2012

Android Benchmarking for Architectural Research

Ira Ray Jenkins

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ANDROID BENCHMARKING

FOR

ARCHITECTURAL RESEARCH

By

IRA RAY JENKINS

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2012

Ira Ray Jenkins defended this thesis on July 2, 2012.

The members of the supervisory committee were:

Gary Tyson
Professor Directing Thesis

David Whalley
Committee Member

Piyush Kumar
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with the university requirements.

To Nana Bo, a woman among women, the grandest of grandmothers, the surest of hind catchers and my eternal best friend.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADK Accessory Development Toolkit

ANR Application Not Responding

API Application Programming Interface

CPU Central Processing Unit

DSP Digital Signal Processing

GPS Global Positioning System

ICS Ice Cream Sandwich

I/O Input and Output

ITU International Telecommunications Union

JNI Java Native Interface

NDK Native Development Toolkit

PC Personal Computer

RAM Random Access Memory

SDK Software Development Toolkit

SoC System on a Chip

SPEC Standard Performance Evaluation Corporation

TLB Translation Lookaside Buffer

UI User Interface

VM Virtual Machine

ABSTRACT

Mobile platforms have become ubiquitous in our society; however, classical benchmarks have not kept up with their growth. When not entirely incompatible, current benchmarking techniques often provide very little useful information about these systems. Herein is proposed an open-source framework, based on existing technologies, for constructing new benchmarks targeting the Android operating system, the current leading mobile platform. This framework is used to construct several micro-benchmark kernels, in both Java and native C/C++, as well as to demonstrate the conversion of existing open-source applications into relevant mobile benchmarks, useful for architectural research. We provide profiles for each benchmark. In addition, a method is proposed for leveraging the speed of the Android emulator to assist in benchmark development on architectural simulators. Finally, in an effort to provide a stable development environment and promote the use of the framework, a virtual machine including all open-source materials is made available.

CHAPTER 1

INTRODUCTION

1.1 Benchmarking

As computer architecture has matured, the increasing complexity of computer systems has made it challenging to adequately characterize a system's capabilities, especially in comparison to other systems. One of the goals of any computer manufacturer is to increase the sales of their respective platforms. To this aim, manufacturers and researchers have proposed many standard methods of system characterization throughout the years; often steering consumers in wildly incorrect directions based on skewed statistics and number gaming.

For many years, consumers have been subjected to the so-called *megahertz myth*, based on the misconception that a faster clock rate, or cycle frequency, produces a faster computer. This assumption belies the impact of instruction set architectures as well as advances in the micro-architectural exploitation of instruction level parallelism. From hardware to software, there are a multitude of factors to consider when comparing different systems.

In an attempt to alleviate the confusion and frustration of system characterization, benchmarks were introduced. Benchmarks are programs designed to demonstrate specific program behavior, like the performance of floating point operations or the latency of memory references. There exists a wide variety of benchmarks, including micro-benchmarks, engineered to encapsulate a single task; synthetic benchmarks composed from profiling statistics, such as the number of loops in a program; and application benchmarks based on real-world applications.

While great in theory, there have been many flaws in benchmark implementations over the years. Again, in efforts to promote themselves over the competition, companies have been known to optimize systems for industry standards, exploiting known benchmark behavior, such as loop structures or specific mathematical operations [22]. Furthermore, benchmarks are often considered proprietary, requiring licensing fees for use and/or publishing rights. These exclusive practices can lead to secret internal methodologies and an unwillingness to distribute source code. This might deter fraudulent behavior, but from a research standpoint, it presents the challenge of not truly understanding what a benchmark is doing or what exactly the results might mean. For trustworthy and meaningful results, benchmarking requires honesty and transparency from all parties involved: creators, vendors, and end-users.

1.2 Mobile Computing

Embedded computing is the act of including a compute device with domain specific functionality, such as digital signal processing, within a larger system. Embedded systems surround us, making up the vast majority of devices we interact with daily. They are present in everything from microwaves and anti-lock braking systems to cell phones. Many benchmarks have been targeted for embedded computing, such as Mibench [29], DSPstone [51], and the suites developed by the Embedded Microprocessor Benchmark Consortium [45].

In recent years, mobile phones have eclipsed the embedded systems category, giving rise to the new term SoC, or *system on a chip*. These systems are no longer single task processors but general purpose computers that can run full operating systems. They include a myriad of their own embedded processors for sensors like GPS, gyroscopes, and accelerometers. In addition to the standard radio communication, they support short range communication with bluetooth or WiFi, as well as audio/video playback and recording. More recent developments include promising multicore architectures [42, 43, 47].

In 2011, smartphone sales topped PC sales for the first time, shattering the barrier with almost 490 million units sold [17]. Even more, the International Telecommunications Union estimates that in developed countries we are quickly heading toward market saturation, with nearly 120 cellular subscriptions per 100 people [33]. Unfortunately, the advances in mobile platforms have largely outpaced the benchmarking industry. Many are still using outdated or irrelevant benchmarks, such as Whetstone [18], LinPack [19], SPEC [46], NBench [14], and Fhourstones [48] to make assumptions about these devices. These applications often showcase highly optimized kernels, with large numerical solvers, compiled in languages like C or Fortran.

Several companies have created benchmarks specifically for mobile devices like AnTuTu Labs [4], Aurora Softworks [6], and Kishonti Informatics [34]. These companies have opted to go with a proprietary model, requiring licensing fees for use or source access. While these benchmarks may contain relevant coverage for mobile platforms, they provide little information at the architectural level, such as the composition of instructions or cache behavior. This closed-source nature presents a barrier to researchers and developers wishing to port these applications to architectural simulators or other environments.

1.3 Goal

It is the goal of this work to produce a set of free, open-source benchmarks that are relevant to the Android platform and useful for architectural research. We choose the Android operating system based on its popularity and open-source model.

