The effectiveness of multicore processors in modern smartphones

Daniel J. R. Home
Darwin College

Supervised by Dr Andrew Rice

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science Option B

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Daniel.Home@cl.cam.ac.uk

June 14, 2013

Declaration

I Daniel J. R. Home of Darwin College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,904

Signed:

Date:

Abstract

It has become increasingly common for modern high end smartphones to include multicore processors for the purposes of enhanced performance, greater battery life and consumer desirability. However, it has recently been claimed by Intel that *Android is not ready for multicore processors*. If this statement is valid, a significant amount of energy is being misused through inefficient CPU usage, especially as Android-based devices accounted for 74% of global smartphone sales in Q1 2013.

This project aims to evaluate the effectiveness of multicore use with the Android platform by performing various microbenchmarks on multicore mobile devices through alternative threading implementations. By building prediction models, I investigate in detail, the effectiveness of multithreading in Android, in terms of performance and energy consumption.

I demonstrate that the choice of threading configuration when developing mobile apps can impact both performance and energy consumption by an order of magnitude, and also that an incorrect threading configuration can even be detrimental to effectiveness, due to the power overhead of activating a second core. Additionally, I show that developers must explicitly make use of all of the multiple physical cores by creating additional threads in their app source code, otherwise the benefits of a multicore processor are not realised.

I make the recommendation that, when offloading work to multiple threads, the number of threads developers should use varies depending on whether they require better performance, or lower energy consumption.

Acknowledgements

I would like to thank my project supervisor, Dr Andrew Rice, for his knowledgeable advice and continued feedback.

Contents

Chapter 1

Introduction

To meet the demand for more processor intensive usage patterns in mobile phones, manufacturers have been required to include more powerful mobile processors in their devices. To do so, processor clock frequency was increased for several years, as demonstrated by Moore's Law with desktop processors, but heat and power concerns resulted in a frequency limit being reached with a single core. In the consumer desktop market, the number of cores per physical processor was increased to allow for parallel computation, rather than running a single core at a higher clock frequency. System-on-a-chip designers, such as ARM, and manufacturers including Qualcomm, Texas Instruments and Nvidia, recognised the potential of multicore processors for mobile devices in terms of enhanced performance and, more importantly in the mobile area, lower power consumption. The first multicore mobile device announced commercially for Google's Android platform was the LG P990 in 2011 [1], containing an Nvidia Tegra 2 SoC with a dual-core processor. The extent to which multicore processors in mobile devices benefit the consumer is a much discussed topic today.

In this report, I analyse the effectiveness of multicore processors in modern smartphones for the purposes of faster performance and reduced energy consumption. I build performance models that estimate the completion times of a batch of work once allocated to multiple threads, and a power model to estimate energy consumption of a device, when work is shared between multiple threads. These multithreading prediction models can be applied to different devices, and types of work. I also demonstrate that the choice of

[1]http://www.engadget.com/2010/11/16/exclusive-lgs-4-inch-android-phone-with-dual-core-tegra-2-and/

threading configuration when developing mobile apps can impact both performance and energy consumption by an order of magnitude, and that an incorrect threading configuration can even be detrimental to effectiveness, due to the power overhead in activating a second core. I make the recommendation that, when offloading work to multiple threads, developers should use the same number of threads as the amount of physical CPU cores in the device for the fastest performance, and that they should use the amount of physical cores, plus one for reduced energy consumption.

The significance of mobile phones in computing is rising consistently — there are 6.835 billion devices in use as of February 2013. Of these devices, smartphones are expected to have a 17.9% growth in sales between 2012 – 2016[2]. Therefore, the energy efficiency of smartphones is a critical area of computing research for the benefit of overall reduced energy consumption in technology.

Mobile phone manufacturers are limited by current Lithium-Ion battery technology in improving battery life. Battery capacity could be increased, but the small form factor of mobile devices prevents larger batteries being used. Thus, the most effective way for them to improve battery life of a mobile device is to reduce the amount of power that the device consumes. To do so, hardware and software optimisations are used. An example of a hardware optimisation was the introduction of multicore processors to mobile devices. It has been argued that quad-core processors present a very real benefit to battery life, as multiple cores working at a quarter of their capabilities can consume less power than one core being worked to its maximum[3].

However, the General Manager of Intel's Mobile and Communications Group, Mike Bell, recently claimed that Android is not ready for multicore processors[4]. He says that it "isn't entirely clear you get much of a benefit to turning the second core on", due to an ineffective implementation of thread scheduling resulting in a power overhead that is not used to improved performance.

Therefore, a key issue with multicore processors is that work has to be allocated to the processor efficiently by an effective thread scheduler, and developers must write code that implements multiple threads in order for the performance and power consumption benefits to be acquired.

It is the responsibility of the threading implementation in the smartphone

[2]http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a#subscribers

[3]http://www.wired.com/gadgetlab/2012/05/quad-core-vs-dual-core-phones-tablets-nvidia-samsung-galaxy/all/

[4]http://www.theinquirer.net/inquirer/news/2182710/intel-claims-android-ready-multi-core-processors

operating system to provide developers with a solid platform to create multithreaded applications. The most widely used smartphone operating system is Android. Developers create mobile applications through an SDK provided by Google. Their applications run on a device through the Dalvik Virtual Machine. The Dalvik VM provides threading support through the underlying Linux kernel in Android [1]. However, it is unclear how efficiently the Java byte code of a multithreaded application is translated into instructions that are recognised by the CPU architecture of a mobile device, particularly as Android devices are fragmented, containing many different hardware configurations.

I build a series of microbenchmarks that are designed to demonstrate the effectiveness of multicore processors in mobile devices, based on two bechmarking algorithms that are executed through several different threading configurations, on both a dual-core device and a quad-core device. The variables contributing to the threading configuration include: the number of threads that are running the microbenchmark, the amount of physical cores, the batch type of workload and the threading runtime. My study reveals that developers have to explicitly make use of the physical cores, by creating threads in their app source code. It shows that, in Android, work is not spread over the physical cores when using only the main UI thread with no additional threads. This is a result that I did not expect, and demonstrates that further work is needed to the Android operating system in order to spread work that is parallelisable, between physical cores for better performance and reduced energy consumption. Based on my results, I recommend that developers use the same number of threads as the amount of physical CPU cores in the device for best performance, and that they should use the amount of physical cores, plus one for reduced energy consumption.

In future work, I would use my findings to create a database of devices describing their most effective threading configurations. I would create an API for use by developers, allowing them to wrap any processing work into the correct multithreading configuration, based on the runtime device.

Contained in this report are the following primary contributions:

- I formulate two models to be used for predicting performance by a mobile device when multiple threads are used. The first can be used for predicting performance with a fixed amount of work per thread, and the second for a fixed total amount of work. These models can be applied to different hardware and types of algorithm.

- I create a model of power that describes the prediction of energy consumption when multiple threads are used, for a fixed total amount of work, shared between threads.

- I show that as soon as additional threads are used, there is a large power overhead in using multicore, but energy used is often reduced due to faster completion times. Additionally, I demonstrate that developers have to define additional threads in their app source code to make use of multiple physical cores.

- I recommend that developers use the same number of threads as the amount of physical CPU cores in the device for best performance, and that they should use the amount of physical cores, plus one for reduced energy consumption.

Chapter 2

Related Work

In this chapter, I review related work to validate the proposition that *reducing the energy consumption of mobile devices is a critical issue, that can be lessened by the effective use of multicore processors.*

Previous work demonstrates the critical issue of energy consumption in mobile devices. It has been argued that the practicality of mobile devices is weakened by limitations in battery life [2]. Either reducing energy consumption or increasing battery capacity can improve battery life. Manufacturers cannot increase battery capacity by using larger batteries due to the constraints of small form factor devices. Additionally, the current, most used smartphone battery technology, Lithium-Ion, is not expected to undergo modifications to increase battery energy density [3]. Therefore, it is crucial that methods are found to minimise energy consumption in mobile devices, because it is doubtful that battery capacity will increase in the near future. To minimalise energy consumption, optimisations need to be made to both hardware and software [4]. Apps can be designed to reduce the energy consumption of mobile devices [5], but this is uncommon, thus over a third of user complaints regarding battery life are due to inefficient apps, known as ebugs [6].

Using the correct threading implementation to make use of multicore hardware effectively is an example of a software optimisation to reduce energy consumption. Dynamically reducing the quality of an app is another example of a software optimisation used to reduce energy consumption. An app can be designed to observe power demand and react to a low level of battery life [7]. An example of this would be a video streaming app recognising a low level of battery life and subsequently reducing the bitrate of the downloading video, resulting in reduced energy consumption by the radio communication hardware component. Additionally, the time of which a sensor is powered on can be shortened, thereby reducing energy consumption, by using middleware that aggregates requests for updates from sensors [8]. Another energy saving method through software involves changes to web browsing [9]: the quality of downloaded images can be reduced through a proxy, and the use of custom mobile stylesheets can reduce the energy consumption by the device components. For example, making use of black backgrounds greatly reduces the energy consumption of AMOLED-based displays.

We need to understand the performance of multicore processors in mobile devices to learn about energy consumption, as they can be directly proportional. Thus, we can examine the previous work undertaken for understanding performance prediction with desktop multicore processors. Multilayer neural networks can be trained using input data from executions on a target platform to predict multicore performance [10]. The advantage of this method in performance prediction is that it captures full system complexity. The parallel benchmarking application SMG2000 is used to analyse the accuracy of this model, and an error within 5%-7% is found.

Additionally, the relative performance between two platforms can be used for cross-platform multicore performance prediction [11]. Using this method, the authors argue that relative performance can be predicted without executing a single parallel application in full. This is due to their assumption that most parallel code is iterative, and will behave predictably after the serialised startup period. The accuracy of this method was analysed, and there was an average prediction error between 12% and 26%.

Due to the the increased energy constraints of mobile computing that are less significant in desktop computing, there is more of an emphasis on performance prediction rather than energy consumption prediction with desktop multicore processors, and little energy prediction work exists. However, we can apply some of the concepts used in these multicore performance prediction models to mobile multicore performance and energy consumption, as performance is proportional to energy used.

The most direct method of understanding energy costs in a modern mobile device is by recording the power consumption. Fine grained measurements cannot be taken using the remaining charge level on the device as given by the battery [2]. Energy consumption can be calculated by using a power model to predict power consumption, or by recording power measurements in from physical device.

The first approach using power models can be achieved by using a fine grained model based on system call traces to model a device as a finite state machine [12]. Developers can also be given a per component approximation of energy use with thw Nokia Energy Profiler [13]. Furthermore, alterations in voltage across the battery could be used to estimate battery use, and energy consumption could be linked with the currently running app, as demonstrated by the PowerBooter app for Android [14].

The second approach in obtaining power data to calculate energy used is through recording measurements in a physical device with power measurement equipment. This method supplies accurate, high precision power data that can be used in calculating energy consumed. The energy used by each device component can also be measured from power rails, but due to the small size of the chipset boards, this is not possible in modern, comercially available mobile devices [15]. Thus, the energy consumption for the entire device can be precisely calculated with synchronised power measurement equipment.

An external power supply can be used to power a device in measuring setup in order to calculate energy used [14]. I improve on this by powering the devices with their own batteries, reducing a source of variance.

To understand performance and therefore energy consumption, we need a way to evaluate parallel CPU performance. Benchmarking tools are a requirement in evaluating parallel performance [16]. SPEC , who defined the CPU2000 benchmark suite for the purpose of evaluating a CPU chip's performance, argue that "if performance depended only on megahertz, there would be no performance differences between two identical chips, and a faster chip would always win".

I will be comparing the performance and energy consumption of two threading runtimes, Java threads run through the Dalavik Virtual Machine, and native pthreads through JNI/NDK. Native C code compiled with the NDK and executed through JNI is up to 30 times faster than the equivalent Java code through the Dalvik Virtual Machine [17]. To my knowledge, my study is the first where it is possible to compare the energy consumption of native code with Java code through the DVM.

In this chapter, I have demonstrated the importance of reducing energy consumption in mobile devices, and examined how previous work into desktop multicore performance and energy consumption prediction can be applied to mobile multicore processors. Effectively making use of mobile multicore processors for the purpose of energy saving is currently not fully understood, thus it is most appropriate to measure the effectiveness of mobile multicore processors by using physical devices to improve our understanding.

Chapter 3

Benchmarking effectiveness of multicore processors in smartphones

In this chapter, I create a series of microbenchmarks designed to demonstrate the effectiveness of multicore processors in mobile devices. I define multicore performance prediction models, and an energy consumption prediction model for multithreaded work. Through using the results of the microbenchmarks, I show that my mobile multicore performance models are accurate, and that power consumption can also be predicted. By analysing the performance and energy consumption of the devices as the number of threads is increased and the threading environment is changed, I demonstrate that the choices made whilst programming mobile apps involving multiple threads can drastically affect both performance and energy consumed.

In order to do so, I will measure completion times to evaluate performance, and will record power readings during my testing to evaluate energy consumption. I also introduce several variables to my experiment, aiming to cover the possible design choices that a developer can make when programming a multithreaded mobile app.

The first variable in the experiment is the number of threads that are running the microbenchmark. The purpose of adding this variable is to discover if the operating system successfully allocates the increasing number of software threads to the physical cores.

I will also be varying the number of cores, through the use of two devices containing CPUs with a different amount of cores. This variable will help in supporting my conclusions, as any patterns that emerge from the results from both devices with a varying number of cores, are more valid and the patterns are likely to be found in other modern devices. As modern mobile devices contain an SoC with a single physical processor, this study also only considers multicore processors within a single die.

Thirdly, I will be changing the workload type. The microbenchmarks will be repeatedly executed for a set number of times per thread. This will depend on whether the total workload is varied or fixed. For the varied total workload, each thread has a fixed amount of work, and the average completion time for the microbenchmark will be calculated, allowing us to find out whether having multiple cores working concurrently affects the independent performance of a single execution. For the fixed total workload, each thread is allocated a varied amount of work as the number of threads is increased, and work is shared between all running threads. This allows us to find out whether it is advantageous in terms of performance and energy consumption to offload a large batch of work to multiple threads.

Finally, different threading runtimes will be used to find out if Android handles the allocation of software threads to physical cores differently depending on the use of Java code, or native C code. The differences on both performance and energy consumption between the two threading runtimes will be analysed.

In the following sections, I cover the outline of the benchmark testing in further detail. Firstly, I will consider the two alternate threading runtimes that will be used to run the microbenchmarks.

3.1 Alternate threading runtimes

In mobile software development, the implementation of threading is a common method used to perform concurrent execution. Concurrent execution is perceived in a single core system, due to rapid time-division multiplexing between the different threads. Truely concurrent execution can only occur with a multicore processor, but only when the operating system effectively schedules threads to physical cores.

Developers can create several threads, and define each one to have a different set of instructions to perform. A common use for threading is to allow one thread to handle a user interface, whilst the others perform some computation. This allows the user interface to still be responsive to the end user, as the dedicated thread waits for and handles any user input.

Developers can implement multithreaded apps for Android in two ways, using Java threads through the **Dalvik Virtual Machine**, or native pthreads through the **Native Development Kit**.

Dalvik Virtual Machine

In Android, all apps are allocated their own instance of the Dalvik Virtual Machine at runtime. Each instance of the DVM has one main thread running when it is started, known as the UI thread. There could also be a few *housekeeping* threads for that instance, that start and end periodically, managed by the Android system, but these perform proportionally minimal processing when compared to my microbenchmarks, thus do not affect the results of this study. Developers therefore have control of the instructions running on the main UI thread, along with any additional threads that they create. These additional threads can be made by creating a new instance of the `Thread` class, and implementing its `run()` method. Each thread has its own call stack for any methods being invoked, along with their arguments and local variables. Threads active within the same instance of the Dalvik VM may communicate through the use of shared objects.

Code 3.1: Java code to create a new thread

```
new Thread() {

        public void run() {
                //code to run in new thread..
        }
}.start();
```

Native Development Kit

The NDK is an add-on for the Android SDK that allows developers to add libraries written in C to their Android apps. Using the NDK, the libraries are compiled down to ARM, MIPS or x86 native code. Native classes can be called from within the Dalvik VM using the `System.loadLibrary` method, that uses the Java Native Interface. Google recommends that the NDK should only be used performance critical code that is CPU intensive, specifying that “the NDK will not benefit most apps”[1].

