+ (T - I_T) \cdot P_{I/O,idle} \end{aligned} \tag{30}$$

The values of active and idle power consumption of each resource are measured during system characterization and are used as constant inputs of the model.

3. MODEL PARAMETERIZATION AND VALIDATION

In this section we present the model parameterization. We perform two baseline runs to collect the model input parameters. With these parameters, we can predict the execution time and energy consumed on any configuration of cores and clock frequency.

3.1 Workloads and System Setup

The workloads studied in this paper are representative for three types of application domain: high performance computing (HPC), web-hosting and financial computing. Table 2 shows the five programs analyzed in this paper. We chose three HPC programs from NAS Parallel Benchmark (NPB) suite that correspond to three degrees of memory contention: low (*EP*), medium (*FT*) and high (*SP*). All HPC programs are implemented in OpenMP, are highly parallel and have no I/O operations. *Memcached* is an in-memory

Domain	Program	Benchmark	Description	Problem Size	Work
HPC	*EP*	NPB 3.3	Monte-Carlo methods	536,870,912 pairs	Float
	FT		Fast Fourier transform	6 iterations, 256×256×128 grid	
	SP		Penta-diagonal matrices solver	1 iterations, 162^3 grid	
Web-hosting	*memcached* 1.4.13	memslap 0.44	In-memory key-value store	594,000 key get + 6,000 key set	Integer
Financial	*blackscholes*	PARSEC 2.1	European options pricing	5 million shares	Float

Table 2: Server Workloads

key-value store widely used as a caching solution for web content. We ran *memcached* on a low-power system with a cache size of 250 MB. Memcached uses pthreads as worker threads to serve requests. The request are issued by program *memslap*[2], which stores 6,000 keys in memory and performs 594,000 get operations. Memslap is run on another system (Intel Core i7), which is connected to the system under test through a 100 Mbps network interface. Memslap continously sends requests to *memcached*, with the thinking time between requests independent on the response rate of the request. *Blackscholes* is a program from the PARSEC 2.1 benchmark suite that computes a series of European options prices by solving numerically the Black-Scholes partial differential equation. *Blackscholes* uses pthreads as worker threads and has no I/O operations. The problem sizes chosen for all programs result in execution time of at least thirty seconds on all configurations of cores and clock frequency. The operating system is Linux using kernel 3.6.0. All programs are compiled using GCC with optimizations (`-O2` for *memcached* and *blackscholes* and `-O3` for the HPC programs). The useful work for HPC and *blackscholes* consists of NEON-VFPv3 floating point operations (without vectorization), and integer operations for *memcached*.

The low-power system used for our test is a ODROID-X with a quad-core Exynos 4412 ARM Cortex-A9 processor. The core frequencies supported are between 200 MHz and 1400 MHz, in increments of 100 MHz, the available bandwidth between cores and main memory is 800 MB/s, and the network device is a 100 Mbps Ethernet card.

`PAPI` version 5 is used to access the processor hardware events counters. We use `strace` and `tcpdump` to log the size and inter-arrival time of I/O requests. For measuring the power and energy consumption, we use a Yokogawa WT210 digital power meter. Figure 3 shows the setup of the system. An Intel Core i7 system is used

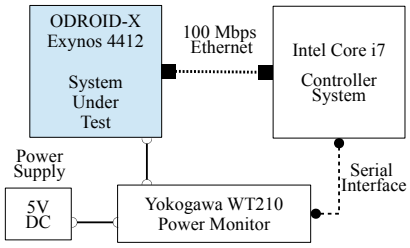

Figure 3: System Setup

as a controller, connected to the system under test using a direct 100 Mbps Ethernet link. The controller starts and stops all experiments and collects all the data. The power monitor outputs every second the average power during the last second, and total energy used.

3.2 Model Parameterization

We perform a series of measurement experiments to parameterize the input parameters of our model. The parameters are either *system characteristics* such as power consumption during different type of activities, memory bandwidth and I/O bandwidth, or *workload characteristics* such as memory request arrival rate, memory burst size or I/O arrival time.

3.2.1 System Characteristics

To determine the system parameters used as inputs by our model, we execute a series of microbenchmarks that we have designed to determine the power consumption during different types of activities. We use three programs that each stresses one type of CPU activity: work integer cycles, work floating point cycles and stall cycles. Additionally, we collect the idle power. To determine the power drawn by memory and Ethernet device, we selectively turn on and off these components. All the power values reported in this section are obtained by averaging the results across three repetitions. Table 3 shows the static power characteristics of the system.