The Android operating system was introduced in 2007 and has steadily gained popularity every year. It became the top mobile platform in 2010 and has continued its climb with projections of 60% market share in 2012 [23, 24, 25, 26]. Perhaps helping its strong market performance, Google's mobile platform comes with a robust, open-source API as well as software, native and accessories development kits. Given its current standing, people are beginning to look at Android as a research platform and developing benchmarks for it, such as BBench [30], a browser benchmark, or AMBench, a multimedia benchmark [37].

This work departs from the previous in our focus on architectural research; specifically, the desire to run our benchmarks within a simulated environment capable of producing system profiling statistics. In addition, we do not focus on explicit application genres, such as web browsing or multimedia. We instead present a framework for rapid benchmark development and the conversion of existing real-world applications into new benchmarks.

The remainder of this work is organized as follows: In Chapter 2, we consider the challenges to architectural benchmarking with Android and propose a framework for benchmark creation. Chapter 3 details a suite of benchmarks, including their analysis, and a prototype for leveraging emulator speeds for architectural simulation. Finally, we conclude with a proposal to assist in benchmark development and a look at future work.

CHAPTER 2

CHALLENGES

The intention of any good benchmark suite is to provide a deterministic, stable representation of system-relevant workloads. Within architectural research, the use of simulated environments is a commonly accepted methodology. Although less precise than real machines, simulators have the advantages of price and customization. They allow researchers to prototype a wide variety of architecture models and experiment with potential new features without the cost of new hardware for every possible combination. Two primary challenges exist to the creation of a benchmark suite, for architectural research, using the Android operating system; particularly, these are choosing a simulation platform that supports the ARM architecture, which allows full-system simulation, and overcoming the inherent limitations of the Android environment itself.

2.1 Environment

In searching for a simulation environment, we desire an open-source, easily customizable simulator. SimpleScalar [13] meets these basic criteria: being exceedingly simple to modify and supporting multiple instruction set architectures, including Alpha, ARM, and x86. Unfortunately, SimpleScalar does not support full-system simulation, nor is it actively maintained.

Qemu [8] is a natural option, considering the Android emulator distributed with the software development kit is based on an earlier version of it. Although not a simulator, Qemu is a free, open-source emulator and machine virtualizer. The key difference between simulators and emulators is the lack of internal state fidelity within emulators. One of the key advantages of Qemu over other platforms is its speed; it uses dynamic binary translation and supports hardware virtualization, previously via KQEMU [9] and more recently with KVM [36].

The major disadvantage of Qemu for this research is its lack of cycle-accurate behavior and profiling abilities. Several attempts have been made to instrument Qemu [28, 31, 41]. As an initial experiment, a memory trace in the Dinero IV [21] format was created for the ARM target, based largely on work previously done [49]. While it is possible to modify Qemu, the resulting performance hit from instrumentation quickly negates much of the original speed benefits at the cost of significant efforts required for development and upkeep.

The gem5 [11] architectural simulator is a free, open-source combination of the M5 [12] and GEMS [40] simulators. M5 contributes multiple ISA support, including Alpha, ARM and x86, as well as several CPU types: atomic, timing, in-order, and out-of-order. GEMS confers much of the memory system, including a domain specific language for easily defining cache protocols. The gem5 system is highly object-oriented, providing the simulation framework in C++ and configuration facilities via Python. With its robust feature set, gem5 presents an easily customizable simulation environment.

For this work, we simulate an ARMv7a Cortex-A9 [5] architecture on a RealView PBX development board. The gem5 ARM port supports both Android 2.3, Gingerbread, and 4.0, Ice Cream Sandwich; however, ICS makes extensive use of hardware acceleration for graphics, which is not fully supported by the simulator. Furthermore, nearly 65% of all Android devices are currently running Gingerbread [2]. Our experimentation includes both atomic and out-of-order CPUs. The atomic model represents a basic in-order machine executing a single instruction per cycle. The more detailed out-of-order model corresponds to a superscalar, pipelined machine with memory latencies.

2.2 Android Limitations

Benchmarking mobile devices presents several challenges, including storage, security, and synchronization. While mobile device memory capacity is growing, it has not yet reached the levels of current desktop PCs, with terabytes of storage and tens of gigabytes of RAM. Naturally, space provides a substantial barrier to many existing benchmarks, either from application or input size.

In addition, the Android platform is a privilege-separated operating system. This means that each application, along with certain operations and disk areas, maintain an explicit set of permissions. The operating system relies on the Dalvik virtual machine; however, security is mainly enforced by user and group id. The overall philosophy is to restrict any accesses that might degrade user experience by harming other applications or the system itself. As many benchmarks read from or write to files on disk, permissions need to be handled properly to ensure correct execution.

The primary challenge comes from the asynchronous nature of modern operating systems, including Android. In a typical benchmark setting, a user would create a script to run a set of benchmarks, then later collect and analyze statistics of interest. The Android activity lifecycle, shown in Figure 2.1, creates certain issues with this traditional approach. Namely, the application launch mechanism used within Android, called *intents*, complicates classic scripting techniques.

Intents work as an asynchronous message passing facility. They typically include a general description of the type of application to launch, for example a web browser, without requiring an explicit knowledge of which application may respond to such a request. To prevent needless waiting, once an intent has been fired, the user regains control and can continue with their workflow. For many user interactions, this method is preferable; however, a script launching an application via intents would have no knowledge of if or when an application might become active.