C code produced and compiled with the NDK can also make use of multi-threading, through a POSIX standard for threads. POSIX defines an API for creating and manipulating threads, known as *pthreads*, and it is bundled with most Linux distributions, including Android.

Code 3.2: C code to create a new Pthread

```
#include <pthread.h>

static void *thread_fn(int arg) {

        //code to run in new thread..
}

int main(void) {

        int thread_arg = 0;
        pthread_t new_thread;

        int r = pthread_create(&new_thread, NULL,
                        thread_fn, thread_arg);

        assert(r == 0 && "failed pthread_create");

}
```

All apps developed for Android must run through the Dalvik VM for the UI. Thus, an app cannot be executed through the Android launcher on the device if it has been created purely with native code, and can only be called from an instance of the Dalvik VM through JNI. The only effective use of native code is for offloading any large processing jobs away from the Dalvik VM, for faster execution.

Dalvik VM threads follow the pthread model, meaning that any Dalvik threads are treated as native pthreads by the Android system [2]. Although the Dalvik threads themselves are seen by the Android system as native threads, the code that is being executed on them is still Java bytecode through the Dalvik VM.

[1]http://developer.android.com/tools/sdk/ndk/index.html

[2]Git clone via http://source.android.com/source/downloading.html

My experiment is carried out using both threading environments, in order to find out if there are differences between the environments in terms of multicore utilisation. As native code has already been proven to be faster, any performance comparisons made in my results will be relative to the single threaded execution in the respective environment. This research is however, the first to compare the amount of energy consumed from the battery when the same algorithms are executed through either native code or the Dalvik VM.

3.2 Multicore benchmarking algorithms

Benchmarking is used in computing to assess performance of a particular system, by running a series of tests against it. A *benchmark* can be made to assess the hardware in a computing configuration, such as the CPU, or to assess the software, such as the performance of a new compiler.

Benchmarks are beneficial for evaluating performance because they allow the tester to analyse the real performance of the object being tested. Without a benchmark, a tester could review the detailed specifications of the analysed object, but it would be difficult for them to picture real performance of the object due to the number of factors involved that contribute to performance. For example, a CPU may have a high clock frequency, but a low amount of instructions per cycle, resulting in lacklustre real performance when compared to a CPU with a mediocre clock frequency and mediocre instructions per cycle. An example of this occurred with desktop processors, where Intel Pentium 4 processors operated at a higher clock frequency than the competing AMD Athlon XP processors, but real performance was on par.

There are several types of benchmarks. *Real program* benchmarks record the performance of actual applications that were not created purely for the purpose of benchmarking. For example, recording the time taken for a word processing software package to find a spelling mistake in a large document. *Microbenchmarks* are designed to measure the performance of a small portion of code. *Kernel* benchmarks use low-level code, abstracted from the actual benchmarking program, to measure the basic architectural features of a machine. *Component* benchmarks are programs designed to measure the performance of a computer's basic components, such as the CPU. *Synthetic* benchmarks are written according to statistics that show the types of operation a group of applications perform. According to these statistics, the benchmark is written to use the same proportion per type of operation. For example, if a group of applications used floating-point arithmetic operations 60% of the time, the synthetic benchmark based on these applications would perform the same proportion of floating-point arithmetic, in relation to the total operations performed by the benchmark. Examples of synthetic benchmarks include the Whetstone benchmark, that primarily measures floating-point arithmetic performance, and the Dhrystone [18] benchmark, that measures integer arithmetic and string operation performance. *I/O* benchmarks measure the performance of input/output, such as the speed in Mbps that a single large file can be read from a hard drive, and transferred to main system memory. *Database* benchmarks are designed to measure the throughput and response times of database management systems. *Parallel* benchmarks are designed to evaluate performance of machines with multiple cores, separate physical processors or distributed systems consisting of multiple machines. The benchmark performs the series of tests accross the multiple cores capable of truely concurrent execution.

Having studied the different types of benchmarks, we can define the benchmarks that I perform in this study as component microbenchmarks, in that they are focussed on evaluating the performance of a single component, the CPU, by using a small portion of code. They are also classed as parallel microbenchmarks, as concurrent execution of the small portion of code will take place across the multiple CPU cores present in the mobile devices. Using these benchmarks in several configurations, I will analyse the effectiveness of multicore processors in Android devices, in terms of performance, and energy used. The benchmarks that I perform are based primarily on two algorithms: an **altered sleep**, and the **Fibonacci** algorithm. Both of these algorithms have external validity, as the types of CPU operation that they perform are commonly used in real world apps — Section 3.2.3 has further details.

3.2.1 Altered sleep

To measure utilisation of physical CPU cores, a `sleep` function, that accepts a sleep interval time as a parameter, can be invoked within software threads. By analysing the completion times, we can identify the performance and energy costs of concurrent execution when compared to a single threaded run of the interval time. This will show how truely concurrent the multicore processor can be, as any completion times above the interval time parameter will demonstrate ineffective allocation of software threads to the physical cores.

However, I cannot use built-in `sleep` functions in this study for three reasons:

- A built-in `sleep` function has more interactions with the operating system, when compared to a function defined within the test program. Thus, there is an increased potential of variance in my results, due to I/O delays where multiple threads attempt to access the shared operating system resource.
- To evaluate the effectiveness of multicore mobile processors from the perspective of a developer, I need to perform analysis using code that follows the *same path to the physical cores*, as code defined by a developer. The implementation of a built-in `sleep` does not do so, as it interacts with the thread scheduler directly to put the current thread into a wait state for the required interval[3]. Additionally, this wait state would drastically reduce power consumption when it occurs within a modern processor, making it difficult to define the testing period in a power trace during my analysis.
- I am performing a controlled experiment, where fairness between devices containing different hardware should be ensured. In using built-in `sleep` functions, it is not ensured that each device will complete the same amount of work.

For these reasons, I define an *altered sleep*. Timing loops are used to increment a variable a set number of times. This amount is adjusted until the desired sleep interval is obtained. Thereafter, the variable is not used, to ensure that no unnecessary computation occurs. However, in doing so, modern compliers recognise that the variable is unused, and the incrementing operation that causes the sleep period is optimised away. There are methods to remove this compiler optimisation, in order to ensure that the computation required to sleep still occurs. In C, the gcc specific `noinline` attribute is used - without inlining, gcc does not recognise that the incremented variable is not needed. In Java, the `volatile` keyword is used to avoid the sleep period being completely optimised away by the compiler. The same parameter value is to be passed to the altered sleep function with all test devices.

Code 3.3: The C implementation of the altered sleep.

```
static __attribute__ ((noinline)) int optimised_sleep(int t) {
        volatile int i, j;
        int x = 0;
        for (i = 0; i < t; i++) {
                for (j = 0; j < 100000; j++) {
```

[3]http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html

```
                    x += i + j;
                }
            }
            return x;
}
```

The implementation of the altered sleep microbenchmark is dependent on integer operations. I expect the results from the altered sleep microbenchmark to vary between devices, based on the performance of integer operations for the CPU in the device. Therefore, the results will be used in evaluating multicore performance, by analysing how concurrent execution affects the defined interval time. We expect the completion time of a single thread to equal the defined interval time. Hence, increased completion times as the number of threads are increased up to and including the amount of physical cores will demonstrate inefficient thread scheduling, and poor concurrent integer operation performance.

3.2.2 Fibbonacci

To analyse whether shared caches during concurrent execution can affect performance and energy consumption, and how each core uses L1 CPU cache, it is beneficial to perform a microbenchmark that creates a large stack size per thread. This is important because ineffective shared use of L1 cache can impact performance, as one thread waits to read from the cache whilst another thread writes to it. One method of creating a large stack size is to perform numeric calculation with recursion. In doing so, I can analyse the stack memory management performance per core, and the cost of making recursive function calls concurrently.

The Fibonacci algorithm is used heavily in benchmarking desktop CPU performance, due to its ability to create a deep recursive stack. It has been used in previous work for benchmarking the overall CPU performance on the Android platform [19], but this study will consider performance and energy consumption as the algorithm is computed concurrently using multiple CPU cores.

Fibonacci numbers can be calculated recursively as follows:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

A single computation of `fib(n)` can can be computed concurrently, whereby *fork/join* is used to share work [4]. However, computing a Fibonacci sequence is a recursive algorithm that intrinsically has dependencies on prior calculations. This reduces the effectiveness of concurrent execution — the dependencies on prior calculations could cause blocked tasks and actually serialise computation. This concurrent implementation is therefore not likely to present reproducible results. Thus, in this study, I will ensure that each `fib(n)` computation is only calculated within one thread. My analysis will focus on concurrent separate computations of `fib(n)`, within multiple threads.

For my benchmarks, I will use the highest `n` value possible, that allows execution to complete on all test devices without the occurence of a recursive stack overflow exception - caused by a full stack in one of the executing threads. If this exception occurs, my results will be incomplete.

3.2.3 Summary of benchmarking algorithms

In the previous sections, I have given an overview of the two benchmarking algorithms to be used for my microbenchmarks, that will be executed in several threading configurations to evaluate the effictiveness of multicore processors in modern smartphones.

Although these benchmarking algorithms are unlikely to be used in *real world consumer* apps, they represent typical CPU operation types. The altered sleep utilises integer operations, whilst the Fibonacci algorithm makes use of deep recursion, that results in a large stack per thread and illustrates the performance cost of concurrent memory management. *Real world consumer* apps make extensive use of these types of CPU operation, any iterative loop uses integer operation, and various functions defined with the Android operating system, such as `XmlRpcHelper` that can invoke other helpers recursively [5]. Therefore as the microbenchmarks also use these types of CPU operation, the results of the study are externally valid.

Additionally, for the purposes of this study, where we are evaluating the effectiveness of multicore processors in modern smartphones, working the independent CPU cores to a high capacity is the priority. This allows us to analyse relative performance between the multiple cores, the efficiency of the Android thread scheduler, and the energy consumption of the entire devices as the amount of CPU cores in use increases.

[4]http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html

[5]http://source.android.com/reference/com/android/tradefed/util/net/XmlRpcHelper.html

3.3 Variation of workload type

In order to test the effectiveness of the multicore CPUs in the devices, it is benefical to produce a benchmarking tool that varies the work pattern. A *variable* total work amount, or fixed work per thread, is used to evaluate whether the average completion time changes across all work for a single execution of the microbenchmark, as the number of threads is increased. A *fixed* total work amount, or variable work per thread, is the more *realistic* pattern, whereby several threads will work together to compute total work.

3.3.1 Variable

With the variable workload, as the number of threads, n, executing the algorithm concurrently is increased, the total work performed, t, also increases. Each thread executes the microbenchmark 100 times, work per thread, p, is fixed.

$$p = 100$$
$$t = p \times n$$

For example, when 3 threads are concurrently executing the microbenchmark, the app records 300 completion times, 1 for each time that the microbenchmark ran. By carrying out this variation of total work, I can find out the average completion time for the microbenchmark, for each time the number of threads is increased. Section 3.4.1 presents a model for performance as the number of threads concurrently executing the microbenchmark is increased, in proportion to the number of physical cores in the device.

3.3.2 Fixed

With the fixed workload, as the number of threads, n, executing the algorithm concurrently is increased, the total work performed, t, stays constant at 400. The total work is effectively spread over all threads. The work per thread, p, varies depending on number of threads.

$$p = t/n$$
$$t = 400$$

For example, when 3 threads are working concurrently to execute the microbenchmark 400 times, the app only records 1 completion time, for the total time the benchmark took to run. By carrying out this variation of total work, I can find out how efficiently the threads can work together to achieve the same collaberative goal, as the number of threads is increased. Section 3.4.1 presents a model for performance prediction, and Section presents a model for energy consumption prediction for this type of workload.

3.4 Multicore prediction models

In this section, I create three multicore prediction models: two for performance prediction, and a model for prediction of energy consumption.

3.4.1 Modelling multicore performance

I define two models of performance prediction for multicore performance. The first model is good at predicting performance for the variable total work load, whilst the second is accurate at predicting performance of the fixed total work load. By illustrating correspondance between measured values and predicted values, I use my models to show how effictively multicore processors are used for performance in modern smartphones.

Model for variable total work/fixed work per thread

To model multicore performance for a fixed batch of work per thread, we begin by assuming that completion time per execution T_n, remains static at the completion time of a single thread T_1, as we increase the number of threads of execution, n:

$$T_n = T_1$$

However, once multiple threads are in use, the performance overhead demonstrated by Armdahl's Law [20] must be considered. Thus, the overhead for the portion of the algorithm that is not parallelizable, $0 \leq S \leq 1$, is added. Therefore:

$$n > 1 \Rightarrow T_n = T_1 \times (S + 1)$$

Where the number of threads of execution exceeds the amount of physical CPU cores C_n, the additional thread(s) work is spread over all of the physical CPUs cores, revealed by analysis of the CPU usage per core measurements in Appendix C. Hence:

$$n > 1 \wedge n \leq C_n \Rightarrow T_n = T_1 \times (S + 1)$$

$$n > 1 \wedge n > C_n \Rightarrow T_n = (n \times \left(\frac{T_1}{C_n}\right)) + (T_1 \times S)$$

For my microbenchmarks, the value of S was estimated to be **0.05**, where 5% of the code was serial when the Dalvik Virtual Machine starts and the single main UI thread initialises. It is not practical for the purposes of this study to find out the true value of S in the Android operating system, as a substantial amount of code is run prior to work beginning, due to a new instance of the DVM being created. The expected completion times through using this model against the actual measured completion times are shown in Section 3.6.1.

All tests were found to be normally distributed, and in subsequent analysis sections, a median average was used where it was required, in order to avoid inclusion of extremitities. Figure 3.1 shows an example of the distribution of measured values for a variable total work benchmark test.

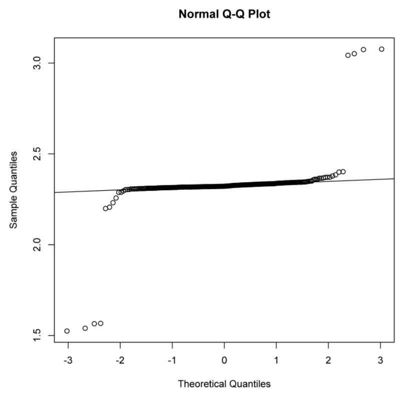


Figure 3.1: The normal distribution plot for a Galaxy Nexus altered sleep test, running with 4 threads through the DVM. Through the variable total work benchmark, the average completion time for the microbenchmark amongst all threads that concurrently ran it can be calculated. The median in this plot is the value at 0 on the x-axis.

Model for fixed total work/variable work per thread

To model multicore performance for a fixed amount of total work, where each thread works towards a shared goal, we begin by assuming that total completion time T_n is calculated by dividing the completion time of a single thread T_1, by the number of threads of execution, n.

$$T_n = T_1 \times \frac{1}{n}$$

Once multiple threads are in use, Armdahl's Law [20] specifies that there is a performance hit. Thus an overhead for the portion of the algorithm that is not parallelizable, $0 \leq S \leq 1$, is also applied:

$$n > 1 \Rightarrow T_n = T_1 \times \left(S + \frac{1-S}{n} \right)$$

Where the number of threads of execution exceeds the amount of physical CPU cores C_n, the additional thread(s) are divided between the physical cores, but each thread computes less. This results in no additional effect on completion time. In this instance, the completion time is equal to the completion time that occured when the numeber of threads of execution equalled the amount of physical cores. Hence:

$$n \leq C_n \Rightarrow T_n = T_1 \times \left(S + \frac{1-S}{n} \right)$$
$$n > C_n \Rightarrow T_n = T_{C_n}$$

CPU clock frequency modifications

When CPU core speed is modified so that two threads work towards a shared goal at half the speed per thread when compared to a single thread at full speed, completion time should remain the equal. For example, one thread executing the microbenchmark 400 times on a core running at 700MHz should take the same amount of time as two threads executing the microbenchmark 200 times each, on two cores running at 350MHz each. Section 3.7.4 analyses the measured values with regards to these tests.