First we determine the total system power under idle load, when changing the core frequency. We measured total system power and processor-only power, which is obtained by discounting the power drawn by the other components. Next, we profile the processor active power when the cores execute two types of work cycles: integer operations and floating point operations. To determine this power, we use a microbenchmark designed by us that achieves close to 100% core pipeline utilization under each type of operations. The results are determined for different number of cores, under each supported clock frequency. To profile the stall cycles, we use a microbenchmark written by us that reads a large amount of data from memory, continuously attempting to miss the last level of cache. This benchmark results into more than 90% stall cycles in the cores pipeline and intense memory activity. For this microbenchmark we measure the total system power and deduct the power incurred by the memory. As shown in the table, for small core frequencies and core counts, the processor does not fully stress the memory bandwidth, and thus, the stall cycles power cannot be measured for these configurations. When applying the model, we approximate by zero the power drawn on these configurations when the cores are executing stall cycles only. The memory idle power is taken from the literature [30] and is approximated as 28 mW. We measure the active memory power by running the microbenchmark that constantly misses the cache. Under this state, the memory draws approximately 248 mW, derived after discounting the power drawn by the rest of the system. The I/O power load is determined by measuring the total power with network card under full load, under idle load and turned off. The Ethernet card draws 200 mW, irrespective of the load.

The service time for one memory request, s_M is determined as the ratio of the cache line to the available memory bandwidth. When there is no I/O operation, $s_M = 38.14$ns.

3.2.2 Workload Characterization

First we perform a baseline run of the target program, on configurations with one core, alternating between the lowest two frequencies of 200 and 300 MHz and the maximum frequencies of 1300 and 1400 MHz every ten seconds. We measure the total number of cycles incurred on all cores, stall cycles and last-level cache misses for each interval of one second spent at frequency f. By using the

[2]Memslap issues requests with constant size and uniform popularity, which may lead to higher CPU utilization than in actual usage of memcached. For practical traffic characteristics see Atikoglu et al. [6]

Frequency [kHz]	Idle Power [mW]		Stall cycles power [mW]				Float power [mW]				Integer power [mW]			
	System	Processor	Number of Cores				Number of Cores				Number of Cores			
			1	2	3	4	1	2	3	4	1	2	3	4
200	1,740	1,512	–	–	–	21	35	70	108	137	43	90	132	176
600	1,796	1,568	22	128	192	255	122	244	360	465	152	306	436	592
1,000	1,880	1,652	150	308	428	583	255	540	779	1,005	310	655	941	1,260
1,400	2,081	1,853	369	649	871	1,114	550	1,079	1,609	2,220	662	1,328	2,079	2,869

Table 3: Static Power Characterization

lowest two frequencies and the highest two frequencies, we obtain four data points of $C(f)$ which allows us to determine whether the program is CPU-bounded or memory-bounded. When operating at the smallest two frequencies, the program is very likely to be CPU-bounded. However, if the nature of the workload requires many off-chip requests, the program can become memory-bounded when operating at the highest two frequencies. According to equation 8, if $C(f)$ does not change significantly when f is changed, the program is CPU-bounded. However, if $C(f)$ increases with an increase in f, the program is memory bounded.

During a second run, we profile the burstiness of memory traffic by measuring the number of last level cache misses performed by one thread during each 1 millisecond interval. The run is executed on all cores, at the maximum core frequency. Additionally, during this run we profile the number of I/O operations, total data transferred over the I/O device and the inter-arrival time of I/O requests. Our analysis of the memory access rate shows that the server workloads can be classified as two types of memory traffic: bursty memory traffic or non-bursty memory traffic. Programs with bursty memory traffic exhibit a clear separation between phases of core activity versus phases of memory activity. Due to this separation, they are less likely to cause memory contention among cores. In contrast, programs which are severely memory-bounded almost always trigger memory requests, and thus, there is a more uniform utilization of the memory bandwidth. To model the size of the memory request, we use a Pareto distribution for programs with bursty memory traffic, and an exponential distribution for programs with non-bursty memory traffic.

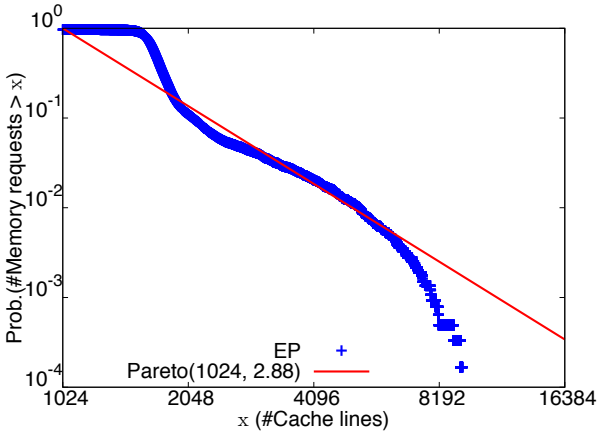

Figure 4: Bursty Memory Traffic: *EP*

Figure 4 shows the probability that the number of last level cache hits incurred during an interval of $1,000,000$ cycles exceeds x, for $x \in [1024, 16384]$. The probability plot, in log-log scales shows a decreasing diagonal line, which confirms the bursty nature of the memory traffic. A Pareto distribution with parameters of minimum size $x_{min} = 1024$ and hazard rate $\alpha = 2.88$ fits the measured data, with coefficient of regression $R^2 = 0.94$. Similar plots are observed for *memcached* and *blackscholes*, with slightly different α parameters. Figure 5 shows the probability distribution for the non-bursty program SP. In constrast with *EP*, the number of last