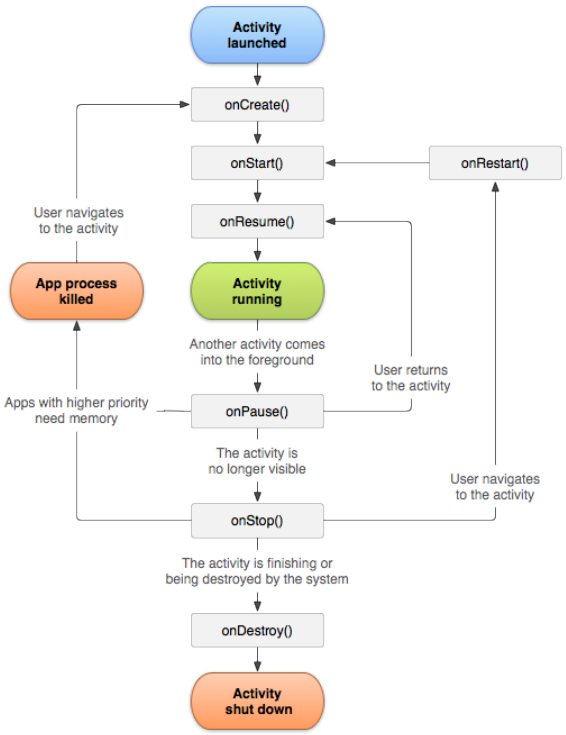

Figure 2.1: Android Activity Lifecycle

To complicate matters more, Android separates the lifecycle into multiple stages, from creation to destruction. An application could be considered active during the third stage, in which it is actually displayed and becomes ready for user input. The prior two stages, though typically fast, could be arbitrarily compute intensive, loading files and managing the previous running state. From a scripting perspective, there is again no way to acknowledge if or when an activity has become functional. The final complication arises from the method for exiting a running application. In a typical workflow, a user may interact with multiple applications. While one program is active it is considered to be "focused" in the foreground, while other activities may be suspended in the background.

According to the lifecycle within Android, once an application no longer controls screen real estate it is eligible to be collected by the system. However, if there are no outstanding resource requests, and unless explicitly terminated by the user, an application can stay hidden almost indefinitely. This can potentially alter successive launches of a benchmark application by allowing the system to cache the previous state and quick-launch programs from the background.

CHAPTER 3

DESIGN

There are a few ways to deal with the above mentioned issues when designing a benchmark suite. To avoid problems with the security features, a device can be "rooted", granting complete access to a device and making it possible to bypass many of the security permissions within Android. While a relatively simple process, rooting has the side effect of potentially altering the execution of certain programs.

To provide a certain level of synchronous behavior, busy-wait polling or blocking I/O can be used. The command line *sleep* functionality can be used to wait until a specified time; however, this method does not truly solve the problems presented by the activity lifecycle. From a script, a blocking pipe can be created. Once a benchmark has completed and is ready to signal an end, the other end of the pipe can be opened, freeing the script process to continue.

These solutions handle the general issues mentioned; however, none of them address the issue of user interaction. The Android operating system is highly event driven; without a method to interact with the system many workloads become infeasible to simulate.

3.1 Proposed Solution

We propose a solution, based on using the Android testing framework, shown in Figure 3.1. The testing framework is based on JUnit 3, the Java unit testing framework, with additional features integrating it with the Android operating system. This approach has been suggested in other benchmark suites [37]; however, implementation details have not been provided. Our work provides these details, in addition to evidence for this method's viability in architectural research and the first benchmark examples, based on this framework, running within an architectural simulator.

This system gracefully handles all of the previously mentioned problems. The testing framework allows the control of application lifecycle, runs in the same process as the application under test and facilitates the injection of user input events. The procedure is to create test packages containing JUnit assertions and Android instrumentation. This package is then given to an InstrumentationTestRunner, which does all the heavy lifting to connect a test to the appropriate application.

Android provides several external methods for interacting with devices, such as *monkeyrunner*; however, these techniques do not work with a running gem5 simulation. The

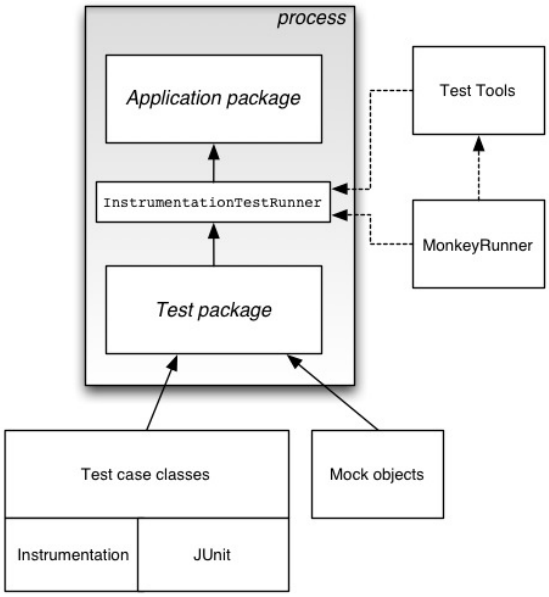

Figure 3.1: Android Testing Framework

one caveat of using the Android testing framework is the necessity of source code for any application tested. To work within Android's permission system, any application and its corresponding test cases[1] must be signed with the same signature, preventing misuse of the framework. Since a goal of this work is to be both free and open-source, this limitation is acceptable. Any benchmarks that are used should also be freely available and open-source.

3.2 Benchmark Framework

The benchmark framework can be seen in Figure 3.2. The test package, aptly named BenchmarkRunner, contains several unit tests, each responsible for launching a separate benchmark. Here individual benchmarks are grouped together, forming a single application. The BenchmarkApp consists of BenchmarkTasks; each task encapsulating a system-relevant workload, the benchmark itself. For the purposes of benchmarking, test cases are kept relatively sparse and simple in an attempt to mitigate any potential instrumentation overhead.