3.4.2 Modelling multicore energy consumption

In this section, I define a prediction model of energy consumption when multiple CPU cores are in use. The model is to be used for predicting energy consumption of a mobile device for the fixed total work load. Using this model, developers could find out how much energy will be used by a device when they offload processing work to multiple threads in their apps. By illustrating correspondance between measured values and predicted values, I use my model to show how effictively multicore processors are used in modern smartphones when energy consumption is considered.

To model multicore energy consumption for a fixed amount of total work, where each thread works towards a shared goal, we first make the assumption that energy used during the testing period, E_n, is proportional to the total completion time of the test, T_n:

$$E_n \propto T_n$$

Following the definition of T_n from Section 3.4.1, we can predict total completion time of the test in seconds. If we recall that $1 watt = 1 joule/second$, the energy used during the testing period is equal to the average watt usage over the testing period for a single thread, W_1, multiplied by the total completion time. Therefore:

$$E_n = W_1 \times T_n$$

However, I expect a significant cost in power consumption when more than one core is in use, with both devices. Therefore the average watts used over the testing period for a single thread is magnified by a *multicore multiplier*, M, when the number of executing threads, n, is greater than one:

$$n = 1 \Rightarrow E_1 = W_1 \times T_1$$
$$n > 1 \Rightarrow E_n = (M \times W_1) \times T_n$$

We assume that energy consumption will remain constant when the number of threads, n, exceeds the amount of physical CPU cores, C_n. This is on the basis that power consumption should remain constant when this occurs, as the CPU is being fully utilised when the number of executing threads equals the amount of physical CPU cores. Additionally, as the total completion time is constant when $n > C_n$ in the prediction model from Section 3.4.1, T_n remains constant in this energy prediction model when the number of threads exceeds the amount of physical cores.

3.4.3 Summary of multicore prediction models

In this section, I have defined two models of performance prediction, one for the variable total workload, and one for the fixed total workload. I have also defined a model for prediction of energy consumption for the fixed total workload.

These models are used in the following analysis sections to consider whether the measured values corresponded to the expected values, in order to evaluate the effectiveness of multicore processors in modern smartphones.

In Section 3.6.1, the measured values are mapped against my performance prediction models, and in Section 3.7.1, the measured values are mapped against my energy consumption prediction model.

3.5 Microbenchmarks experimental setup

In this section, I describe the experimental setup and power modelling of a dual-core Samsung Galaxy Nexus, and a quad-core LG Nexus 4, both running the Android mobile operating system. Although my experiment considers just these two devices, the results should show a pattern that applies to other Android handsets, with similar ARM-based hardware. Due to the effects of the *silicon lottery*, I find small performance differences between devices of the same model. The device configuration defined in this chapter mitigates the effects of the differences between devices to the extent that they are not significant enough to affect any patterns emerging from the results. The study of mobile device performance and power consumption in reality is not straightforward: they are both affected substantially by many factors, that are difficult to discover and modify to the advantage of the experiment. I consequently perform an experiment with high internal validity, where the factors that affect performance and power consumption of the mobile devices are controlled.

The hardware components of both the mobile devices used in this experiment are shown in Tables 3.1 and 3.2. Of particular note is the larger, directly mapped L1 CPU cache in the dual-core Galaxy Nexus. The quad-core Nexus

4 shares half the L1 cache over four cores through 4-way set associative, which could result in unexpected results in the benchmarks for the higher clocked Nexus 4. With a set associative cache and unlike direct mapped cache, a cache block is only available after a cache hit/miss. This means that if there is a cache miss, there is an additional overhead to recover from with a set associative cache, which could introduce slower performance for any benchmarks that rely heavily on CPU caching. Both of the devices contain symmetric design multicore CPU chips, each core is of equal compuational power. For recording performance, my benchmarking app records completion times to a file, please refer to Section 3.5.2 for further details.

For measuring the power consumption of the devices, a 0.05Ω resistor is placed in series between the device's battery and the device, along with two power supplies, each delivering 3.7V of electrical potential. A National Instruments compactRIO DAC meter is used to record the current and voltage through the resistor, and therefore, the power consumed by the device from its battery. The voltage drop is measured at 250kHz by the meter, and sent through ethernet, to a a computer, which downsamples the data to 10kHz in order to decrease any sampling error. The data is recorded to a binary file, as double precision floating point values. This setup has been used in a previous experiment [21], with the exception of the use of two power supplies at 3.7V supplying compared to two 3.7V batteries. The two power supplies are only used for measuring power across the resistor, as such, the devices are only supplied power through their own batteries, which improves on previous work [14] where an external power supply was used to power the device, rather than using its own battery. This method could affect results due to inconsistencies in the device's external power source, whereas by using the device's own battery, I improve the external validity of the experiment. To calculate the energy used by the device during a test, a script is executed that parses the binary file, and detects the start and end of the test. Figure 3.2 illustrates the experimental setup.

3.5.1 Device configurations

Both of the devices used in the experiment are "Nexus" devices, and run a stock version of Android 4.2.2 Jelly Bean. Although this version of Android contains few running services when compared to the versions installed on other non-Nexus Android devices in the commercial market, many of these services are not required for the purposes of my experiment, and will result in unnecessary "spikes" of power use and performance slowdowns as the

Component	Description
CPU	1.2 GHz dual-core ARM Cortex-A9
Floating Point Unit	VFPv3-D16 (typical), VFPv3-D32 (pipelined)
ARM NEON	Yes, 64-bit wide
L0 Cache	N/A
L1 Cache	32 kB + 32 kB
L2 Cache	1 MB
RAM	1 GB
Operating System	Android 4.2.2 Jelly Bean
Flash storage	16 GB
SoC	Texas Instruments OMAP 4460
Battery	1750 mAh, 6.48 Whr, 3.7V Lithium Ion

Table 3.1: Hardware specifications of Galaxy Nexus.

Component	Description
CPU	1.5 GHz quad-core Qualcomm Krait
Floating Point Unit	VFPv4 (pipelined)
ARM NEON	Yes, 128-bit wide
L0 Cache	4kB + 4kB direct mapped
L1 Cache	16 kB + 16 kB 4-way set associative
L2 Cache	2 MB 8-way set associative
RAM	2 GB
Operating System	Android 4.2.2 Jelly Bean
Flash storage	16 GB
SoC	Qualcomm Snapdragon S4 Pro APQ8064
Battery	2100 mAh, 8.00 Whr, 3.8V Lithium Polymer

Table 3.2: Hardware specifications of Nexus 4.

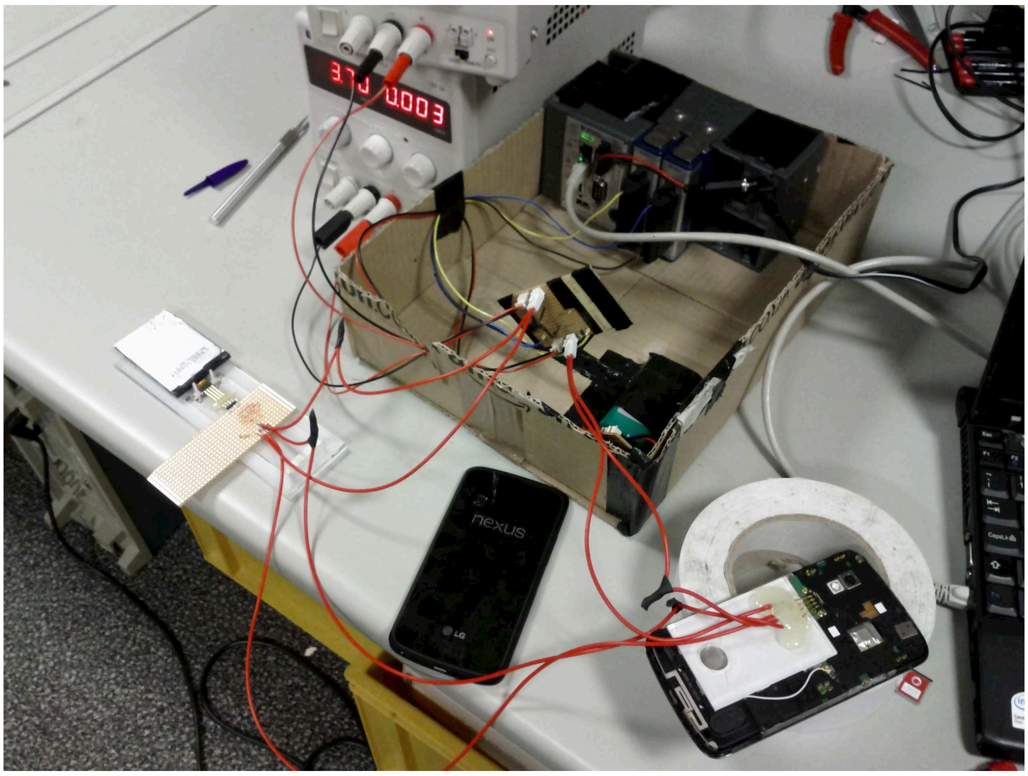


Figure 3.2: The LG Nexus 4 in the custom made power measurement equipment. The original battery is used in the setup to measure voltage across a 0.05Ω resistor. The setup adds minimal resistance when compared to the battery being directly in the device, due to short wires. More photos are shown in Appendix B

CPU cores perform other operations. For this reason, the services running on the devices were reduced to the minimum necessary in order to run the microbenchmarks. Additionally, the CPU clock frequency and governor were fixed at the desired settings. For thorough device configurations, please refer to Appendix A.

3.5.2 Test client

All benchmarks were performed through a client benchmarking Android app, that runs on the devices. The app is autonomous once running, and is not operable from the device itself, so that no unnecessary CPU time is taken for drawing a user interface. For this reason, the parameters required by the benchmark are to be modified in the source code for the app. Once a test is ready to run, the code is packaged and the APK file is copied to the device, where the app starts.

To begin, the app sleeps for 8 seconds, time to allow for the removal of the USB cable, which charges the device when it is plugged into a computer, affecting power consumption readings. The app then turns off the screen, which would also increase variance in the readings as it consumes a high proportion of the total energy consumed when it is turned on. Next, the app takes a *partial wakelock* to ensure that the CPU is not in sleep mode. Finally, the CPU clock frequency is set by the app to the speed set in the source code. This is done to ensure that the device's CPU frequency is at the desired speed prior to the test beginning, as the act of turning off the screen in Android forces the CPU into the "sleep" frequency, even when the partial wakelock is taken and the *performance governor* is set in *CPUfreq*, the Linux kernel's CPU scaling subsystem.

The app then starts the microbenchmarks according to the parameters set in the source code, such as the number of threads to use, the threading environment, the algorithm to run and the total work variation type.

If we have set the app to use pthreads through the JNI, the `startJNIdelay()` or `startJNIfib()` methods are called, to start the altered sleep or Fibonacci microbenchmark respectively through the JNI. Both of these methods take two parameters, the number of threads to execute the microbenchmark, and the number of executions of the microbenchmark per thread. For the variable total work type, the number of executions per thread is always 1, and the method is called 100 times separately. For the fixed total work type, each thread is given an amount of executions that contribute to the total work to be shared between threads, and the method is called once.

When using Java threads that execute purely through the Dalvik Virtual Machine, new threads are created the required number of times, and allocated their set work. For the variable total work type, an outer-loop iterates 100 times, and an inner-loop creates the required number of threads. For the fixed total work type, a loop creates all the required threads, and each thread is allocated work to complete the microbenchmark a set number of times.

For both threading environments, a sleep occurs between each of the 100 executions of the microbenchmark when running the variable total work type, ensuring that the previous iteration of the 100 has completed before the next begins. Not doing so would result in cumulatively longer completion times with a large number of threads, and higher average completion times for a single microbenchmark execution. Therefore, the variable total work type results in 100× the set number of threads being created during the course of the benchmark, and it is assumed that threads from any previous iteration will have finished their own work and been destroyed by the time the next iteration takes place, allowing the new thread to be properly scheduled to a physical core. Figure 3.3 illustrates how sleeping in between iterations for the variable total work benchmark affects power consumption.

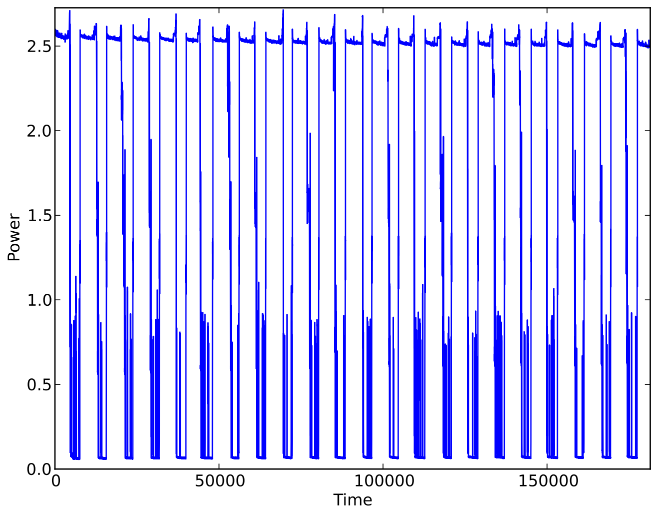


Figure 3.3: An example power trace of a variable total work benchmark, illustrating peaks and troughs due to sleep in between each iteration.

The implementation of the two algorithms, the altered sleep and Fibonacci, are as close as possible between the two threading environments, Java and C, and are programmatically identical. The iterations in the loop used for the altered sleep was set to be 950. This value resulted in an average completion time of 1.01 seconds when run through JNI in the quad-core Nexus 4 device. For the Fibonacci algorithm, `fib(35)` was the highest possible value to use, due to the dual-core Galaxy Nexus not being able to handle a larger recursive call stack, thus in order to keep the experiment balanced, `fib(35)` was used for all tests.

Without a USB cable attached and no visual confirmation available due to the devices screens being turned off, the app plays a 400ms sound using the device's loudspeaker to signal the end of the benchmark test, allowing the power measurement equipment to be stopped in a timely order. This does not need to be exact as power usage traces will be filtered, through manually removing the start and end of the tests using plotting software, to contain only the time period whereby the benchmark ran. Figure 3.4 shows an example power trace prior to the device idle time, after the benchmark ran and sound played, being removed.

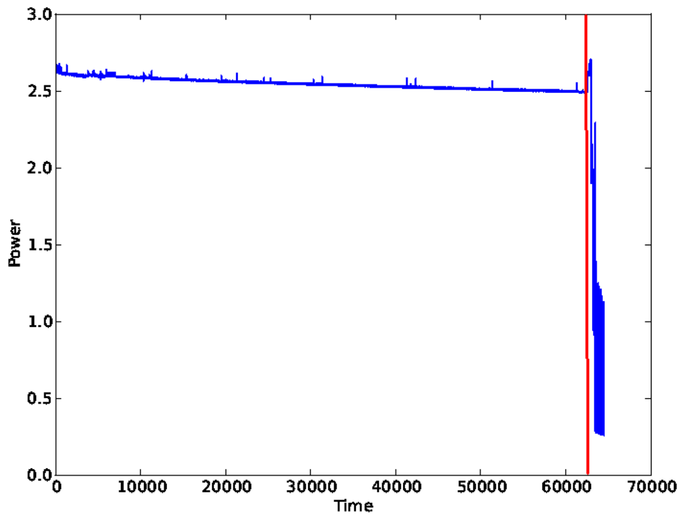


Figure 3.4: An example power trace prior to device idle time being removed from the tail. The high spike before the tail drops is visual representation of increased power usage from when the sound played. The decline in power usage throughout the test is due to some of the testing threads finishing their work earlier than others. The vertical red line represents the end of the test.

For recording performance, completion times are appended to a text file on the devices internal flash memory. The process of writing to a file did not have any adverse effects on completion times, as this method was compared to printing the completion times to a connected computer through a USB cable. Additionally, all tests wrote completion times to a file, ensuring fairness between benchmarking configurations.

3.5.3 Controlled experiment

To ensure that a controlled experiment is performed, thus I can evaluate performance and energy consumption as the threading configuration changes in the benchmarks, the following factors are controlled that could influence performance and power used by the devices.

Power measurement setup increasing resistance

The length of the cables between the device battery, the resistor, and the device, are kept as short as possible, in order to keep internal resistance to a level comparable to having the battery connected directly into the device. As found by previous work [21] with this setup, the extra resistance added is less than 1% in the recorded values. This is due to the choice of a low 0.05Ω resistor, that is subsequently amplified to supply readings, adding only 0.6Ω resistance to the +ve pole, and 1.0Ω resistance to the -ve pole.