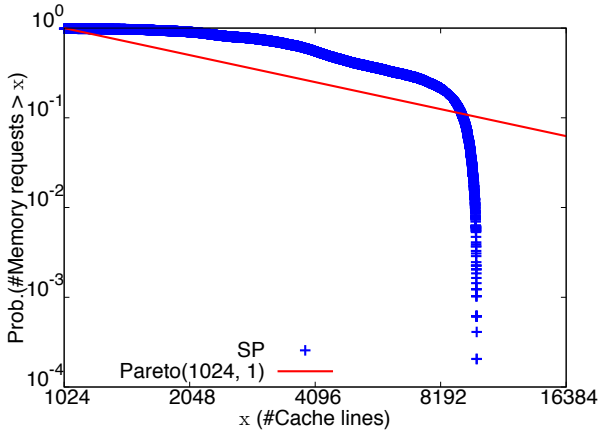

Figure 5: Non-bursty Memory Traffic: *SP*

level misses occurring during each $1,000,000$ cycles interval is much larger, and is not significantly bursty. The Figure also shows that even the least-bursty Pareto distribution with $x_{min} = 1024$ and $\alpha = 1$ does not fit the data. Therefore, we use an exponential distribution to model the average size of the memory request performed by programs with bursty memory traffic. HPC program *FT* has a similar behavior, but with lower memory request arrival rate.

Memcached is the only program with significant I/O operations. During each *get* requests, *memcached* reads 70 bytes and writes 1100 bytes, while a *set* request reads 1105 bytes and writes 8 bytes. The get operations are performed in batches of 24 requests. The total amount of data transferred is 700MB, while the average inter-arrival time between batches of get requests is measured at 2.5 ms. Thus, $I_T = 56$s and $I_B = 5.1$s. Because the controller and the system under test are connected using point to point network, we observe a zero packet loss rate, thus the network I/O bandwidth utilization is 100% during periods of data transfer.

Table 4 summarizes our workload characterization for the five programs. For HPC programs and *blackscholes*, we observe that

Program	Distribution	Parameters	r_M	$w+b$
EP	Pareto	$\alpha = 2.88$	1.09×10^6	1.20×10^{11}
FT	Exp.	$\lambda = 7.1 \times 10^5$	1.46×10^8	4.10×10^{10}
SP	Exp.	$\lambda = 1.7 \times 10^6$	5.11×10^8	6.16×10^{10}
memcached	Pareto	$\alpha = 2.20$	8.90×10^8	5.64×10^{10}
blackscholes	Pareto	$\alpha = 2.75$	4.43×10^7	9.20×10^{11}

Table 4: Workload Characterization

work cycles are completely dominated by executing floating point operations. Thus, when applying the power model, we use P_{WORK} values from the floating point power characterization. In contrast, for *memcached* we use P_{WORK} from the integer power characterization.

3.3 Validation

We validate the model against measurements of execution time and energy usage. First we show summary validation results for

all five workloads. Due to space constraints, we focus the validation discussion on programs *memcached* and *SP*, which are the programs with largest relative error between model and measurement.

Program	Error	
	Time	Energy
EP	1.0%	8.7%
FT	9.3%	9.9%
SP	11.1%	11.3%
memcached	11.4%	10.7%
blackscholes	9.4%	8.2%

Table 5: Relative Error between Model and Measurement

Before the validation experiments, we applied two baseline runs for each program, during which we collect information about the useful work, memory requests and I/O requests, as described in the previous section. To validate the model, we predict the execution time $T(n, f)$ and energy $E(n, f)$ considering the number of cores n and clock frequency f fixed throughout the program execution. We compare this prediction against measurements of $T(n, f)$ and $E(n, f)$. To change the number of cores that the program is using we change the number of worker threads of the programs. For each program, we validate four cores and 13 core frequencies, giving 52 core-frequency configurations. Table 5 summarizes the relative error between measurements and predicted values, averaged over all core-frequency configurations. Overall, the error across all experiments is around 9%, and 70% of the predicted values are under 5% error.