Tracing the execution path, a script triggers the benchmark process by calling an InstrumentationTestRunner, specifying the necessary BenchmarkRunner. The BenchmarkRunner extends the functionality of Android's ActivityInstrumentationTestCase2, a special test case that allows the application under test to run in a normal execution environment. Other test cases create a mock environment in which interaction with other system activities is not possible. Each test function within the BenchmarkRunner packages an intent with which

[1]In the remainder of this work, a test or test case is synonymous with a benchmark application launcher.

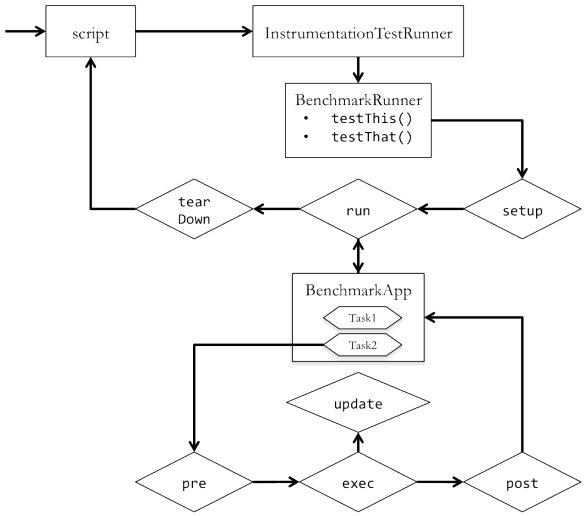

Figure 3.2: Benchmark Framework

benchmark to run and any extra information it may need to successfully operate. Once the intent has been launched, the BenchmarkRunner synchronizes on the new benchmark application, awaiting notification of its completion. This synchronization avoids the possible overhead of busy-wait polling. Finally, when a BenchmarkTask has executed the workload, the BenchmarkApp notifies the BenchmarkRunner of its completion, subsequently returning control back to the initiating script.

The BenchmarkTasks are based on Android's AsyncTasks, allowing a benchmark to be segregated on its own thread and avoid causing an ANR, or "application not responding", error. The BenchmarkTask object encapsulates the classical idea of a benchmark, with pre- and post-execution methods used to setup any necessary environment and do bookkeeping after the benchmark has run. During the execution phase of the benchmark process, the BenchmarkTask inherits the ability to send callbacks to the BenchmarkApp, updating it of any progress. While it is not necessary, and may produce some overhead if not used judiciously, it does allow an otherwise unknowing user to keep track of the execution status of a benchmark.

This framework solves all of the previous complications with architectural benchmarking on the Android platform. The BenchmarkRunner and the BenchmarkApp, either a custom creation or existing application, are both signed with the same developer's signature and execute within the same process. This resolves any security issues the operating system may present. In addition, the setup and teardown methods of the BenchmarkRunner help assure that an application is not currently running when a benchmark begins and is successfully ended upon its completion. Without this functionality, benchmark results could be skewed by the skipping of certain initialization and destruction sections of the activity lifecycle.

3.3 Initial Analysis

A sample application, screenshot shown in Figure 3.3, is used to evaluate the performance impact of this framework. The ProgressBar application demonstrates an ideal setting in which no user interaction is required and the application runs for a set period of time, destroying itself when complete. The program creates and displays a simple progress bar, which incrementally increases by ten every second for ten seconds before stopping itself.

Figure 3.3: ProgressBar Application

The baseline for this experiment consists of ten runs, scripted from the command line. The script launches the application and sleeps during the known execution time frame. Its proper execution was manually confirmed. In contrast, the benchmark framework is used to instrument and launch the ProgressBar application, again ten times consecutively. The profiling comparison is presented in Table 3.1.

Table 3.1: ProgressBar Profile

	Baseline		**Instrumented**	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	12.1760474	3.632577722	11.40602678	2.021143638
Cycles	3331194327	35.12993991	3562212723	15.93138658
Instructions	1565649673	28.52781996	1676047434	11.61630537
DTLB accesses	379839604.7	38.81289279	426885316.1	18.11673401
DTLB hit rate	99.3155971	0.158898726	99.2837604	0.111735633
ITLB accesses	177438115.4	35.4344382	195502228.4	15.97946189
ITLB hit rate	99.9081252	0.067586574	99.8773481	0.043124889
Dcache accesses	311997377.6	38.90743192	354910895.1	17.73166591
Dcache hit rate	97.350687	0.318992858	97.350687	0.261765338
Icache accesses	177093050.2	35.33575148	195075298.8	15.92593696
Icache hit rate	99.0005315	0.458556218	98.8426668	0.371738411

We find that the benchmark framework and associated instrumentation techniques represent about 10% overhead on average. However, the average variation for each statistic of the baseline execution is nearly double its counterpart in the instrumented version. This confirms that the testing framework is an acceptable method for benchmark development. While some overhead is incurred, applications are far more stable than when manually run. Furthermore, any overhead will likely pale when compared to the concurrent running of multiple applications, as seen in a typical Android workload.

CHAPTER 4

IMPLEMENTATION

4.1 Benchmarks

To showcase the proposed framework, a suite of benchmark applications is developed. The following applications primarily demonstrate the ease of benchmark creation within the framework; however, their profiles are included in Appendix A for reference.

4.1.1 Micro-benchmarks

Micro-benchmarks are specific, single-task kernels that help demonstrate certain aspects of a system. The implemented benchmarks represent reasonably common tasks within a larger project, such as searching and sorting. Many of the algorithms use techniques like recursion, a common performance problem for virtual machines, and dynamic programming. The key design ideas were simplicity and relevance, with implementations stressing uniformity over optimizations. The implemented micro-benchmarks are as follows:

- Dijkstra's Single-Source-Shortest-Path
- Extended Euclid
- N^{th} Fibonacci
- Huffman Coding
- Knuth-Morris-Pratt Substring Search
- Longest Common Subsequence
- Longest Common Substring
- Matrix Multiply
- QuickSort
- Subset Sum

The Android platform is largely written and programmed in the Java language. Despite encouraging the use of Java, Android supports development in native languages such as C and C++ through its Native Development Toolkit. This toolkit allows the creation of native code libraries and their subsequent packaging into Android applications. The use of native code within Android quickly increases the complexity of development. While performance increases are possible [38, 39], naive native implementations rarely achieve significant gains.