System caching of app

By redeploying the app for every separate test, I ensure that the instance of the app deployed for the previous test has been completely destroyed. When an app is changed and redeployed through the Android SDK, all of the previous currently allocated resources to the app are destroyed, if the app was already running. This is because the instance of the Dalvik VM executing the app is finished when a new version of the app is deployed to the device. The app's optimised Dalvik executable files, ODEX files, are also replaced, so every test runs under the same conditions.

Recording completion times

Writing the completion times to files for the purpose of performance analysis added less than a 1% overhead when compared to running the benchmarks without writing of files. This overhead is also applied to all tests, so that the experiment is fair. The text files that were written to for results were created prior to running the tests, and results were appended to these files, avoiding the additional work required to create a file and keeping the overhead to a minimum.

CPU Frequency scaling

By default in both devices, CPUfreq, the Linux CPU scaling subsystem, adjusts the CPU clock frequency according to current workload. The clock frequency is lowered in order to save power when a higher frequency is not required. Thus, at some periods of the tests, the CPU clock frequency could be different if changed by CPUfreq to save power. To remedy this, the CPUfreq scaling governor was set to *performance*, and the clock speed fixed at the highest frequency. Appendix A describes the devices configurations in further detail.

Physical environment

Initial tests with the quad-core Nexus 4 resulted in the device shutting down as a *safety cut-off*, due to high battery temparature sensor readings. These cut-offs occured more frequently when compared to an unmodified device, as CPUfreq was not allowed to lower the CPU clock frequency to anything below the maximum, as outlined above. Under normal conditions, when it becomes too hot, the device lowers the clock frequency and allows the CPU to draw less power and reduce heat produced. To counter these heat issues, the room temparature was decreased to 15 °C, and both devices ran the tests without a back cover, so that any generated heat could leave the device more efficiently. For the quad-core Nexus 4, this technique was particularly effective as the back cover insulated heat — no heat related shutdowns subsequently occured.

Battery

A healthy and fully-charged battery was used for all tests.

Device peripherals

There were no SIM cards in either device, as it has been shown [6] that their presence can reduce battery life. The screens and notification LEDs were off when recording results. Additionally, the devices were both in Aeroplane Mode, to prevent unneccessary use of radio components, that would affect power readings.

3.5.4 Error analysis

The results may still contain random and systematic error, even though the experiment is designed to mitigate any sources of variances in the results. Sources of error include:

External power fluctuations

Previous work with this power measurement setup used two 3.7V batteries connected, in series, with the resistor to measure current [21]. By introducing two power supplies to replace the batteries, there is the possibility of power fluctuations caused externally to the setup. However, with this setup, performance effects caused by either of the two batteries running flat are mitigated. Any fluctuations would also be easily recognisable in the results, when compared to an experiment whereby the device itself is also powered through an external power source instead of its own battery.

Accuracy of DAC when measuring power

A systematic error may be added by the digital-analog-convertor as it reads the current across the resistor and sends the information to the computer. Inconsistencies in the meter's internal clock would cause error in the results, as energy used is calculated based on the assumption that power use is constant through the sampling interval. Additionally, the results are downsampled, and a median[6] average power is to be calculated per second. If there are timing issues with the DAC, the accuracy of this calculation is reduced.

Increased resistance

Further systematic error of less than 1% on the recorded values is added with this setup, due to the addition of cables and a resistor between the device and its battery.

Device background services

As further specified in Appendix A, all running applications that are not part of the Android operating system are Force Stopped. Background services that are not needed to continue testing are also ended. However, even with these precautions, the operating system performs various system tasks, such as garbage collection, at random times during testing. To ensure that any emerging distributions are still apparent, tests are repeated multiple times and averages are calculated.

[6]Mean averages were also calculated, and differences between the mean and median average were minimal.

Shared counter variable

For the fixed total work benchmarks, a shared counter variable is used in order to check for when all of the working threads have finished their contribution to the total work, and the overall test is complete. The shared variable is incremented by each thread every time it runs an iteration of the microbenchmark. By using this approach, issues related to cache inconsistency may occur — threads could continue with a further iteration, as when they make their check, total work has not been reached, but another thread has incremented the variable in the meantime that has resulted in total work being completed. This will lead to the test `finish()` method being called more than once and thus more than one completion time would be written to the performance file. In this instance, the first record in the performance file is used in the analysis. An atomic compare and swap at every iteration in each thread would resolve this issue, but at an additional performance overhead that would not be beneficial to the results.

3.5.5 Summary of setup

The benchmarking setup can be summarised with the following characteristics:

Test outline

All tests are run with an increasing number of threads, from 1 thread up to 8 threads. The microbenchmark is run a set number of times, depending on whether the test is for variable total work, or fixed total work. The two benchmarking algorithms are used, with both Java threads through the Dalvik Virtual Machine, and native pthreads. All tests will be completed on both the dual-core Galaxy Nexus, and the quad-core Nexus 4. This results in a total of 128 separate tests, whereby the four different microbenchmarks, Java sleep, pthread sleep, Java Fibonacci and pthread Fibonacci, are executed. These tests are all executed at the maximum CPU frequency of the device.

Additional CPU frequency modification tests

In addition to the 128 separate tests executed at full CPU frequency, I also run benchmarks where the CPU frequency is modified. By running two threads at half the CPU frequency and one thread at double CPU frequency, I can find out how expensive in terms of power usage and performance it is to utilise the two physical cores in the devices when compared to one that runs at a higher frequency. The threads in these tests work towards the fixed total work benchmark. For example, comparisions are drawn in computing `fib(35)` 400 times, between one core running at 700MHz, and two cores running at 350MHz, where each computes the microbenchmark 200 times.

Overall, there are 152 unique test configurations, and performance and power consumption is recorded for every test.

No control software

No additional software is needed to run on the devices to record power consumption. Performance is recorded through the app to a file. Therefore, device resources are not wasted through recording results. The computer attached to the DAC meter runs the software required to record power.

No user interaction

The tests are fully automated once started, removing a potential variance source. The benchmark configuration parameters are defined in the source code prior to running the tests in batch. The built-in Java sleep, that consumes little power when compared to the altered sleep microbenchmark, is used between in order to distinguish when repeated tests started in the power results.

Unmodified device hardware

The addition of the battery dock used in order to intercept the circuit for power measurement is the only device modification. The device's internal hardware is not changed to increase experimental validity.

3.6 Microbenchmark multicore performance analysis

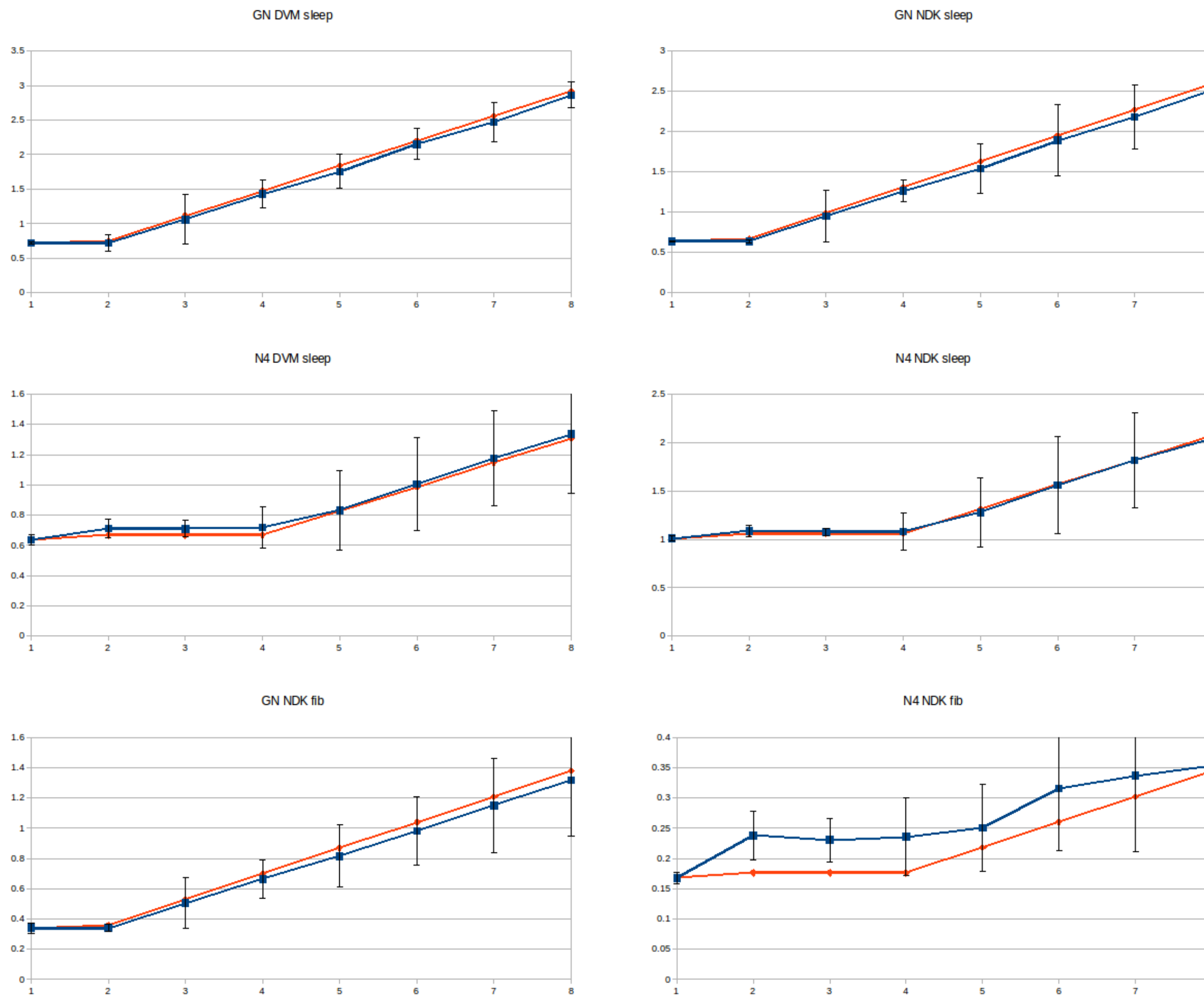
I now present the experimental results of the study of multicore performance through the use of two microbenchmarks, executed through alternate threading environments, and with different total work batch variations. Additionally, I show that developers should use the amount of physical CPU cores as the number of threads when offloading work for concurrent processing to additional threads, for maximum performance. Furthermore, the differences in multiple threads against a single thread completing the same work are dependent on the device and its CPU type.

3.6.1 Mapping multicore performance to prediction model

In Section 3.4.1, I claim that completion times remain constant under the variable total work load/fixed work per thread tests, as the number of threads is increased up to and including the amount of physical cores in the device. Thereafter, completion times increase as the additional threads work is spread over the physical cores. I also claim that, under the fixed total work load/variable work per thread tests, the completion times decrease as number of threads increases, until the amount of physical cores is reached, thereafter performance remains constant. I now use the results of the microbenchmark tests to validate these claims.

Variable total work/fixed work per thread

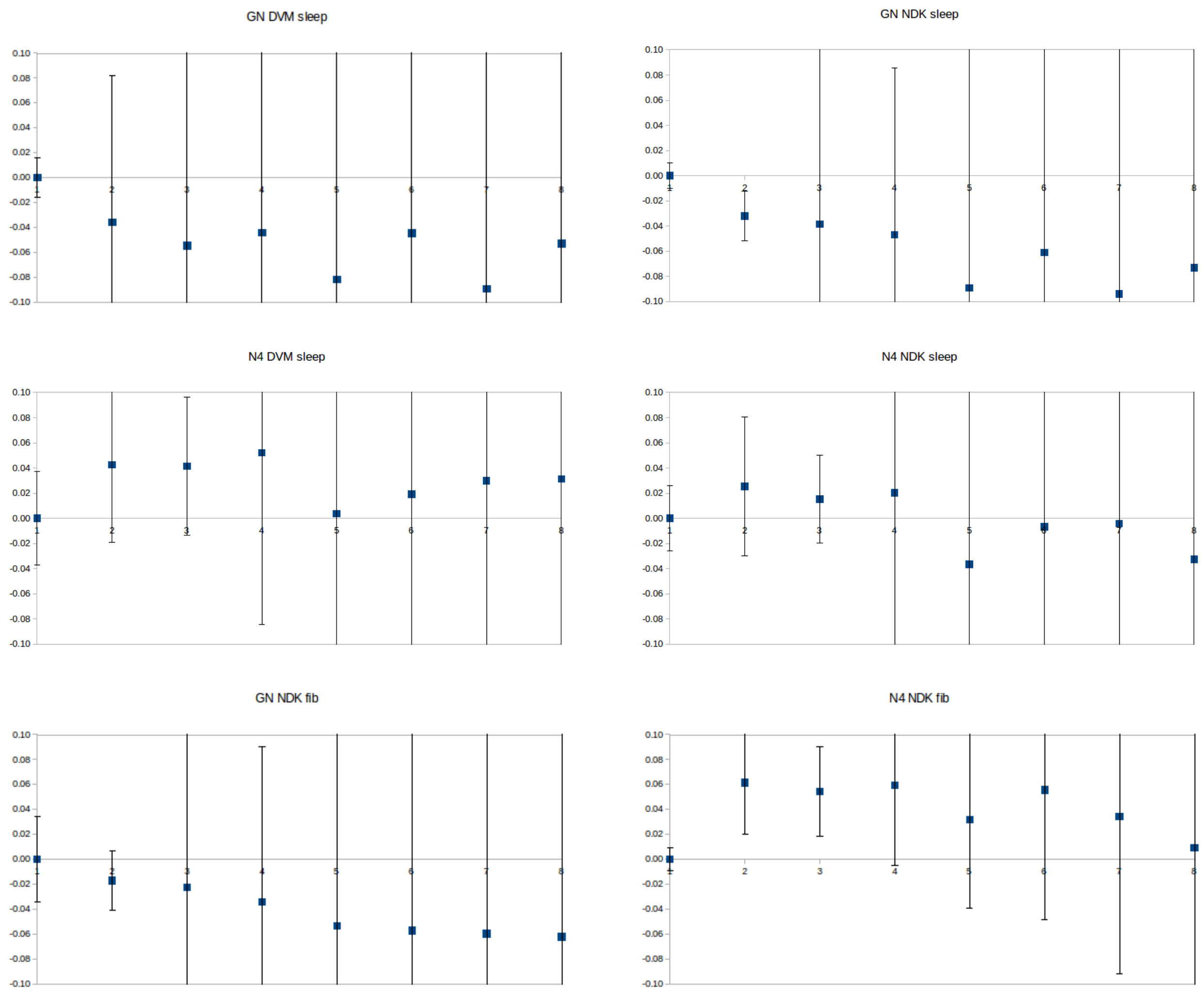
Figure 3.5 illustrates the median average completion times of a single execution of a microbenchmark, as threads concurrently executing the microbenchmark are increased. This test was performed with both the dual-core Galaxy Nexus, and the quad-core Nexus 4, through the use of both Dalvik Virtual Machine-based Java threads, and NDK/JNI-based native pthreads, also with both benchmarking algorithms. The figure shows that the performance prediction model is largely accurate — the actual results lines are distinctly close to the prediction lines. The main exception to this is the quad-core Nexus 4 Fibonacci test through native NDK pthreads, though this is only due to the significantly smaller scale of the completion time so differences are more magnified when compared to other tests. Magnification occurs as the Nexus 4 took a median average of only 0.354 seconds to complete the



Key: ● Predicted ■ Actual

x-axis: Number of threads
y-axis: Completion Time (seconds)

Figure 3.5: The average completion times as the number of executing threads are increased, for the variable total work performance tests. Predicted values found using the model in Section 3.4.1 are shown against the actual measured values.



x-axis: Number of threads
y-axis: Residual value (seconds)

Figure 3.6: The corresponding graphs of residuals of Figure 3.5. The small scale of the residual graphs makes it easier to analyse differences between the predicted values and the measured values, but it amplifies error substantially.

microbenchmark with 8 threads running simultaneously, compared to the 2.862 seconds required for the dual-core Galaxy Nexus to compute the sleep algorithm through 8 Dalvik Java threads.