We identify three factors that affect the accuracy of the model. The most significant source of error comes irregularities during execution. For example, *memcached* incurs more instructions on higher core frequencies, which are caused by a polling mechanism used to monitor the network sockets. This significantly increases the energy used, but does not reduce the execution time. This increase causes our model to underestimate by up to 23% the CPU cycles incurred by *memcached* on one core. The second cause for model inaccuracy is the accuracy of the system characterization parameters. In particular, the power values for active cycles, stall cycles and idleness differ by up to 20mW. This variability translates into a slight underestimate of the average power, especially for configurations with low frequencies or low core counts. Third, the measured values of execution time and energy show a variation of 3 to 11%. Figures 6 and 7 show the validation of execution time and energy used for *SP* and *memcached*.

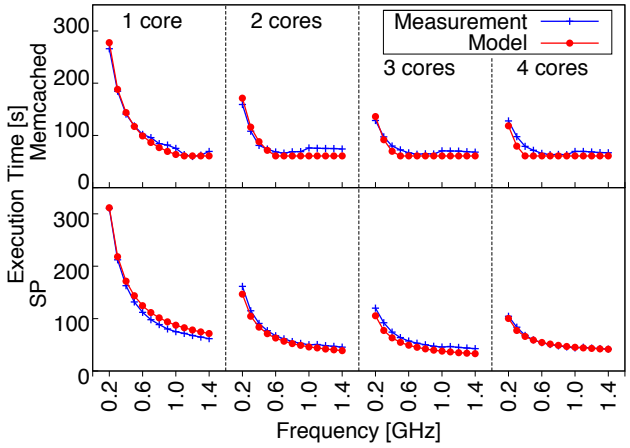

Figure 6: Execution Time Validation

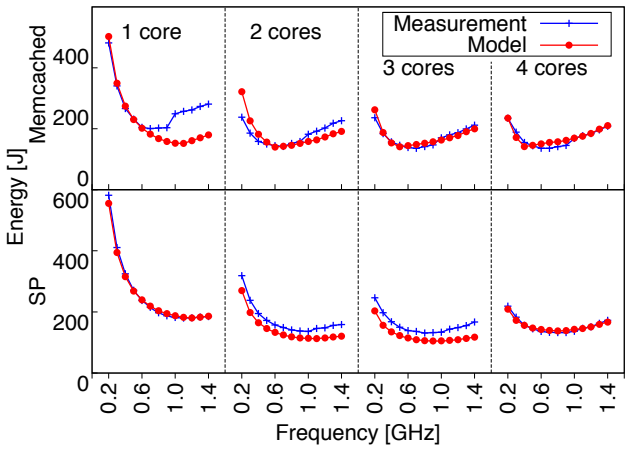

Figure 7: Energy Validation

4. ANALYSIS

We present three studies performed using our model. Firstly, we predict the configuration of number cores and clock frequency, (n, f) that achieves the minimum execution time without wasting energy, and discuss the energy savings that can be achieved. Secondly, we show that some types of workloads incur higher energy cost on low-power multicore than on traditional x64 systems. Thirdly, we show that contrary to intuition, the most energy-efficient way to balance the utilization of core, memory and I/O resources is by adding more memory and I/O resources, rather than by turning off idle cores.

4.1 Optimal Core-Frequency Configuration

We apply the model to predict the configuration of core counts and clock frequency that achieves the minimum execution time. If several configurations achieve execution times within 5% to the minimum, we chose the one with the minimum energy usage, considering that all cores are kept powered on. The results in table 6 shows that only workloads consisting entirely of arithmetic instructions make full use of all four cores. In contrast, all memory- or I/O-bounded programs achieve best performance on less than full core count and around half of maximum core frequency.

Program	Bottleneck	Configuration			
		Min. Time		Min. Energy	
		n	f [GHz]	n	f [GHz]
EP	Cores	4	1.4	4	1.4
FT	Memory	3	1.4	3	0.7
SP	Memory	3	1.4	3	0.9
memcached	I/O	2	0.7	2	0.7
blackscholes	Cores	4	1.4	4	1.4

Table 6: Minimum Time and Energy Configurations on Default Configuration of up to 4 Cores and 1.4 GHz

The predictions for optimal core-frequency configuration allow significant energy savings if un-necessary cores are turned off. For example, for *memcached*, by selecting the minimum time configuration we can power off two cores. Unfortunately, the operating system on our system does not allow selective power off of a subset of cores, and thus we cannot measure directly the energy savings. However, using figures from related work [23, 33], by shutting down two out of the four cores we can estimate a reduction of processor power between a conservative 25% and an optimistic 50%. With these figures, applying the configuration predicted by our model allows for a reduction in total energy savings between

13% and 31%, and without compromising the execution time performance.