Performance notwithstanding, each algorithm is implemented in Java using standard Android APIs, as well as in native C/C++ using the Android Native Development Toolkit's Java Native Interface. Where possible, the implementations were kept similar, using the same overall code structure for uniformity and simplicity.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Figure 4.1: Matrix Multiply Algorithm

Matrix Multiply. Here we detail the implementation of the matrix multiply micro-benchmark and include its architectural profile. The matrix multiplication is a standard, un-optimized matrix product algorithm implemented as three loops, as shown in Figure 4.1. The small version of matrix multiply reads in and multiplies two 25x25 matrices of floating-point numbers from a file on disk. The larger version multiplies two 200x200 matrices and exhibits nearly double the performance for each performance counter statistic. When comparing their performance, shown in Tables 4.1 and 4.2, the Java version displays much greater variation. This could be the result of background noise, such as scheduled system processes or the garbage collection running.

Table 4.1: Matrix Multiply – Java – Profile

	Small		**Large**	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.659548	0.129703346	1.4984035	21.56711367
Cycles	1319027241	0.129688574	2956194361	22.21673163
Instructions	640727127.5	0.142331039	1012068399	30.01275155
DTLB accesses	149343144	0.347604617	291869112.8	27.69121058
DTLB hit rate	99.6543682	0.00251076	96.5617965	0.862048629
ITLB accesses	71406458	0.408747362	119977761	29.72608697
ITLB hit rate	99.9228172	0.000242726	99.8726684	0.01152484
Dcache accesses	124936833.8	0.388566059	241486799.3	27.89394689
Dcache hit rate	97.7584639	0.034113609	97.276894	0.118363862
Icache accesses	71288221.75	0.4082315	119708590	29.73291549
Icache hit rate	99.0565617	0.020756093	98.4855368	0.030006031

Table 4.2: Matrix Multiply – Native – Profile

	Small		**Large**	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6591975	0.299178812	1.576788	1.201178063
Cycles	1318185583	0.310930077	3110774210	1.135262667
Instructions	640025586.8	0.218578123	1149187422	1.652672123
DTLB accesses	149125979.8	0.499738711	296762255.5	3.31151725
DTLB hit rate	99.6526004	0.006130445	96.909157	0.013744519
ITLB accesses	71324711.25	0.536057614	126539822	2.244071979
ITLB hit rate	99.9211501	0.000506622	99.923394	0.024014068
Dcache accesses	124709896.5	0.543148704	247934687.5	3.130713439
Dcache hit rate	97.7785234	0.001776936	97.7204305	0.13057266
Icache accesses	71204516.75	0.535361275	126345982.3	2.219597308
Icache hit rate	99.0226625	0.026728266	99.1010885	0.103378189

Dijkstra's Single-Source-Shortest-Path. This benchmark represents a common graph search algorithm. Provided with an edge-weighted graph and a source vertex, this algorithm proceeds to find the shortest distance between the source and every other vertex.

Extended Euclid. This benchmark implements recursively the extended euclidean algorithm for finding the greatest common divisor of two numbers, as well as solving Bezout's identity, $ax + by = gcd(a, b)$.

N$^{\text{th}}$ Fibonacci. This benchmark finds the n^{th} number of the Fibonacci sequence, using its defining recurrence relation $F_n = F_{n-1} + F_{n-2}$.

Huffman Coding. This benchmark uses a well-known, entropy encoding algorithm. Text is read from a file, compressed in the requisite format and then decompressed.

Knuth-Morris-Pratt Substring Search. This benchmark is an optimized string search algorithm, finding a search pattern within a given string. Here, two strings are read from a file on disk and compared. Whenever a mismatch of items is found, previously matched characters are not re-examined. The input is the same for each string comparison benchmark.

Longest Common Subsequence. This benchmark solves the problem of finding a longest subsequence of non-consecutive items within a set. The input is the same as the above benchmark.

Longest Common Substring. Unlike the previous benchmark, this program finds the longest subsequence of consecutive items within a set. The input is the same as the above benchmarks.

QuickSort. This benchmark uses the well-known, simple sorting algorithm. The implementation is stable and uses in-place partitioning. To allow future comparisons with classical benchmarks, the input sets are derived from MiBench's automotive category qsort benchmark.

Subset Sum. This benchmark discovers whether or not it is possible for a subset of a given set to sum to a given value. It is implemented by recursively either taking or not taking a set member.

4.1.2 API Demos

The Android SDK provides many sample applications showcasing the capabilities of the system APIs. As an intermediate step between micro-benchmarks and full-scale applications, these demos are instrumented to measure ease of transition from standalone applications to runnable benchmarks. Unless otherwise stated, the source code of the original application is not modified. This restriction is to further help in gauging transitional difficulty. Modification of the original source has the potential to alter the behavior and memory footprint of the application. As a result of this limitation, applications not designed for testing can require intensive techniques, such as reflection, to properly instrument.

AudioFx. The AudioFx demo, screenshot shown in Figure 4.2, plays a 53 second audio clip while visualizing the waveform of the audio and providing an equalizer for adjusting sound properties. This benchmark is instrumented to navigate through several menus and

Figure 4.2: AudioFx Application

launch the AudioFx application. In this example, the audio player must be reflected so that the test application knows when the audio clip is finished. Shown in Table 4.3, the demo application shows very little variation over ten runs.