The model includes an overhead, S, due to Armdahl's Law, and we can see from the results that the applied overhead was more accurate for the dual-core Galaxy Nexus. This could mean that there is a higher amount of serial code before the parallel portion is reached on the Nexus 4, such as more work required to translate Java bytecode into CPU instructions, so the value of 5% serial code in the model may not be as accurate for the Nexus 4 if it takes longer to initialise the microbenchmark, and does so using a serial method.

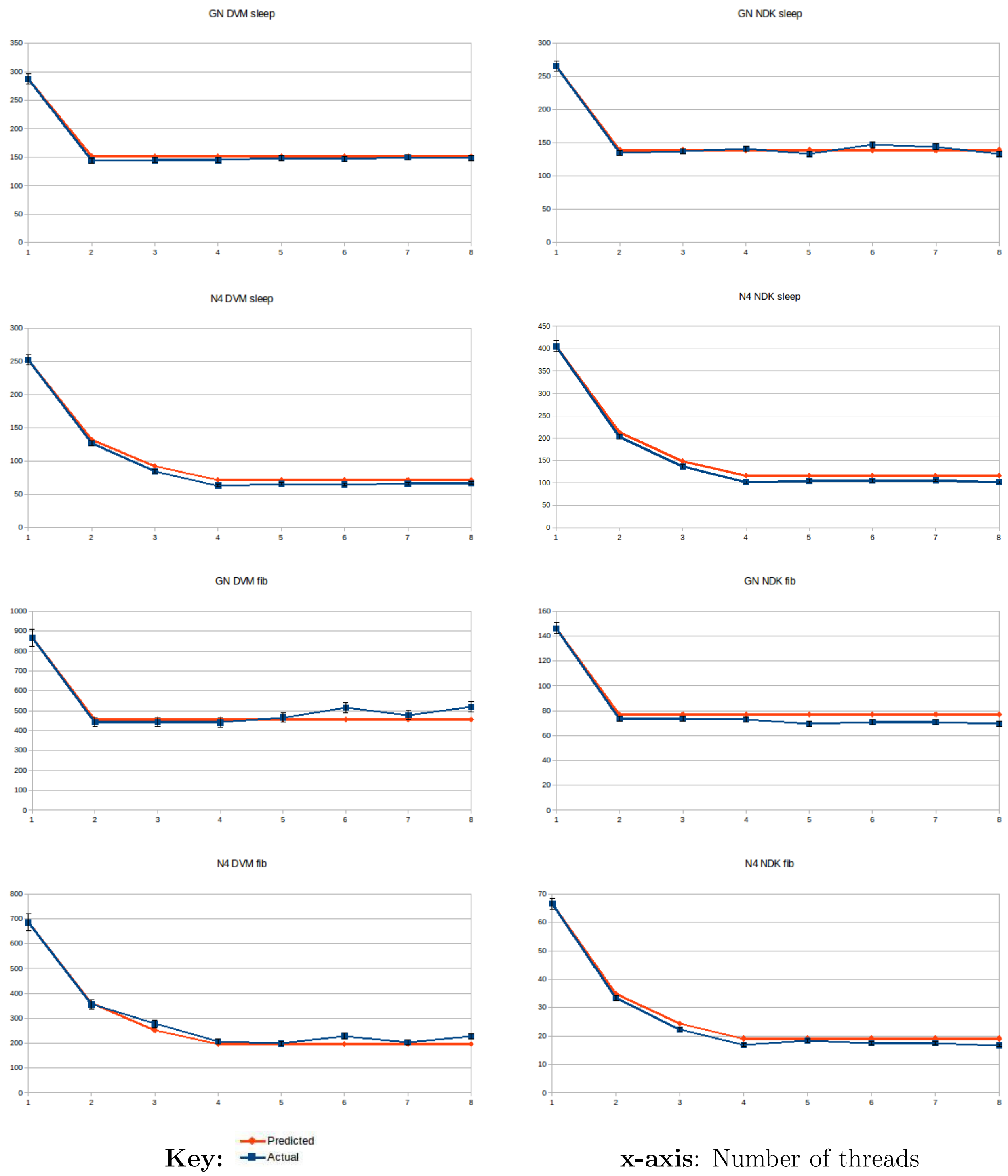
Additionally, the actual results from the dual-core Galaxy Nexus always outperfom the results based on the predicted model. On the other hand, the quad-core Nexus 4 consistently did not perform as well as it should have, based on the prediction model. This suggests that there is more of a performance overhead when multiple cores are introduced with the Nexus 4, because the prediction values at a thread number higher than 1 are based on the single thread performance.

The graphs of residuals shown in Figure 3.6 confirm the accuracy of my variable workload prediction model, as the small scale of the graphs demonstrates little difference between the predicted values and the measured values. The error bars on the graphs also cross the y-axis at 0 in 45 out of 48 tests, so we can conclude that the prediction model is accurate within error.

Fixed total work/variable work per thread

Figure 3.7 shows the total completion time for computing a microbenchmark 400 times, with work shared concurrently between an increasing number of threads. The figure shows the fixed total work performance prediction model to be accurate - the actual result lines are again very close to the predicted values lines.

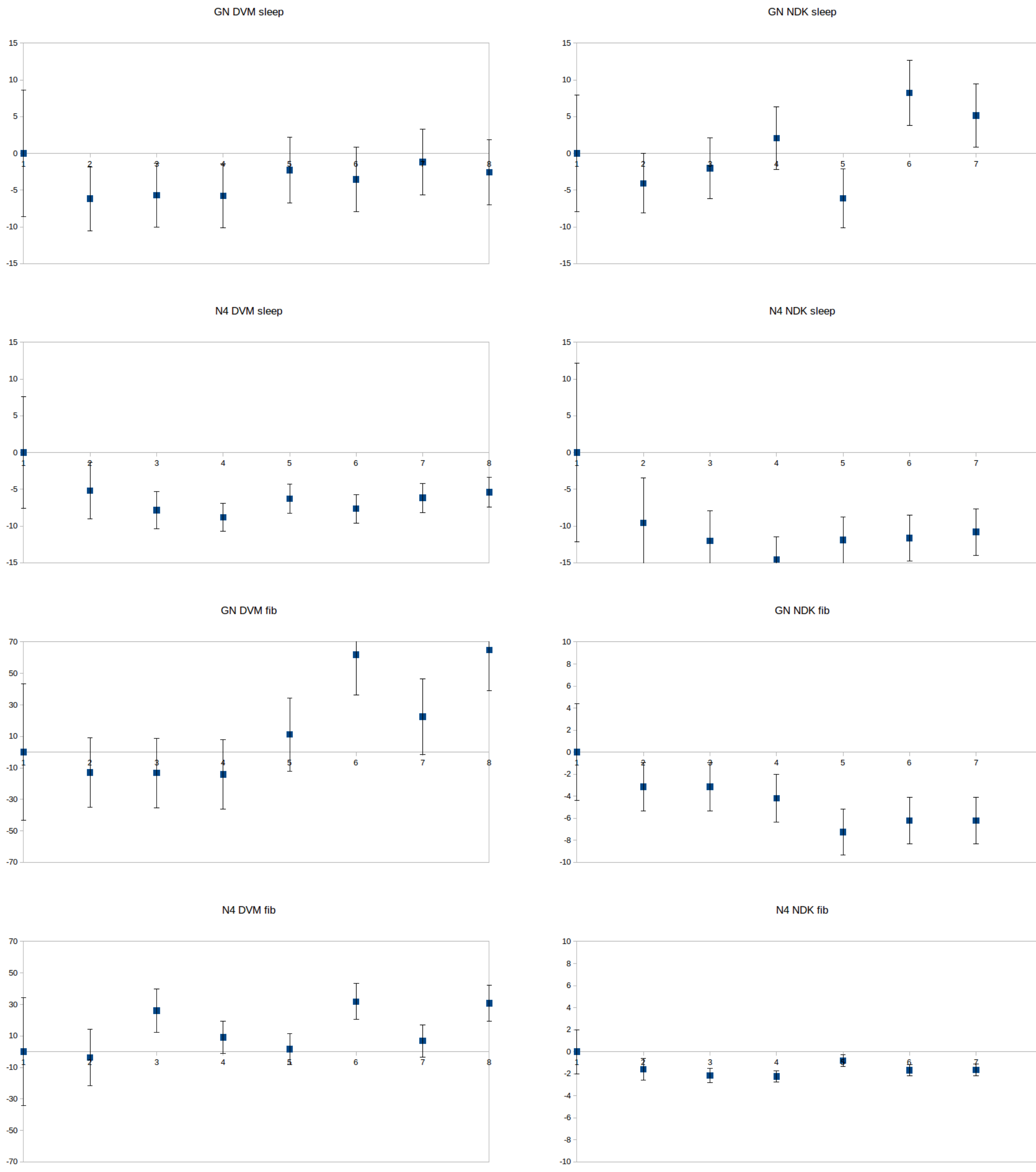
Performance was slightly better than predicted with the quad-core Nexus 4 running the sleep microbenchmark, showing that the introduction of multiple threads benefited performance more than expected, as the model is based on the performance of a single thread. Additionally, as I will show later, the Qualcomm Krait CPU chip in the Nexus 4 consumes a significantly greater amount of energy when more than one core is in use. This could result in marginally better performance for the single core that consumed a lower amount of energy in the single thread tests, as more power is consumed by



Key:

x-axis: Number of threads
y-axis: Completion Time (seconds)

Figure 3.7: The total completion times as the number of executing threads are increased and workload is shared, the fixed total workload. Predicted values found using the model in Section 3.4.1 are shown against the actual measured values.



x-axis: Number of threads
y-axis: Residual value (seconds)

Figure 3.8: The corresponding graphs of residuals of Figure 3.7.

the SoC overall, and power consumed is not proportional to the number of cores in use.

Both devices exhibited worse performance than predicted when running the Fibonacci test through DVM threads. This is largely due to the recursive implementation of the Fibonacci microbenchmark — the DVM has a stack size of 8KB per thread, which quickly becomes full and affects performance. Noticeably worse performance than predicted occurs when we reach 6 concurrent threads with both devices.

In three of the four NDK tests, the actual measured values were lower than the predicted values. This suggests that there was less than 5% serial code, as estimated in Section 3.4.1, and thus the overhead applied through Armdahl's Law was slightly greater than required when native pthreads through the NDK were used.

As shown by the graphs of residuals in Figure 3.8, the fixed workload performance model is not as accurate as the variable workload performance model, when the graphs are magnified to a smaller scale. The error bars cross the y-axis more with the Galaxy Nexus, suggesting that the Nexus 4 did not perform as predicted. Overall, in proportion to the total values, Figure 3.7 confirms the accuracy of the model.

3.6.2 Effects as amount of physical cores varies

In all performance tests, we have seen the expected results as shown by the prediction models with regards to using devices with a different number of physical cores.

In the variable total workload tests, where the average completion time for a microbenchmark was calculated as thread amount increased, the results for the dual-core Galaxy Nexus showed a substantial increase in completion time when the number of threads was greater than 2. Conversely, the results for the quad-core Nexus 4 showed a large increase in completion time when the number of threads was larger than 4. As expected by my prediction model, completion time was relatively constant with both devices where the number of threads did not exceed the amount of physical cores.

As shown in Figure 3.9, the fixed total workload tests demonstrated a decrease in completion time as the number of threads was increased, as expected by my prediction model. This decrease occured until the number of threads exceeded the number of physical cores, where the measured completion times

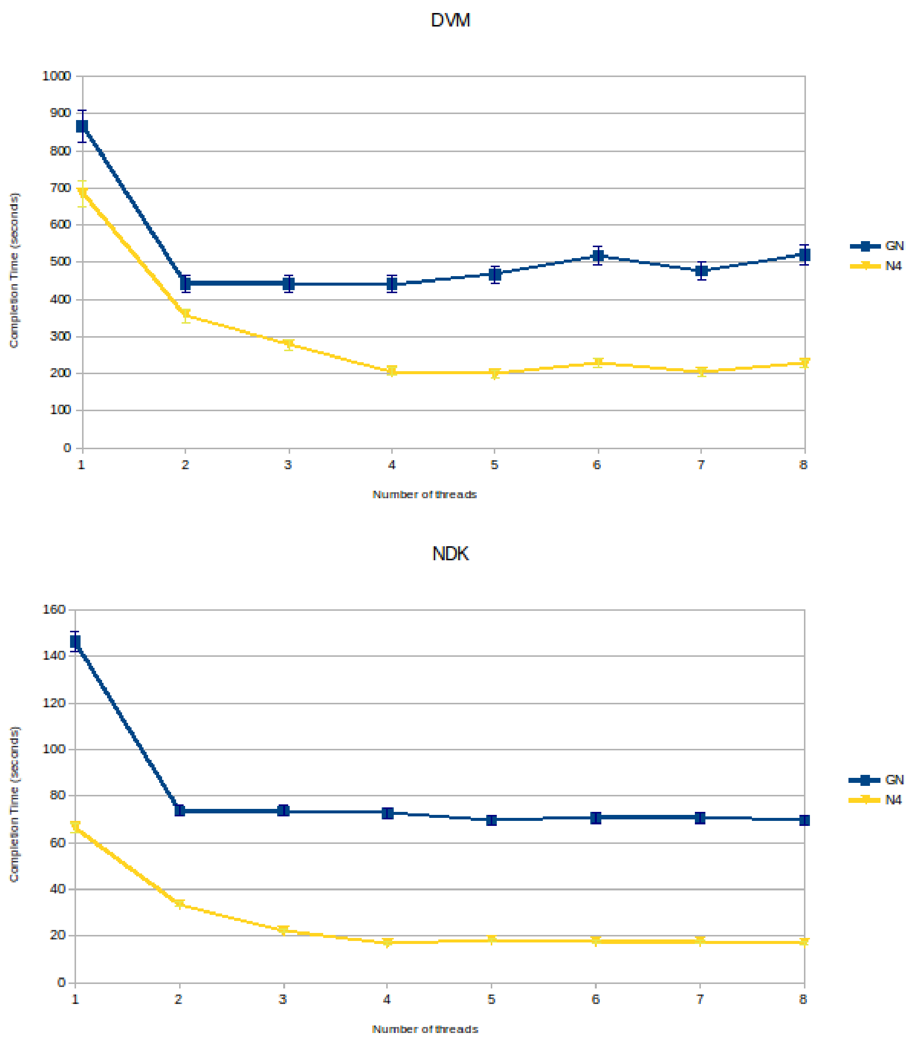


Figure 3.9: The total completion time for the fixed workload, where threads share the total work of executing the Fibonacci microbenchmark 400 times. The two threading environments are separated due to differences by an order of magnitude between their measured results.

remained stable as thread amount increased. Figure 3.9 shows that the dual-core Galaxy Nexus had a substantial drop in completion time from 1 to 2 threads, with completion times remaining constant thereafter. The figure also shows that the quad-core Nexus 4 had drops in completion time as number of threads increased up to 4 threads, with completion time remaining constant when the microbenchmark was divided between 4 threads of more. The Nexus 4 results also show that there was a greater performance gain from increasing the number of threads from 1 to 2, than from 2 to 3 and any subsequent increases.