4.2 Impact of Memory Bottleneck

We analyze the energy cost of low-power multicores that have limited memory bandwidth. We compare the execution of memory-bounded *SP* on low-power Cortex-A9 with traditional x64 multicore.

The x64 system used is Intel Xeon X5650 at 2.67 GHz, dual processor with 12 cores/24 hardware threads, 24 MB of L3 cache and two memory controllers, each with two memory channels. We ran program *SP* with a large input size of 400 iterations on a grid of 162^3 (input size *C* from NPB). The input results in a working set large enough to exceed the caches of both systems, but fits into 1 GB of main memory.

We apply our prediction for execution time on the low-power multicore for all core-frequency configurations, and observe that the minimum execution time exceeds 17,000 seconds. In contrast, the measured execution time on the x64 system is 330 seconds, using all cores at maximum frequency. The large gap between execution times appears because the Intel system is equipped with much larger caches, and thus, incurs 15 times less cache misses. Furthermore, the main memory bandwidth is around 8 times higher, while the bandwidth of the caches are ten times (for L1) and four times (when comparing Intel L3 to ARM L2) higher. The average power used by the Intel system (with disks turned off) is around 210W.

We extend this analysis for datacenters, factoring the additional power required for power conversion and cooling the systems. The Power Usage Effectivness (PUE) of a datacenter measures the total power required to deliver 1 Watt of IT power. From literature, we identify two PUE bounds: the lower bound is PUE=1.13, in a Google datacenter [3], while an the upper bound is 1.89 [4]. Figure 8 shows the predicted values of energy consumption on low-

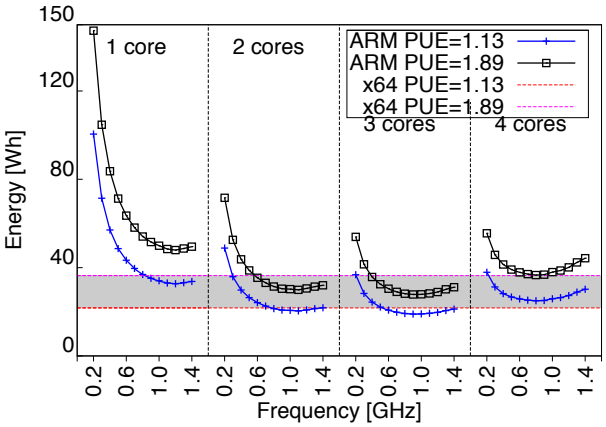

Figure 8: *SP* Energy Usage: ARM Cortex-A9 vs x64

power multicore, for all configurations of cores-frequency, compared with an execution on all x64 cores at maximum frequency, for PUE between 1.13 and 1.89. Few ARM configurations manage to achieve lower energy cost than x64, with less than 7% energy reduction, but at a cost of incurring execution time more than 50 times higher. However, the x64 execution is not energy efficient because all cores are used at maximum frequency even though the memory is the bottleneck. We conclude that memory-bounded programs can be unsuitable for low-power multicores, even if optimizing the core-frequency configuration to achieve best performance at minimum energy cost.

4.3 Energy Proportionality of Server Workloads on Low-power Multicores

Energy proportionality refers to the ability of a system to consume power proportional to their performance [7]. ARM multicores typically have good energy proportionality when used as mobile computers, due to their sleep states and low-power operation [33]. However, our previous analysis shows that resource imbalances lead to large energy wastage in server workloads. Thus, leveraging on the idea that ARM systems are highly configurable, we apply our model to understand how to improve the energy proportional executions of server workloads.

As the key to improving energy proportionality is system balance [5], we apply our model to predict the performance of program *memcached* under different hardware configurations that balance the system resources. Figure 9 shows the response times of different resources for the original hardware configuration (100Mbps Ethernet, one memory controller), when using two active cores. This number of cores is selected as it achieves the best performance at minimum energy cost. For small core frequencies, the CPU work

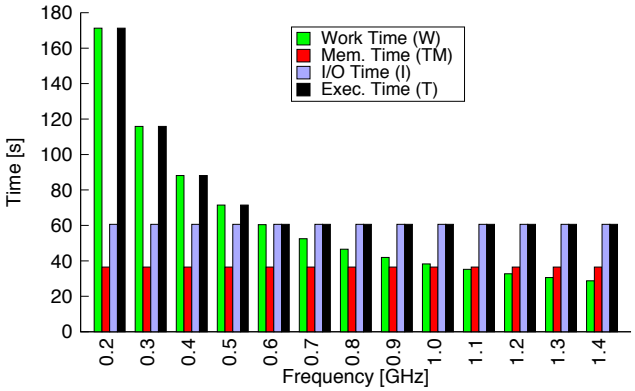

Figure 9: *Memcached* Response Times

time ($W = \frac{w+b}{f}$) is the bottleneck, but at 600MHz the CPU response time matches the I/O response time. Beyond 600MHz, the I/O bandwidth becomes the bottleneck, and the exection time does not reduce anymore.