Table 4.3: AudioFx – Atomic – Profile

	Mean	**Co. of Var.**
Runtime (sec)	56.7071455	0.495669544
Cycles	113413651249	0.49558621
Instructions	9761405726	4.984808126
DTLB accesses	4027482050	5.041953521
DTLB hit rate	99.89654881	0.022040119
ITLB accesses	9943508533	4.998294654
ITLB hit rate	99.99229389	0.001651655
Dcache accesses	3908616500	5.018261443
Dcache hit rate	99.55096760	0.061568786
Icache accesses	9943507327	4.998287526
Icache hit rate	99.68972167	0.007476754

JetBoy. The final sample application coming from the Android SDK is the JetBoy arcade game, screenshot shown in Figure 4.3. This application allows a user to shoot asteroids for a period of time, with the goal of shooting 50 before the time is up. Like the AudioFx demo, reflection is required here to access the Jetboy view, enabling the benchmark to know when the game has finished. Instrumentation is added to shoot twice every second. The source code of this application was modified to make the asteroid field more predictable, specifically replacing calls to a random number generator with an integer modulus. The JetBoy application shows even less variation than the AudioFx demo, showcasing great cache and TLB utilization.

Figure 4.3: JetBoy Application

Table 4.4: Jetboy – Atomic – Profile

	Mean	Co. of Var.
Runtime (sec)	74.2030667	0.205923156
Cycles	148403275843	0.205890164
Instructions	141803076219	0.166253904
DTLB accesses	29047114898	0.309590518
DTLB hit rate	99.88755034	0.0010826091
ITLB accesses	141954586316	0.167963902
ITLB hit rate	99.99689692	0.0000206619
Dcache accesses	25706971328	0.33743948
Dcache hit rate	98.29602632	0.0022273621
Icache accesses	141954585352	0.1679638
Icache hit rate	99.95526982	0.0007073207

4.2 Event Recording

Benchmarking within the gem5 simulation environment can be several orders of magnitude slower than the Android emulator. While graphical input is available, it is tedious and slow, requiring tens of seconds to process single click events on the fastest, atomic CPU. To answer this challenge, a prototype system has been developed to allow the Android emulator to be used to emulate, record, and playback workflows.

These workflows can contain arbitrarily many input events, such as clicks, button presses, swipes and more. This system is not limited to the Android emulator. Identifying the input devices available on real hardware systems allows the creation of workflows on these devices as well.

The Android operating system uses standard Linux input device protocols. In particular, each input action is recognized with a device identifier, an input type, an input code, and an input value. The event specification and playback translation can be seen in Figure 4.4.

In addition to playing back previous workflows, these events can be translated into Java statements, to later be included as instrumentation code within a benchmark. Figure 4.5 shows the results of capturing the user input of “Hello” and translating it into Java code.

```
/dev/input/event0: 0001 002a 00000001 → sendevent /dev/input/event0 0001 42 1
/dev/input/event0: 0001 0023 00000001 → sendevent /dev/input/event0 0001 35 1
/dev/input/event0: 0001 0023 00000000 → sendevent /dev/input/event0 0001 35 0
/dev/input/event0: 0001 002a 00000000 → sendevent /dev/input/event0 0001 42 0
/dev/input/event0: 0001 0012 00000001 → sendevent /dev/input/event0 0001 18 1
/dev/input/event0: 0001 0012 00000000 → sendevent /dev/input/event0 0001 18 0
/dev/input/event0: 0001 0026 00000001 → sendevent /dev/input/event0 0001 38 1
/dev/input/event0: 0001 0026 00000000 → sendevent /dev/input/event0 0001 38 0
/dev/input/event0: 0001 0026 00000001 → sendevent /dev/input/event0 0001 38 1
/dev/input/event0: 0001 0026 00000000 → sendevent /dev/input/event0 0001 38 0
/dev/input/event0: 0001 0018 00000001 → sendevent /dev/input/event0 0001 24 1
/dev/input/event0: 0001 0018 00000000 → sendevent /dev/input/event0 0001 24 0
```

Figure 4.4: Event Playback

```
/dev/input/event0: 0001 002a 00000001 → sendKeys(KeyEvent.KEYCODE_SHIFT_LEFT);
/dev/input/event0: 0001 0023 00000001 → sendKeys(KeyEvent.KEYCODE_H);
/dev/input/event0: 0001 0023 00000000 ↗
/dev/input/event0: 0001 002a 00000000 ↑
/dev/input/event0: 0001 0012 00000001 → sendKeys(KeyEvent.KEYCODE_E);
/dev/input/event0: 0001 0012 00000000 ↗
/dev/input/event0: 0001 0026 00000001 → sendKeys(KeyEvent.KEYCODE_L);
/dev/input/event0: 0001 0026 00000000 ↗
/dev/input/event0: 0001 0026 00000001 → sendKeys(KeyEvent.KEYCODE_L);
/dev/input/event0: 0001 0026 00000000 ↗
/dev/input/event0: 0001 0018 00000001 → sendKeys(KeyEvent.KEYCODE_O);
/dev/input/event0: 0001 0018 00000000 ↗
```

Figure 4.5: Event Translation

4.3 Packaging

A key aspect of benchmarking is the deterministic stability and ease of use for a given set of benchmarks. Benchmarks are of little use if their results are not predictable or they are too difficult to implement. As with any open-source work, maintaining the correct dependencies, such as compilers and libraries, for a series of versions is a difficult task. To facilitate their utility, and in the spirit of transparency, we will distribute a virtual image containing all of the required source materials to begin using and extending our framework and its associated benchmarks. This will allow developers and researchers to preserve a stable and consistent environment for future comparisons and alleviate the frustration of tracking down hidden or hard to find dependencies

CHAPTER 5

CONCLUSION

The possible extensions to the framework presented are without limit. The implementation of the benchmark framework allows easy modification and customization. The proposed framework is not limited to simulation environments, functioning equally well on real hardware. The gem5 simulator represents a solid system for continuing architectural research focusing on the Android operating system. Its community is extremely active and should be assisted with the development of additional features, including the graphics hardware virtualization used in ICS and Android's recent support of the x86 architecture.