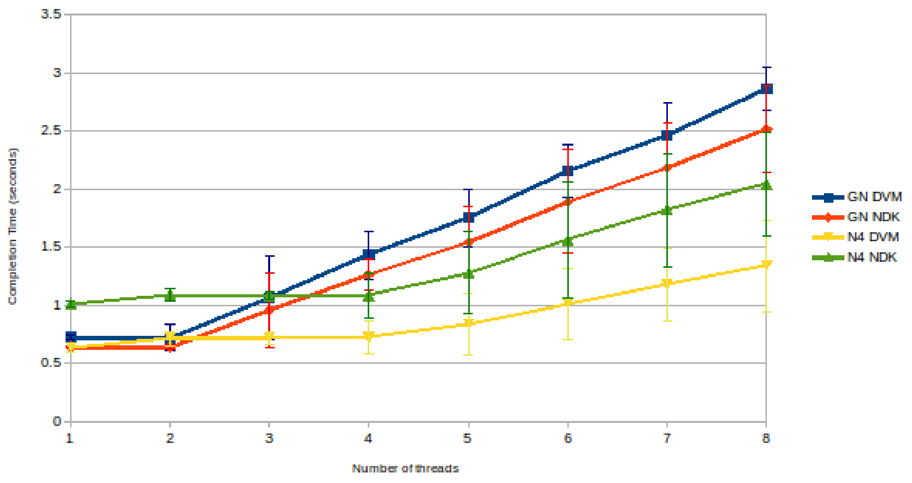
3.6.3 Comparison of threading runtimes

On the whole, using native pthreads had better performance when compared to Java threads through the Dalvik Virtual Machine. However, as illustrated in Figure 3.10, the exception to this was the quad-core Nexus 4 where it actually performed worse when using native pthreads to perform the sleep microbenchmark. On the other hand, the NDK code did outperform Dalvik Virtual Machine code for the Fibonacci benchmark with the Nexus 4. This inconsistency with the Nexus 4 NDK performance when running the sleep microbenchmark may well be due to the Snapdragon S4 Pro SoC in the Nexus 4 having poor performance with simple logical integer-only operations[7], which are the type of operation used for the sleep microbenchmark.

The Fibonacci microbenchmark performed an order of magnitude faster through using native pthreads with the NDK, rather than using Java threads through the DVM. This is due to the recursive implementation of the algorithm, requiring a large stack size. A default DVM Java thread has stack size of 8KB, whereas NDK native pthreads have a 1MB stack size each.

This section has shown that, on the whole, for best performance it is beneficial to use the same number of threads as physical cores when offloading processing work to multiple threads. In terms of threading runtimes, the poor performance of the Nexus 4 when using native pthreads through the NDK shows that the choice of threading runtime is device dependent. However, the Fibonacci microbenchmark had much better performance with both devices using native pthreads, so if a developer must choose a runtime to use, then using the NDK to create native pthreads is the *safest* option.

[7]http://www.openssl.org/ appro/Snapdragon-S4.html



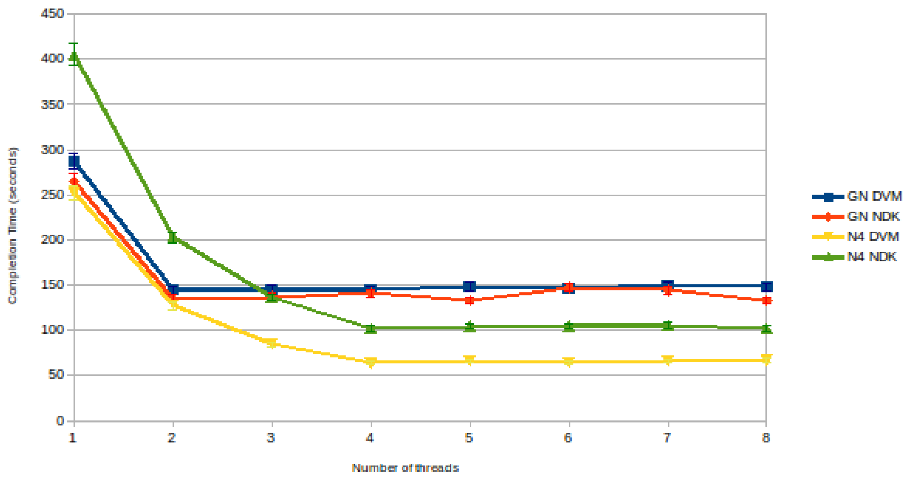


Figure 3.10: Top graph: The average completion times of the sleep microbenchmark, calculated using the measured values of the variable total workload tests. Bottom graph: The total completion time of 400 runs of the sleep microbenchmark, shared between threads through the fixed workload tests. The results from both devices are plotted with each device running tests through the two threading environments.

3.7 Microbenchmark multicore energy consumption analysis

I now present the experimental results of the study of multicore energy consumption through the use of two microbenchmarks, executed through alternate threading environments, and with different total work batch variations. Additionally, I show that developers should use one additional thread above the amount of physical cores in a device when offloading work for concurrent processing to additional threads, in order to save energy.

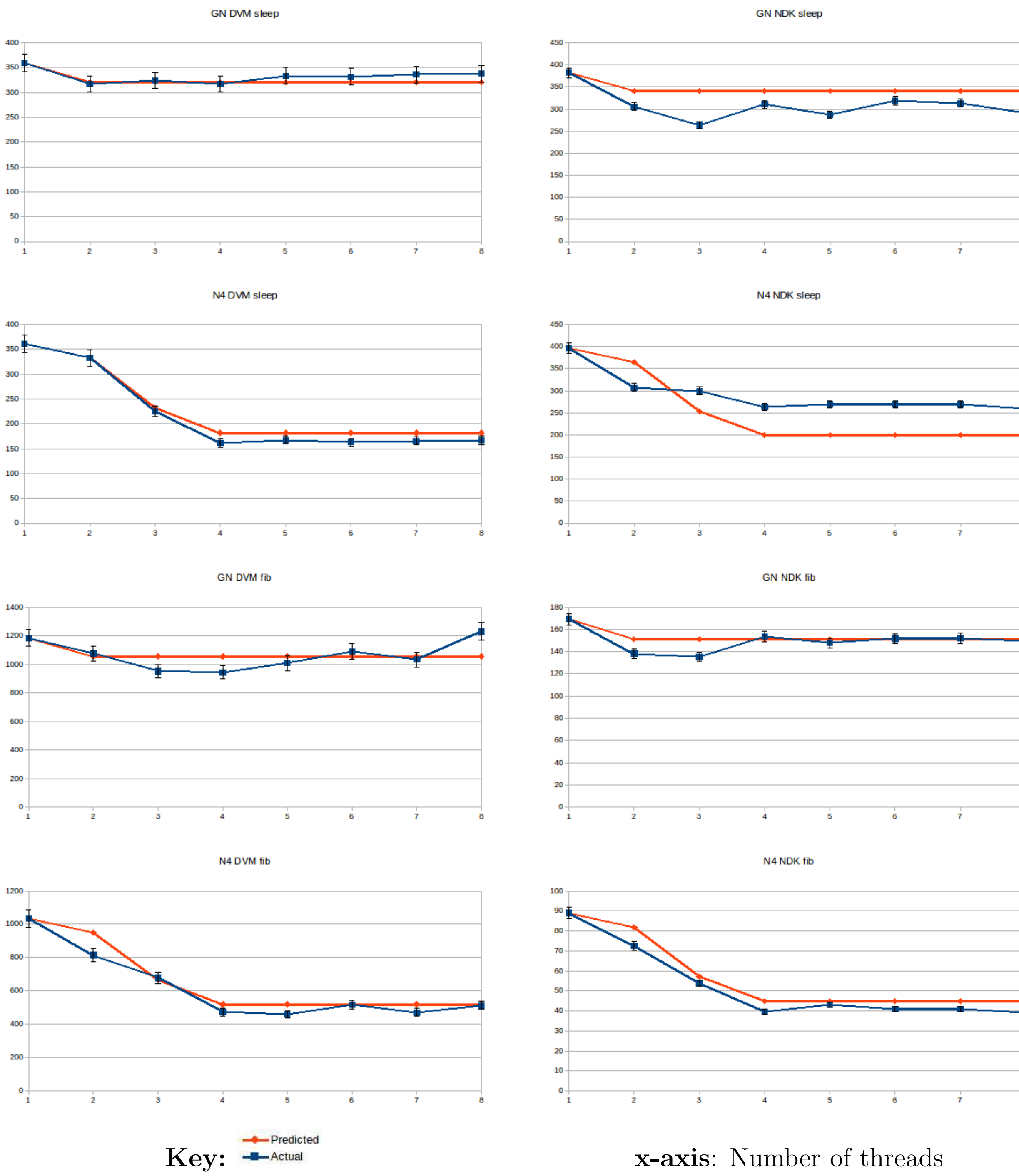
To calculate energy consumption in the following sections, the average amount of watts measured over the course of the respective test was multiplied by the corresponding completion time from the performance analysis results.

3.7.1 Mapping multicore energy consumption to prediction model

In Section 3.4.2, I claim that energy consumption will decrease, as the number of threads executing sharing the total work is increased, up to and including the amount of physical cores in the device. Thereafter, energy consumption remains constant. In Section 3.4.2, I claim the existence of an energy cost applicable once more than one core is in use, the *multicore multiplier*. I found this value to be 1.7 for the dual-core Galaxy Nexus and 1.75 for the quad-core Nexus 4. In itself, this shows that the CPU in the Galaxy Nexus has less of an energy overhead if it makes use of its second core, when compared to the CPU in the Nexus 4.

Figure 3.11 illustrates the energy used in computing a microbenchmark 400 times, with the work shared between an increasing number of concurrently executing threads. It also shows the accuracy of my energy consumption prediction model. The model appears to be accurate from Figure 3.11, but the graph of residuals in Figure 3.12 shows that in most cases, the predicted values were not within the error of the measured values. Additionally, in six of the eight tests, the actual results use less energy than the predicted results. The exceptions to this are the Galaxy Nexus DVM sleep microbenchmark with 5 or more threads, and the Nexus 4 NDK sleep test. The Nexus 4 issue is most likely due to the same NDK performance issues described in Section 3.6.3.

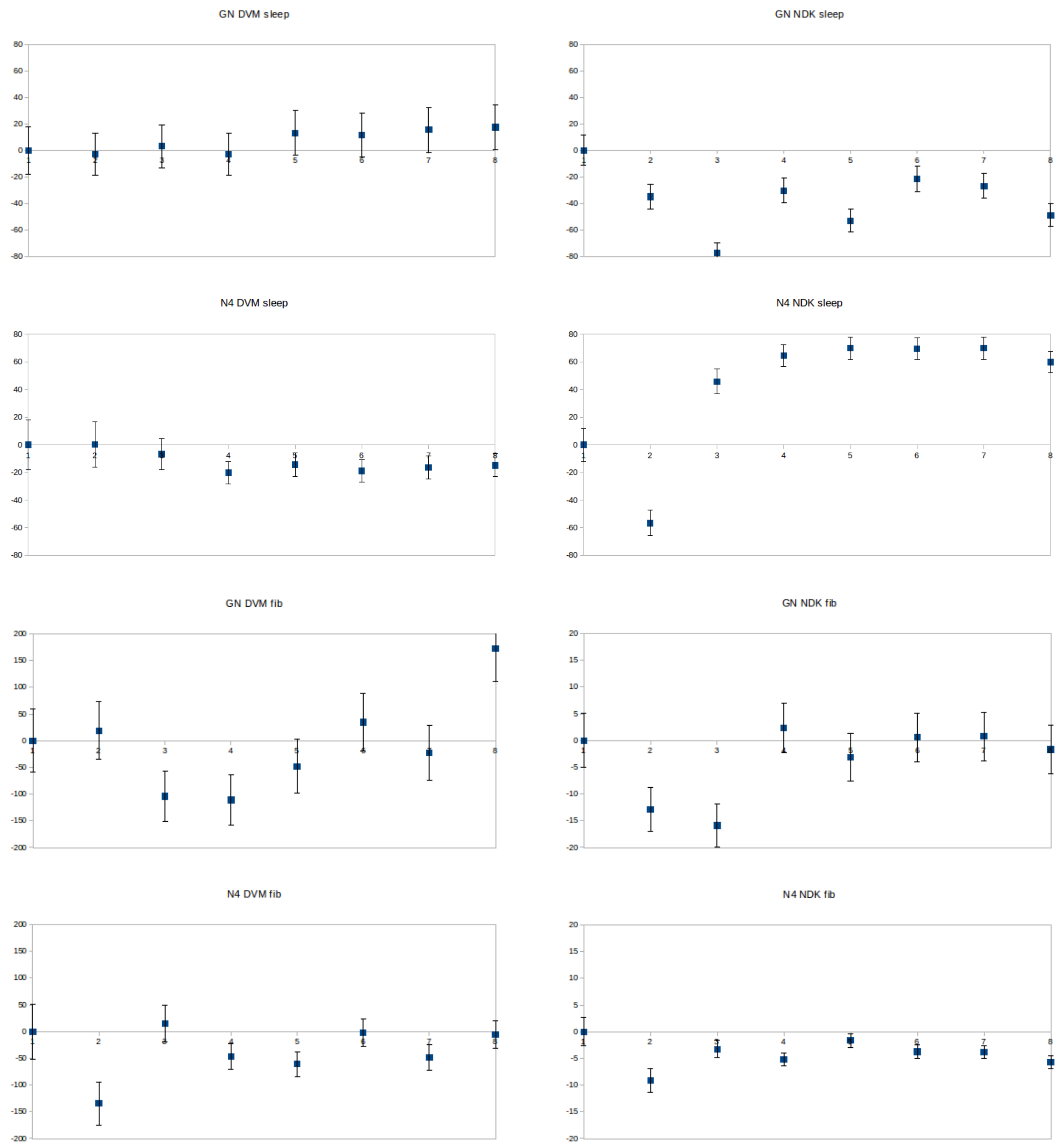
Additionally, the actual results show that the initial overhead of using more



Key:

x-axis: Number of threads
y-axis: Energy used (joules)

Figure 3.11: The energy used as the number of executing threads is increased and workload is shared, the fixed total workload. Predicted values using the model in Section 3.4.2 are shown against the actual measured values.



x-axis: Number of threads
y-axis: Residual value (joules)

Figure 3.12: The corresponding graphs of residuals of Figure 3.11. The small scale of the graphs magnifies the differences between the predicted values and the measured values.

than a single thread was not as large as predicted, with less energy being used when using 2 threads in six of the eight tests, including all of the NDK-based pthread tests. This shows that the multicore multipler used in the prediction model for each device was slightly too high for the initial switch to multicore use at 2 threads, but increasingly accurate when more threads are used. The prediction model could be modified in future work to account for this, by allowing the multicore multiplier to be tweaked depending of number of executing threads. The multicore multiplier may also be modified to account for processors of a CPU architecture other than ARM, as described in Section 5.2.

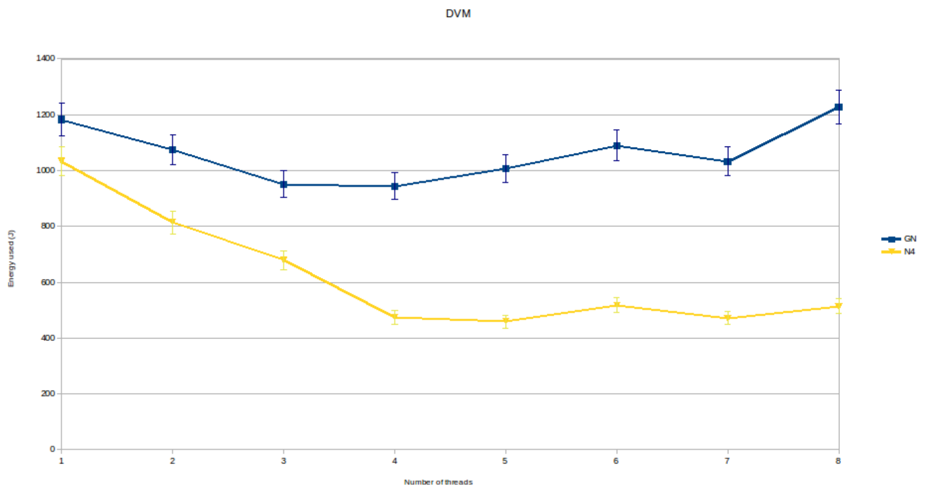
In the context of the study, these findings suggest that, due to the actual measured values using less energy on the whole than predicted, multicore processors in smartphones are used effectively with regards to energy consumption. The power used increases when a second thread is used, but the overall energy cost is reduced due to shorter completion times. The energy cost is reduced as the number of threads used increases, up to the amount of physical cores. Thus, in the next section, I will analyse the effects of variance in the amount of physical cores.

3.7.2 Effects as amount of physical cores varies

Figure 3.13 demonstrates the energy consumption results for the fixed workload Fibonacci tests. With the quad-core Nexus 4, the least energy was consumed when the number of executing threads was 4, as completion time was substantially reduced, but power consumption stayed relatively constant after 2 threads were introduced. The figure also shows that the Nexus 4 had more stable energy consumption as threads were increased when running through NDK pthreads, demonstrated by the straighter gradient line when compared to the more erratic Dalvik Virtual Machine energy consumption, particularly at 3 running threads.