We analyze the performance impact of replacing two system components. First, the 100Mbps Ethernet is replaced with 1GBps Ethernet, without modifying any other component. In this analysis, we consider a gigabit Ethernet adapter with a power consumption of 600mW, which is typical for a power-efficient network card. The I/O time for *memcached* is composed of transfer time $I_T = 56$s and blocking time $T_B = 5.1$s. With a Gbit Ethernet, the total I/O time becomes $I = 10.7$s. However, because the I/O device is memory mapped, we consider that the gigabit Ethernet will utilize 125MB/s out of the 800MB/s memory bandwidth. Thus, s_M increases from 38 ns to 45ns. Applying Equation 16, T_M increases from 36.5s to 47s. Thus, the effect of moving to 1GBit Ethernet is a reduction from of execution time from $T = 61$ s to $T = 47$ s, and the system bottleneck becomes the memory. Due to the increase in I/O power by 400mW and due to the increase in stall cycles, the average power increases by approximately 500mW. Figure 10 shows that total energy decreases by switching to a Gigabit Ethernet because the decrease in execution time offsets the increase in average power. Since the new bottleneck is the memory, we consider next the impact of doubling the effective memory bandwidth (ARM Cortex-A9 systems can be configured to up to quad-memory channels, while the next generations ARM Cortex-A15 and ARM Cortex-A50 support more outstanding memory requests and can be configured to use LPDDR3). With the double memory bandwidth, the memory

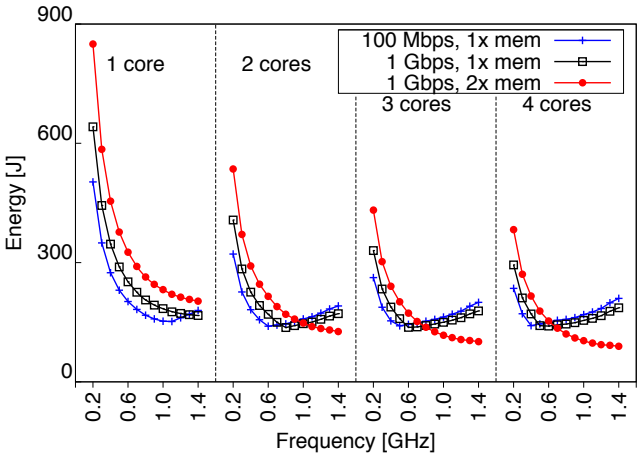

Figure 10: *Memcached* with 1 Gbit Ethernet and Double Memory Bandwidth

response time drops to 18.1s, and the system bottleneck becomes the core. We consider a pessimistic scenario, where power consumption is the quadruple of the original memory system (100mW idle memory power and 1W active memory power). However, the energy consumption still decreases by more than 50%, and *memcached* becomes CPU-bounded. It achieves best performance and minimum energy when using all cores at full frequency.

This analysis showed that reducing the imbalance among core, memory and I/O leads to lower energy usage. However, counter to intuition, we showed that balancing the resources by adding *more hardware resources* is the key to improving energy proportionality, even if this results in higher average power consumption. Considering that future ARM systems such as ARM Cortex-A15 and the 64-bit ARM Cortex-A50 family target much improved memory and I/O throughput, it is expected that future multicore systems based on low-power ARM multicores will deliver better energy-proportionality than current ARM Cortex-A9, even at a cost of higher power usage.

5. RELATED WORK

Previous work on understanding the performance and energy efficiency of server workloads can be broadly classified based on three criteria: (i) level of analysis (i.e. microarchitecture level, system level or cluster/datacenter level performance) (ii) workload characteristics and (iii) approach for understanding the energy usage. We discuss each category, contrasting our approach with previous research.

Level of Analysis. A large body of related work considers using voltage and frequency scaling to balance core and memory performance in single-core workloads [26, 28, 39, 40]. However, with the reduction of transistor size with each technological generation, an increasingly large component of the energy cost is caused by fixed leakage power required to keep the components powered on [23]. As such, DVFS techniques are facing diminishing returns, as they are able to reduce an increasingly small fraction of the energy cost [22]. At the other end of the spectrum, in clustered systems or datacenters, a dominant performance issue is the energy wastage due to low processor utilization. Research in this area has focused on understanding how to increase utilization by workload scheduling [15, 38] or the impact of adjusting system power [13, 19]. At multicore level, a large body of work addresses the question of how the throughput of multicore system is affected by the dichotomy of few *brawny* cores versus many *wimpy* cores, given a fixed performance-density budget [12, 17, 29]. However, these studies do not consider the impact of off-chip resources such as main memory or I/O periferals.