This work has presented the design and implementation of a free, open-source framework that allows for the creation of Android relevant benchmarks, useful for architectural research. The benchmark framework described is based on existing technologies within the Android system. It resolves the issues of permission and security enforcement, can manage the application lifecycle, and helps assure proper synchronization. This framework has been validated as a reasonable methodology for architectural research in comparison to current techniques.

We have demonstrated this framework in developing several micro-benchmarks consisting of common and relevant tasks. We include both Java and Native C/C++ implementations to facilitate profiling comparisons between the two. We have also established the usefulness of the Android test framework in allowing the creation of real-world application benchmarks from any open-source package. These applications can be scripted, instrumented and are deterministically stable for benchmarking requirements.

To help speed the development of benchmarks, a prototype system for recording and translating UI events into Java has been described. This system allows a researcher to develop full workflows, consisting of multiple concurrent applications, and encourages the use of these workloads in Android benchmarking.

Finally, to provide a stable environment for benchmark development and to promote the use of our framework, we provide a virtual machine packaged with all our open-source materials, including the gem5 simulator, the Android sample applications, our micro-benchmarks and our event translation scripts.

APPENDIX A

BENCHMARK PROFILES

Table A.1: Dijkstra's SSSP – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.5784602	8.682703564	0.657963	0.425147679
Cycles	1148391218	8.805713772	1313200503	0.478158108
Instructions	549454279	10.38167107	638237935	0.270081231
DTLB accesses	138929264.2	7.534111808	148179261.5	0.797933752
DTLB hit rate	99.4589843	0.044863716	99.6691503	0.00125717
ITLB accesses	67546606.4	7.915314726	70696394.5	0.689224224
ITLB hit rate	99.8655043	0.013696151	99.9263264	0.000849471
Dcache accesses	119002177	7.154674018	123718017	0.741458953
Dcache hit rate	97.5361981	0.159297201	97.7846491	0.011747425
Icache accesses	67383083.4	7.931075885	70584840	0.695928488
Icache hit rate	98.3687753	0.14813403	99.0703793	0.015714813

Table A.2: Dijkstra's SSSP – Large – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	12.3420455	7.114521031	2.0184925	14.79822175
Cycles	22372520070	7.177141332	3927191865	11.26109407
Instructions	16152809976	0.126633524	3804150360	0.181243525
DTLB accesses	8143688945	0.537506165	865230473.5	3.818814453
DTLB hit rate	99.5528806	0.05880656	99.9356911	0.003238065
ITLB accesses	3206578626	0.255184379	234929704.5	0.72176722
ITLB hit rate	99.9953463	6.57939E-05	99.6014321	0.019495646
Dcache accesses	7195717066	0.109654297	792712381.5	0.391916308
Dcache hit rate	99.4135138	0.012765176	99.466135	0.033515267
Icache accesses	3206235499	0.254947458	233908580.5	0.703111335
Icache hit rate	98.9322002	0.606182519	99.6395815	0.010473202

Table A.3: Extended Euclid – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6558345	0.808094927	0.656049	0.74132884
Cycles	1310935605	0.736584332	1311364916	0.669682169
Instructions	637294983	0.506529883	637500456	0.44378367
DTLB accesses	147684497.5	1.28386198	147784093	1.146247773
DTLB hit rate	99.6683031	0.004609208	99.6666466	0.003932308
ITLB accesses	70504345.5	1.277611048	70546955.5	1.113324778
ITLB hit rate	99.9271053	0.000762372	99.927226	0.001226064
Dcache accesses	123312469.5	1.33971946	123408351.5	1.171161752
Dcache hit rate	97.7844791	0.009469541	97.7889097	0.010198947
Icache accesses	70393802.5	1.28329784	70436563.5	1.119113317
Icache hit rate	99.070357	0.038981831	99.0701679	0.031816496

Table A.4: N^{th} Fibonacci – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6598335	0.593369425	0.656748	0.873617342
Cycles	1318347961	0.459508985	1311812818	0.70032852
Instructions	641796170.5	0.193311345	637656776.5	0.480162042
DTLB accesses	150486410.5	0.575869488	147833983	1.226799842
DTLB hit rate	99.6703077	0.003436482	99.6685154	0.003161066
ITLB accesses	71668406.5	0.516323744	70557863.5	1.203884745
ITLB hit rate	99.9281976	0.000325581	99.9269051	0.001227625
Dcache accesses	125774971	0.500610915	123457723.5	1.261386406
Dcache hit rate	97.821851	0.000894648	97.7822563	0.005046127
Icache accesses	71556115.5	0.513329484	70446176.5	1.208780586
Icache hit rate	99.0700344	0.024586552	99.0655912	0.040113772

Table A.5: Huffman Coding – Java – Profile

	Small		Large	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6761975	50.11389484	2.6125735	0.175303417
Cycles	1344534540	50.92009627	4071486804	0.05596134
Instructions	703201932.5	49.00749932	2007581849	0.041170287
DTLB accesses	167419559.5	41.73049882	877665584	0.014225835
DTLB hit rate	99.4939659	0.16566329	99.6778362	0.000547697
ITLB accesses	79561989.5	42.62264569	338766836	0.048494574
ITLB hit rate	99.873728	0.057552142	99.9784579	0.000087884
Dcache accesses	142183137.5	40.43206187	813666286.5	0.036329344
Dcache hit rate	97.6623418	0.227916033	95.8962263	0.013215011
Icache accesses	79392630.5	42.6864991	338579229.5	0.048148238
Icache hit rate	98.625642	0.276120264	99.5321751	0.002349947