However, the results from the fixed total workload for the dual-core Galaxy Nexus show a different conclusion — using 3 threads in the dual-core device was more energy efficient. This is due to completion times being similar from 2 to 3 threads, but power usage was lower at 3 threads in the Galaxy Nexus. The result is unexpected, but consistent so it is a valid conclusion.

The energy consumption for the variable workload, as shown in the top graph of Figure 3.14, illustrates that more energy is consumed for a single execution of the sleep microbenchmark as more threads concurrently execute the



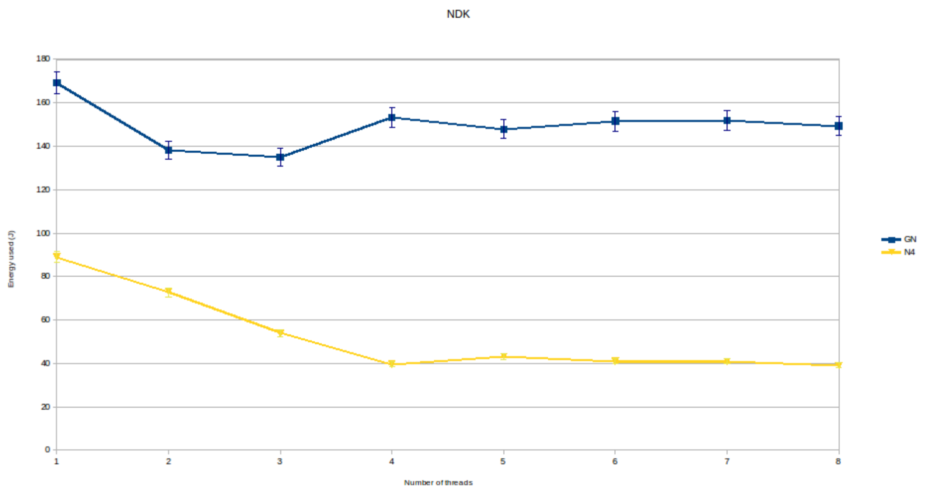


Figure 3.13: The energy used for the fixed workload, where threads share the total work of executing the Fibonacci microbenchmark 400 times. The two threading environments are separated due to differences by an order of magnitude between their measured results.

Number of threads	Average power (W)	Energy used (J)
1	1.51	1033.69
2	2.29	815.04
3	2.45	678.61
4	2.29	473.64
5	2.31	459.09
6	2.26	517.89
7	2.31	471.34
8	2.26	514.61

Table 3.3: Energy used and average watt measurement throughout the Fibonacci microbenchmark test using Dalvik Java threads on the Nexus 4 for the fixed total workload.

microbenchmark. The rate of which the energy consumption increases is dependent on the device and the threading environment.

Table 3.3 shows watt averages before conversion to joules for energy used. It demonstrates that even with a higher average power reading throughout the tests, once the corresponding completion times are applied, energy used actually goes down as computation completes in a faster time, mitigating the extra power required to run the additional cores. It also shows that the least energy was consumed with 5 threads for this test with the quad-core Nexus 4, despite there being a slightly higher average power reading. This result with 5 threads did not occur with all of the Nexus 4 tests, but the difference between 4 and 5 threads with the Nexus 4 was always minimal in terms of energy consumption.

From my results, it is apparent that for the fixed total workload, where threads share work, it is beneficial for developers to offload processing work to a number of threads based on the amount of physical cores + 1, for the purposes of the lower energy consumption. For the quad-core Nexus 4, 5 threads consumed a similar amount of energy compared to 4 threads, and for the dual-core Galaxy Nexus, 3 threads consumed less energy than 2 threads in the many cases, sometimes significantly so.

Using the amount of physical cores + 1 for the number of threads is not a new approach, as the UNIX `man` page for the `make -j` command for parallel builds recommends the same method[8]. Their rationale is that when one of the jobs is sleeping, for example on on I/O, there will be another thread ready to go. We could be witnessing the same behavior with the Galaxy Nexus:

[8]http://vim.wikia.com/wiki/Auto-detect_number_of_cores_for_parallel_build

I/O is performed by the microbenchmarks in writing completion times to a file, although this is unlikely due to the minimal data being written to the file in comparison to a parallel `make`.

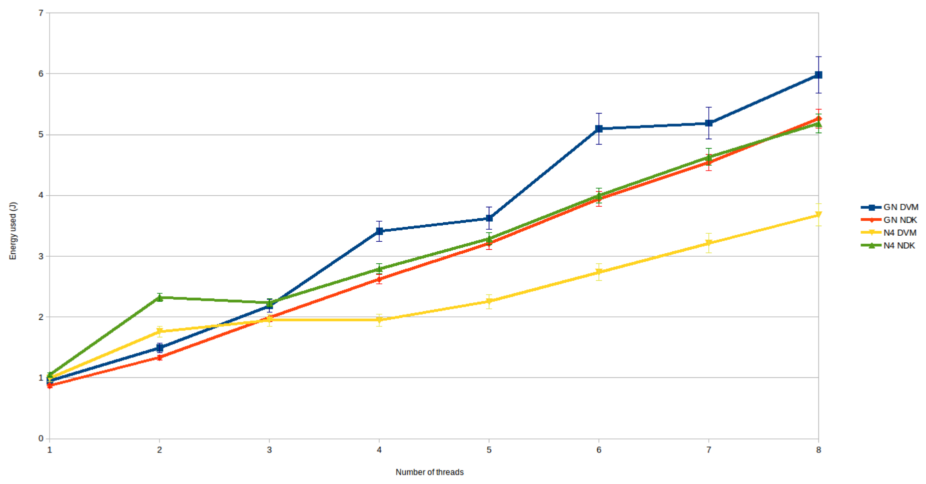
Putting these results into context, these findings suggest that the effectiveness of multicore processors in modern smartphones in terms of energy consumption is dependent on programming design choices. An incorrect selection of the number of threads to use in relation to the amount of physical cores can be detrimental to energy consumption if too many threads are used. I will now analyse whether the programming choice of which threading runtime to use also affects energy consumption.

3.7.3 Comparison of threading runtimes

As shown in Figure 3.14, the sleep microbenchmark consumed less energy when running through NDK pthreads on the dual-core Galaxy Nexus. The poor NDK performance for the sleep microbenchmark with the quad-core Nexus 4 as described in Section 3.6.3 is also evident here for energy consumption, as Dalvik Java threads consume less energy when compared to native pthreads with the Nexus 4.

On the other hand, the Fibonacci microbenchmark is more conclusive in favour of using NDK pthreads when aiming to reduce energy consumption. This is demonstrated by Figure 3.13, where the differences in energy consumption between the threading environments were in an order of magnitude, with both devices.

We have seen that programming choices have contribute to the effectiveness of multicore processors in modern smartphones, with regards to energy used. These results suggest that developers should, use native pthreads for processing work where possible, although this is not conclusive as demonstrated by the poor Nexus 4 energy consumption for the sleep microbenchmark when running through native pthreads. Section 5.2 describes a potential solution to this, by allowing a device to choose the best threading runtime when an app starts. In the previous sections, the effectiveness of multicore processors in modern smartphones has been dependent on programming choices. The next section considers how the governor in the Linux kernel adjusts the CPU clock frequency under load, and how the choices it makes affect the efficiency of multicore processors in modern smartphones.



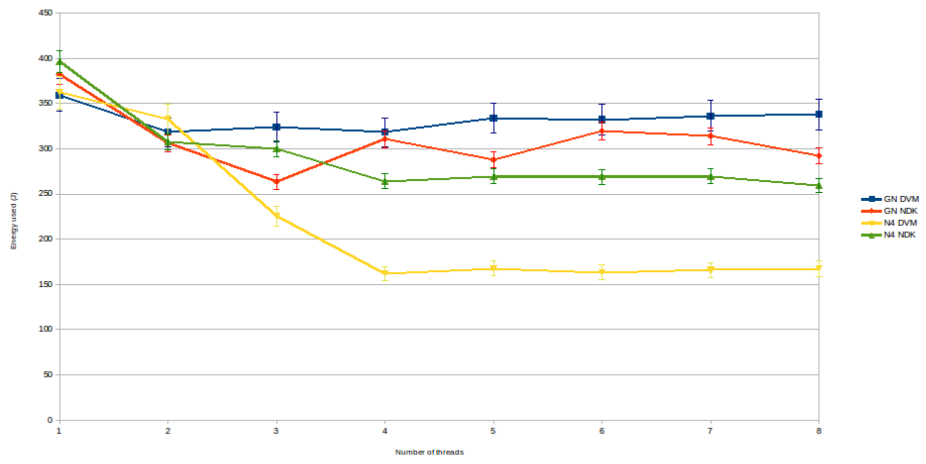


Figure 3.14: Top graph: The energy consumption per execution of the sleep microbenchmark, calculated using the measured values of the variable total workload tests. Bottom graph: The energy consumption of 400 runs of the sleep microbenchmark, shared between threads through the fixed workload tests. The results from both devices are plotted with each device running tests through the two threading environments.

Configuration	Full-speed, fewer threads	Half-speed, more threads
GN DVM: 1x700MHz & 2x350MHz	400.29 (**491.83**)	**220.73** (494.72)
GN NDK: 1x700MHz & 2x350MHz	232.83 (**442.37**)	**155.03** (446.46)
N4 DVM: 2x1458MHz & 4x702MHz	339.63 (132.05)	**247.61** (**97.49**)
N4 DVM: 1x1458MHz & 2x702MHz	354.22 (253.07)	**349.71** (**198.08**)
N4 NDK: 2x1458MHz & 4x702MHz	**316.19** (210.51)	407.79 (**182.16**)
N4 NDK: 1x1458MHz & 2x702MHz	350.42 (403.7)	**317.85** (**311.49**)
GN DVM: 1x700MHz & 2x350MHz	**693.15** (1498.2)	1203.75 (**1482.67**)
GN NDK: 1x700MHz & 2x350MHz	**80.94** (237.57)	126.74 (**236.54**)
N4 DVM: 2x1458MHz & 4x702MHz	820.35 (372.18)	**597.41** (**253.93**)
N4 DVM: 1x1458MHz & 2x702MHz	949.34 (710.05)	**844.95** (**548.18**)
N4 NDK: 2x1458MHz & 4x702MHz	78.19 (**34.66**)	**19.94** (36.37)
N4 NDK: 1x1458MHz & 2x702MHz	84.11 (66.01)	**79.89** (**55.23**)

Table 3.4: Energy used in Joules for the fixed total workload tests with CPU clock frequency modifications. Total completion times in seconds are in brackets (). The least energy used and faster completion times are highlighted in **bold**. Top-half: Sleep microbenchmark. Bottom-half: Fibonacci microbenchmark.

3.7.4 CPU clock frequency modifications

Table 3.4 shows the results in both energy used and completion times when CPU frequency modifications were made. Note that the Nexus 4 tests could not be performed at exactly half-speed, due to no matching frequency pairs being available in `scaling_available_frequencies`, provided by the governor. It is possible to recompile the governor to include unsafe and unsupported frequencies, but that may have damaged the device, and lack of availability of additional Nexus 4 devices made this unfeasible. However, as shown in the results, this did not affect my conclusion, as the 2x702MHz cores outperformed 1x1458MHz core, even though their effective cumulative frequency was only 1404MHz.

As shown by the bracketed values in Table 3.4, for the Fibonacci microbenchmark, more slower threads resulted in lower completion times rather than fewer faster threads. This is with one exception with the quad-core Nexus 4 NDK at 2x1458MHz where the difference in completion time was comparitively minimal.

However, when energy consumption is also considered, the recommendation to use more threads when computing the Fibonacci microbenchmark is not so conclusive. The dual-core Galaxy Nexus consumed far less energy with one thread at 700MHz rather than two at 350MHz, despite completion times being similar. Thus, for computing the Fibonacci microbenchmark, it is more effective to use a single faster thread with the Galaxy Nexus, and the Nexus 4 benefits from using more threads working at a slower frequency.

For the sleep microbenchmark, use of a greater number of threads was advantageous for reduced energy consumption in five of the six tests, with the only exclusion to this being the Nexus 4 NDK test at 2x1458MHz & 4x702MHz. On the contrary to the results of the Fibonacci microbenchmark, the Galaxy Nexus also consumed less energy when more threads were in use.

In terms of completion times, the quad-core Nexus 4 always produced faster performance with more threads, whereas the dual-core Galaxy Nexus benefited from a higher CPU clock frequency. This may be due to the Nexus 4 having poor integer operation performance as described previously. Therefore, the advantage of the additional threads is that more cores are used and the reduced amount of instructions per cycle in the Nexus 4 becomes less of an issue.

In Table 3.4, there are occurences where a bold value is present in both columns. There are six instances where this occurs, and in all but one of the instances, it is favorable to use the configuration that uses the least energy, because completion times are similar, whereas differences in energy are proportionally greater.

Overall, performance is largely as expected in Section 3.4.1 for these tests, with completion times being similar between the CPU freqency modification pairs. Thus, the energy consumption should provide recommendations for developers to make a dynamic choice, based on which configuration uses less energy. However, on the whole, the *safest* recommendation is to use more threads at half the speed for the purpose of reduced energy consumption.

3.8 Summary of microbenchmarks

In this section, I have created and run a series of tests, designed to evaluate the effectiveness of multicore processors in modern smartphones. Two benchmarking algorithms are used, an altered sleep and the Fibonacci algorithm, through both Java thread, and native pthread implementations. All tests are run on both a dual-core Galaxy Nexus, and a quad-core Nexus 4. Performance and power consumption were recorded for analysis, and sources of error were reduced.

The microbenchmarks are executed in order to justify the assumptions made in my models of multicore performance and energy consumption prediction, as the number of concurrently executing threads are increased. The accuracy of the models can be summarised in Figures 3.5, 3.7 and 3.11. The primary error in my model of energy required to compute a fixed workload concurrently is due to the multicore multiplier that is applied being too high when only 2 threads are in use. The prediction models have been validated through mapping the measured values to the predicted values with two devices with different hardware, thus I believe these prediction models will also apply to other devices.

Additionally, I demonstrate that when offloading work to multiple threads for processing, the number of threads used should equal the amount of physical CPU cores in the device, for best performance results. However, my energy consumption results show that developers should use the the amount of physical CPU cores *plus one*, threads for processing purposes. Doing so added a small performance overhead in few tests and no overhead in most, but a large energy saving can be made, depending on the CPU in the device.

Chapter 4

Evaluation

In this study, I have produced results that were used in evaluating the effectiveness of multicore processors in modern smartphones. The validity of the study was increased due to the reduction of variance amongst results. Analysis has shown that the measured values of the experiment largely follow my prediction models. The study covered a range of threading conditions, as a microbenchmark could be executed in several ways, further validating my conclusions.

Some issues were encountered with the quad-core Nexus 4 hardware in the experiment. Initially, the device powered down during testing due to overheating. The rear cover was removed, but the issue still occured. I discovered that, when Qualcomm manufacture the APQ8064 CPU chip present in the Nexus 4, the finished chips are grouped into 4 types, based on how it met quality standards. The 4 *binning* types are *slow*, *nominal*, *fast* and *faster*. The software kernel manages how much voltage is allocated to the CPU based on current clock frequency, but these voltages are slightly different based on the binning type. For example, the *faster* chip can operate at the same clock frequency as a *slow* chip, but at a lower voltage, resulting in less heat produced. Thus, due to persistent overheating during testing, I replaced the initial Nexus 4 device containing a *nominal* chip with one that contained a *fast* chip. No subsequent overheating occured and all tests were performed again for fairness.

There were a few areas of my study that could be improved. For the energy consumption analysis, the median average power reading over the complete testing period was used to calculate the energy used. For example, as the power was measured at 250kHz and downsampled to 10kHz for recording, a 3 second test had a power reading calculated from the median average of 30,000 values taken. The total median average was then multipled by the total completion time to calculate the energy used. The analysis could have been made slightly more accurate by calculating the energy used in the test by summing up the the median averages for each second in the test, where every second contains 10,000 readings.