Our findings are orthogonal with previous work. Because we focus on understanding the impact of memory and I/O constraints on execution perfromance, our model can predict the optimal number of cores for a workload, our model allows a better understanding of the impact of work-aggregation or different scheduling techniques in clusters of multicore servers. Furthermore, our work enables a greater reduction of energy over DVFS techniques developed for single-core systems, because it can reveal the number of unnecessary cores that can be turned off.

Workload Characteristics. The suitability of low-power systems to achieve energy-efficient executions of different type of workloads has been studied in the past for many types of systems [25, 35], including ARM [33] and Intel Atom systems [5]. Previous work on understanding the power and performance of ARM systems focused mostly on embedded [33] or mobile workloads [8], which have mostly CPU and GPU requirements. Closer to our targeted workloads is FAWN, a cluster of low-power “wimpy" AMD Geode nodes [5], designed for energy-efficient I/O intensive applications. However, FAWN nodes are single-core 500 MHz processors, thus they are severely bottlenecked by the CPU processing ability and by I/O storage bandwidth. In contrast, the multicore Cortex-A9 systems addressed by us are bottlenecked by memory or network speed. Furthermore, our analysis includes non-I/O bounded HPC programs that are more impacted by the “memory wall”. Work that addresses specifically data-intesive workloads are characterized by [6] in terms of traffic characteristics and by [32] for opportunities of power savings in datacenters. Our work, although using much narrower traffic patterns, validates the conclusion that system balance is key for energy proportionality.

Modeling Approaches. Previous analytical models for predicting execution time include trace-driven analysis for predicting the speedup and speedup loss due to data-dependency [37], the impact of network communication [20] or memory contention of HPC and real-world programs in traditional multicore systems [27]. In contrast, our focus is not on performance loss, but rather on understanding energy wastage due to resource imbalance. A significant body of research has addressed analytical modeling of computer power and energy using models derived from first-order principles or linear regression [10, 21, 24, 26, 28]. Closest to our model is the approach presented by Curtis-Maury et al. [10]. They derive an empirical analytical model for predicting the impact of program concurrency and core frequency on energy usage of HPC programs in traditional multicores. Their approach is based on linear regression over measured data acquired over many training runs, but without considering the impact of resource contention. In contrast, we provide closed-form equations that explicitly model the impact of number of cores, allowing us to study directly the impact of multicore on resource imbalances. More importantly, our analysis suggests that increasing power usage can lead to energy reductions, if the added power contributes to balancing the resource utilization.We share the methodology of modeling the power usage by correlating static power characteristics with hardware events counters with [24, 26, 28]. We do not use linear correlation methods, but derive close-form equations for the power and energy consumption. Our work furthers these studies by considering the case where multiple cores execute the same workload, modeling the impact of multicore on resource contention. This increases the predictive value of our models. Furthermore we use validation against direct measurement of server workloads covering HPC, web-hosting and financial computing.

6. CONCLUSIONS

This paper proposes a trace-driven analytical model for understanding the energy usage of server workloads on low-power multicore systems. We model the effects of multicore on achieving energy-efficient executions of representative server workloads covering high performance computing, web hosting and financial computing. The key idea is the modeling of the overlap between resource demand in a program. Since the power consumed is the product of power utilization and execution time, the model first estimates the execution time of a program by considering the overlap between response times incurred by cores, memory and I/O resources. CPU time, which accounts for both cores and memory response time, is modeled as an M/G/1 queuing system. Our workload characterization shows that bursty memory traffic fits a Pareto distribution and non-bursty memory traffic can be modeled using an exponential distribution. Validation shows a relative error of 9% between model and measured execution time and energy. Applications of our model to analyze the optimal core-frequency configuration, impact of memory bottleneck and energy proportionality in multicore systems reveal a number of insights. We observe that low-power multicores may not always deliver energy-efficient executions for server workloads because large imbalances between cores, memory and I/O resources can lead to under-utilized resources and thus contribute to energy wastage. Next, resource imbalances in HPC programs may result in significantly longer execution time and higher energy cost on ARM Cortex-A9 than on a traditional x64 server. In this instance and without compromising execution time, our model predicts core frequency configurations that balance the resources with energy reduction of up to one third. Finally, we show that higher memory and I/O bandwidths can improve both execution time and energy utilization, even if it means higher power usage. Thus, it is expected that ARM Cortex-A15 and the ARM Cortex-A50 family, which target larger memory and I/O bandwidths, will deliver more energy-efficient servers than currently available ARM Cortex-A9.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their constructive comments, and Lavanya Ramapantulu for helping us to uncover some aspects of the ARM Cortex-A9 processors.

7. REFERENCES

[1] *Chip maker Calxeda receives $55 million to push ARM chips into the data center*, Oct 2012. http://www.webcitation.org/6BSIjQzCM.

[2] *Dell Reaches for the Cloud With New Prototype ARM Server, PCWorld Magazine*, May 2012. http://www.webcitation.org/6BVBj0Oyz.

[3] *Google Data Center Efficiency: How We Do It*, Oct 2012. http://www.webcitation.org/6C8PjIMYd.

[4] *Uptime Institute 2012 Survey*, Oct 2012. http://uptimeinstitute.com/2012-survey-results/.

[5] D. G. Andersen et al. Fawn: a fast array of wimpy nodes. *Proc of SOSP*, pages 1–14, 2009.

[6] B. Atikoglu et al. Workload analysis of a large-scale key-value store. *Proc of SIGMETRICS/PERFORMANCE*, pages 53–64, 2012.

[7] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[8] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. *Proc of USENIX ATC*, pages 21–21, 2010.

[9] J. Corbet et al. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[10] M. Curtis-Maury et al. Prediction models for multi-dimensional power-performance optimization on many cores. *Proc of PACT*, pages 250–259, 2008.

[11] F. M. David et al. Context Switch Overheads for Linux on ARM Platforms. *Proc. of ExpCS*, 2007.

[12] J. D. Davis et al. Maximizing cmp throughput with mediocre cores. *Proc of PACT*, pages 51–62, 2005

[13] A. Gandhi et al. Optimal power allocation in server farms. *Proc of SIGMETRICS*, pages 157–168,2009.

[14] A. Gandhi et al. Are sleep states effective in data centers? *Proc of IGCC*, pages 1–10, 2012.

[15] D. Gmach et al. Workload analysis and demand prediction of enterprise data center applications. *Proc of IISWC*, pages 171–180, 2007.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. 2006.

[17] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[18] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[19] T. Horvath and K. Skadron. Multi-mode energy management for multi-tier server clusters. *Proc of PACT*, pages 270–279, 2008.

[20] Y. Hu et al. I/o scheduling model of virtual machine based on multi-core dynamic partitioning. *Proc of HPDC*, pages 142–154, 2010.

[21] V. Kumar and A. Fedorova. Towards better performance per watt in virtual environments on asymmetric single-isa multi-core systems. *SIGOPS Oper. Syst. Rev.*, 43(3):105–109, July 2009.

[22] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. *Proc of HotPower*, 2010.

[23] E. Le Sueur and G. Heiser. Slow down or sleep, that is the question. *Proc of USENIX ATC*, 2011.

[24] A. W. Lewis et al. Runtime energy consumption estimation for server workloads based on chaotic time-series approximation. *ACM Trans. Archit. Code Optim.*, 9(3):15:1–15:26, Oct. 2012.

[25] K. Lim et al. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. *Proc of ISCA*, pages 315–326, 2008.

[26] M. Y. Lim et al. Softpower: fine-grain power estimations using performance counters. *Proc of HPDC*, pages 308–311, 2010.

[27] F. Liu et al. Understanding How Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. *Proc of HPCA*, 2010.

[28] C. W. Lively et al. Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems. *Computer Science - R&D*, 27(4):245–253, 2012.

[29] P. Lotfi-Kamran et al. Scale-out Processors. *Proc of ISCA*, pages 500–511, 2012.

[30] K. Malladi et al. Towards energy-proportional datacenter memory with mobile dram. *Proc of ISCA*, pages 37–48, 2012.

[31] D. Meisner et al. PowerNap: Eliminating Server Idle Power. *Proc of ASPLOS*, pages 205–216, 2009.

[32] D. Meisner et al. Power management of online data-intensive services. *Proc of ISCA*, pages 319–330, 2011.

[33] R. Mijat. System level benchmarking analysis of the cortextm-a9 mpcore. *ARM Connected Community Technical Symposium*, 2009.

[34] D. Molka et al. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *Proc of PACT*, pages 261–270, 2009.

[35] A. S. Szalay et al. Low-power Amdahl-balanced blades for data intensive computing. *SIGOPS Oper. Syst. Rev.*, 44(1):71–75, Mar. 2010.

[36] Y. C. Tay. *Analytical Performance Modeling for Computer Systems*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2010.

[37] B. M. Tudor and Y. M. Teo. A practical approach for performance analysis of shared-memory programs. *Proc of IPDPS*, pages 652–663, 2011

[38] A. Verma et al. Server workload analysis for power minimization using consolidation. *Proc of USENIX ATC*, 2009.

[39] M. Weiser et al. Scheduling for reduced cpu energy. *Proc of OSDI*, 1994.

[40] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. *Proc of CASES*, pages 238–246, 2002.