The Fibonacci microbenchmark, when executed through the Dalvik Virtual Machine with the variable total workload, produced unreliable results due to the sleep between the iterations being too short. Therefore, the completion times were cumulatively longer as the test progressed, especially with a higher number of threads. The issue did not affect the native code execution because running the microbenchmark was an order of magnitude faster through the NDK, so the previous iteration had always completed. As a result, the Dalvik-based variable total workload tests with the Fibonacci microbenchmark were excluded from the analysis, and only the native results were used.

As seen in my analysis, some unexpected results were obtained from the Nexus 4 testing. These were due to the poor integer performance and reduced instructions per cycle of the Qualcomm Krait CPU, as described in previous sections. The results from the Nexus 4 also showed some instances with the sleep microbenchmark where the performance with native code was worse than through the Dalvik Virtual Machine, which is contrary to all previous work that compares the threading environments. It may have been beneficial to the study to also include another quad-core device, but with a different CPU that does not inherit these issues.

Additionally, more benchmarking algorithms that made a wider use of the CPU by using different types of operations, would further validate my conclusions, detailed further in Section 5.2.

Chapter 5

Conclusions & Future Work

5.1 Conclusions

This study has evaluated the effectiveness of multicore processors in modern smartphones, in terms of performance and energy consumption. I have presented two models of performance that estimate completion time, and a model of power that estimates the energy cost of performing a fixed batch of work with multiple threads. I have shown that these models are an accurate predictor of performance and energy cost, by creating a set of tests, based on the execution of microbenchmarks running through different threading configurations.

For the fastest **performance**, developers should use the same number of threads as the amount of physical CPU cores in the device. I have demonstrated this by showing how completion times increase when the number of threads are increased above the physical core number for the variable total work load. Completion times are inconsistent when the number of threads is increased above the amount of physical cores for a fixed total work load, ranging from 3% faster to 25% slower, meaning that the *safest* programming practice is to use the same number of threads as the amount of physical cores.

Recommendations for **energy consumption** are less clear as they are device dependant: in the dual-core Galaxy Nexus tests, 3 threads consistently used less energy than 2 threads, whereas the quad-core Nexus 4 tests showed that using 4 threads used less energy. However, the difference 2 and 3 threads on the Galaxy Nexus was a 19% saving, and the difference between 4 and 5 threads on the Nexus 4 was only a 2% cost, meaning that using the amount of physical cores, plus one for the number of threads, is the *safest* programming practice recommendation for reduced energy consumption. We have also seen from the energy consumption analysis that using more than one additional thread than the amount of physical CPU cores in the device uses more power and takes longer, therefore more energy is used unnecessarily.

I have shown throughout that developers have to explicitly make use of the physical cores, by creating threads in their app source code. In Android, work is not spread over the physical cores when using only the main UI thread, as confirmed in Appendix C. Additionally, as soon as additional threads are used, there is a large power overhead in using multicore, but energy used is reduced due to faster completion times. Therefore, for example, using two threads and thus two cores in the quad-core Nexus 4 is energy inefficient as the large multicore power overhead is applied, but the improved performance is not realised — so the only choice for number of threads with the quad-core Nexus 4 should be between 1 thread and 4 threads.

I believe that the presence of other apps performing processing work in the background would not affect the conclusions found in this study. The apps that consume the highest amount of CPU usage and energy are those that run in the foreground, as chosen by the user, such as CPU intensive games. Additionally, if all developers followed the conclusions summarised here, then even when their app is running in the background, it will consume less energy than if they were to use a poor threading configuration.

This study also compared the performance and energy consumption of the two threading runtimes. Considering that in Android, only the main UI thread can interact with the user, the primary purpose of any additional Java threads through the Dalvik Virtual Machine are for offloading any processing to for concurrent execution. If that is the case, then from my results, it would seem that where possible, all concurrent processing should be handled through pthreads using the NDK. However, due to the presence of extensive Java libraries, a developer must make the dynamic choice between reduced development time in making an app, and better performance and reduced energy efficiency once an app has been published.

5.2 Future Work

This study has evaluated the effectiveness of multicore processors in modern smartphones, in terms of performance and energy consumption. In future work, these findings could be used extended across many devices to define a database of the most energy efficient methods of using multiple threads with different mobile chipsets. This database would be referred to when building the operating system software and kernel for a device, and the most efficient methods for that device would be included in the build. Any third-party apps would then make an API call that follows the defined threading method when offloading work to threads. For example, a programmer may need to perform some processing work, and could use a predefined API that wraps the work in the most energy efficient threading configuration at application runtime. This threading configuration would be based on information from the database of mobile chipsets, and would consist of the most energy efficient number of threads to make use of the physical cores, and the threading environment for the work.

Furthermore, this study has only considered two benchmarking algorithms, that make use of integer operations and deep recursive call stacks. My benchmarking tool could be extended to include additional benchmarking algorithms that make wider use of CPU operations. These additional benchmarks could emulate typical mobile usage patterns, as found by data recorded by the Device Analyser Android app [22].

Additionally, as seen by the graphs of residuals in the analysis sections, the performance models looked to be more accurate than the energy consumption models. Future work would consider the addition of another variable factor to the energy consumption model, based on these results.

As described in Section 4, the CPU chip in the quad-core Nexus 4 is grouped into one of four categories at the end of the manufacturing stage, known as *CPU binning*. I witnessed variations in heat produced between two Nexus 4 devices due to changes in the voltage allocated to the CPU. Future studies could analyse whether these differences of chips in the same model of device translate into differences in threading performance in terms of performance and energy consumed.

This controlled experiment involved running the majority of tests with both devices at their highest CPU clock frequency by changing the current governor and setting the frequency in the initialisation phase of my client app. Further studies could include the multithreaded performance and energy consumption when using the unmodified kernel, and default *ondemand* governor that adjusts clock frequency dynamically.

Both devices used in the experiment contained CPUs based on the ARM architecture. Alternatively, further work could consider the x86 architecture with Intel CPU chips using an x86 port of Android. The results of this work could be mapped to my performance and power models to discover whether the same conclusions are reached with a different architecture.

Appendix A

Configuring Android devices for performance and power consumption benchmarking

I modified the software configuration of the two devices used to complete my experiment, in order to reduce variance in the measured values. Both the devices were Google *Nexus* devices, and hence ran a stock version of Android 4.2.2. The following changes were made to the software:

- The kernel was replaced with a custom-made kernel, that I built from AOSP source. By disabling all other cpufreq governors in the `make menuconfig` at build time, I ensure that the device can only use the *performance* governor.
- The devices were *rooted*, and root permissions were granted to my benchmarking client app. Thus, CPU frequency could be set as part of the app initialisation phase after the screen is turned off, by echoing the desired frequency to the `scaling_cur_freq` file, in accordance to those frequencies specified in the `scaling_available_frequencies` file.
- Non-essential apps were removed from the devices, including Maps and Talk.
- Any background services that were not required for the client app to run were force closed prior to starting the tests.
- The devices were not logged in to a Google account.
- Autosync was deactivated.

- The screen and notification LEDs were turned off during testing.
- A static black background was used rather than the default live wallpapers.
- Icons and widgets were removed from the homescreen. Additionally, all icons were removed from the application launcher.
- All forms of radio communication were switch off by using Aeroplane Mode. Near Field Communication and GPS were also disabled.
- No contact was made with the touchscreen or the physical buttons during the testing period.

Appendix B

Photos of experimental setup

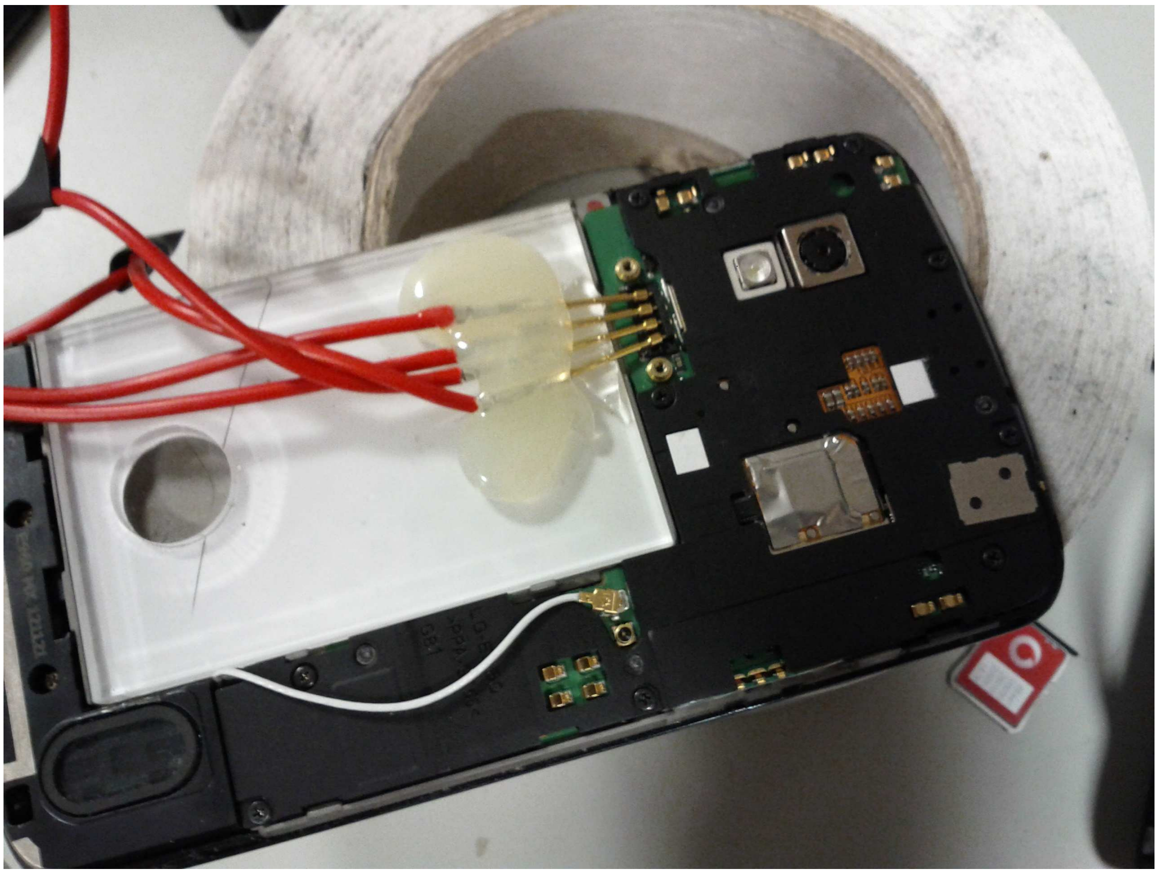


Figure B.1: The quad-core Nexus 4 with the rear cover removed. The four sprung contacts on the dummy battery push onto the power terminal pins.

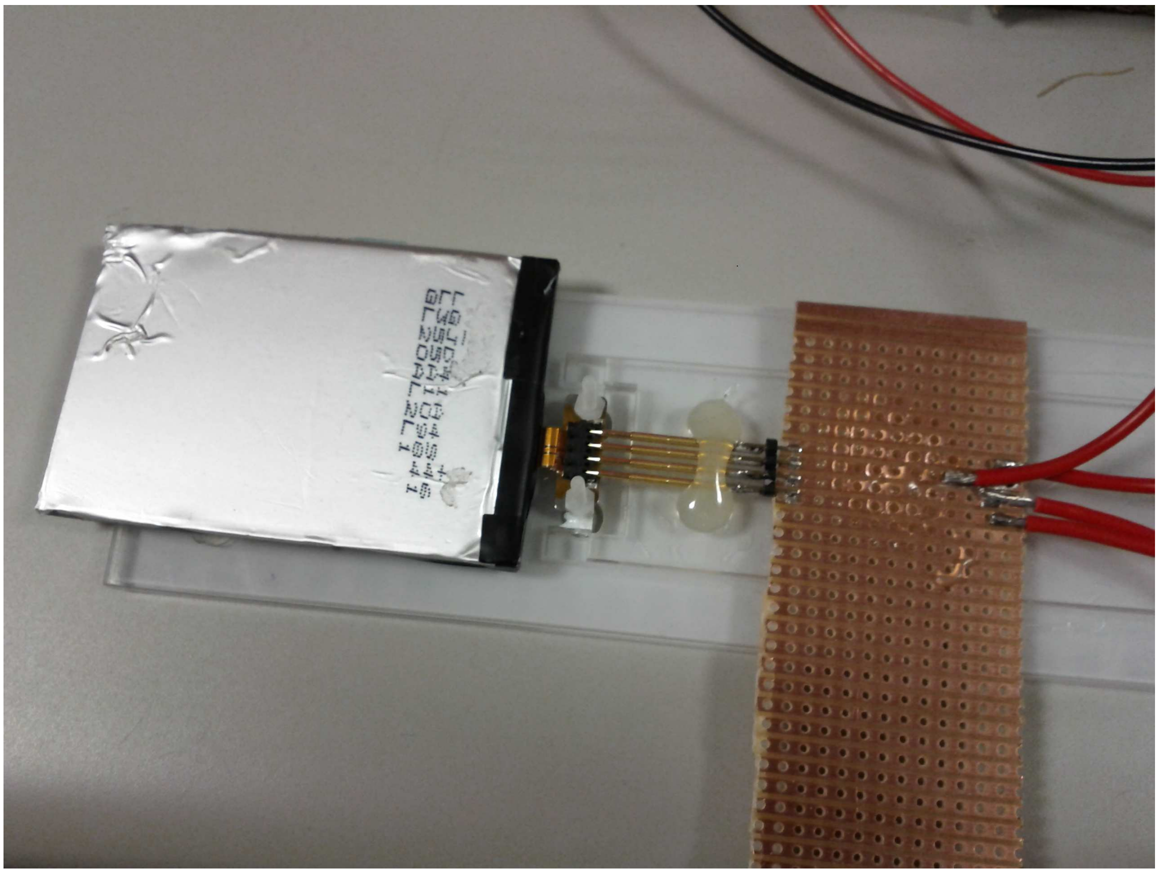


Figure B.2: The battery holder for the Nexus 4 battery. The original battery is screwed into the holder, and four pins slide into the contacts in the battery connector plug.

Appendix C

Recorded CPU usage per core

1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
c0: 8%	c0: 94%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 6%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 10%	c0: 97%	c0: 100%	c0: 100%	c0: 97%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 79%	c1: 95%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 0%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 6%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 94%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 0%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 9%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 11%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 9%	c0: 100%	c0: 94%	c0: 100%	c0: 99%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 89%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 11%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 8%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 11%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%
c0: 4%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%	c0: 100%
c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%	c1: 100%
c2: 0%	c2: 0%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%	c2: 100%
c3: 9%	c3: 0%	c3: 0%	c3: 100%	c3: 100%	c3: 100%	c3: 100%	c3: 100%

Table C.1: Sample of recorded CPU usage (from N4 sleep fixed DVM test).

Bibliography

[1] David Ehringer. The Dalvik Virtual Machine Architecture. *Techn. report (March 2010), elastos.org*, 2010.

[2] Ahmad Rahmati, Angela Qian, and Lin Zhong. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, MobileHCI '07, pages 265–272, New York, NY, USA, 2007. ACM.

[3] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.

[4] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, 15(1):179–198, 2013.

[5] C.S. Ellis. The case for higher-level power management. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 162–167, 1999.

[6] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[7] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM.

[8] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 315–330, New York, NY, USA, 2010. ACM.

[9] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 41–50, New York, NY, USA, 2012. ACM.

[10] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 40–, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[13] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[14] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[15] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[16] John L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[17] Aubrey-Derrick Schmidt Leonid Batyuk. Developing and benchmarking native linux applications on android. *MobileWireless Middleware, Oper-*

ating Systems, and Applications, Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 7:381–392, 2009.

[18] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[19] Dow Lin, Lin and Wen. Benchmark Dalvik and Native code for Android System. *IBICA '11 Proceedings of the 2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pages 320–323, 2011.

[20] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[21] A. Rice and S. Hay. Decomposing power measurements for mobile devices. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 70–78, 2010.

[22] Daniel Wagner, Andrew Rice, and Alastair Beresford. Device analyser. In *HOTMOBILE 2011 12th Workshop on Mobile Computing Systems and Applications*, 2011.