Table A.6: Knuth-Morris-Pratt Search – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6582065	0.566259957	0.6575485	0.995683465
Cycles	1315118314	0.465306184	1312845800	0.792669318
Instructions	638535277.5	0.167544042	637549572.5	0.478330026
DTLB accesses	148427477.5	0.550429791	147877150.5	1.257429643
DTLB hit rate	99.6651014	0.003198579	99.6669009	0.002272969
ITLB accesses	70845622.5	0.428619023	70575211	1.198680425
ITLB hit rate	99.9268715	0.000552391	99.9269741	0.000753663
Dcache accesses	123955544.5	0.454034661	123433577	1.242346544
Dcache hit rate	97.7799533	0.003794935	97.7797043	0.009591679
Icache accesses	70733268	0.434339242	70463502	1.19461389
Icache hit rate	99.0480944	0.016465707	99.0586167	0.043137881

Table A.7: Knuth-Morris-Pratt Search – Large – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.908344	1.354826632	0.655668	0.719759186
Cycles	1813168798	1.316243388	1310178399	0.602350817
Instructions	955119133.5	3.087636103	636663592	0.282645865
DTLB accesses	208905111	0.334535576	147396962	0.827947959
DTLB hit rate	99.6563252	0.042420643	99.6678073	0.002721338
ITLB accesses	101545288.5	1.02133588	70318494.5	0.704317365
ITLB hit rate	99.9268325	0.015362996	99.9272457	0.000299457
Dcache accesses	175048421	0.541035889	122991168	0.754540565
Dcache hit rate	97.7444791	0.20097628	97.7786488	0.005570797
Icache accesses	101382031	1.041183763	70209689.5	0.704975788
Icache hit rate	99.0247338	0.202942534	99.0835352	0.03605547

Table A.8: Longest Common Subsequence – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.476922	14.82854169	0.6563935	0.681798162
Cycles	948654127.5	15.24141534	1311628186	0.564586539
Instructions	538594124	11.47929935	637587956	0.256834773
DTLB accesses	133916222.5	10.1446279	147896948.5	0.778505249
DTLB hit rate	99.49153	0.083762053	99.669437	0.001812006
ITLB accesses	63156642.5	10.22265303	70495125	0.664868774
ITLB hit rate	99.9001253	0.01260468	99.9276498	0.000154149
Dcache accesses	116353113.5	10.4292761	123459632.5	0.693024555
Dcache hit rate	97.4233971	0.249788836	97.7779703	0.006273339
Icache accesses	63028541	10.20971396	70383917	0.660799621
Icache hit rate	98.6390381	0.184406843	99.0789024	0.02514323

Table A.9: Longest Common Substring – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.9482505	0.144441351	0.6574375	0.66673333
Cycles	1877816667	0.266387482	1313715935	0.549690942
Instructions	992279143	0.049342596	638295735.5	0.242082609
DTLB accesses	215860221	0.241240241	148258626	0.731650062
DTLB hit rate	99.7414815	0.000950517	99.6676218	0.002984204
ITLB accesses	103815255.5	0.15299801	70728162	0.627116939
ITLB hit rate	99.9477689	0.00014357	99.9271274	0.000414309
Dcache accesses	180312084.5	0.154049928	123805889.5	0.652350366
Dcache hit rate	98.1429555	0.000193806	97.7803337	0.004453662
Icache accesses	103683724	0.155251012	70619282.5	0.625978095
Icache hit rate	99.2863327	0.010238293	99.060754	0.029788049

Table A.10: QSort – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	6.3512442	4.57468278	0.940673	0.642706124
Cycles	3891309920	23.50760466	1877530865	0.362977045
Instructions	1767898362	9.989519435	999761162.5	0.047127266
DTLB accesses	517127580.8	29.12093565	220021178	0.266849268
DTLB hit rate	99.1163199	0.159749804	99.7686775	0.000011687
ITLB accesses	231765537	24.3793593	104584643.5	0.10493829
ITLB hit rate	99.8817278	0.076938377	99.9462867	0.003072291
Dcache accesses	442703800.8	28.12114856	183925454.5	0.112253717
Dcache hit rate	97.7981288	0.478765042	98.1892116	0.006635011
Icache accesses	231259326.8	24.25199754	104417115.5	0.102640312
Icache hit rate	98.7464845	0.588999373	99.3596657	0.017590662

Table A.11: QSort – Large – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	31.39360333	0.938912642	1.456347	6.060150897
Cycles	57029106431	2.35386841	2880614242	4.56087471
Instructions	17667356818	1.107725751	1999687933	0.105442185
DTLB accesses	10359172094	1.241076207	472262729	2.053004834
DTLB hit rate	98.1007257	0.034091548	99.8762598	0.000698111
ITLB accesses	4328210072	1.302601593	300161540	0.214450893
ITLB hit rate	99.6626967	0.026983877	99.9779221	0.000038176
Dcache accesses	9493415637	1.206533092	417759856	0.196322741
Dcache hit rate	98.4988981	0.067256437	99.0767447	0.018963001
Icache accesses	4307488884	1.336550314	299709529	0.21410758
Icache hit rate	94.7972506	0.122382562	99.6971547	0.010562441

Table A.12: Subset Sum – Small – Profile

	Java		Native	
	Mean	**Co. of Var.**	**Mean**	**Co. of Var.**
Runtime (sec)	0.6579965	0.866694001	0.7751145	20.78504739
Cycles	1314166678	0.709090692	1526223416	19.04387542
Instructions	638548421.5	0.473801694	734291591.5	18.00950402
DTLB accesses	148393747.5	1.217742074	174426867.5	19.98293584
DTLB hit rate	99.6684188	0.003193297	99.6379986	0.042821635
ITLB accesses	70856332.5	1.197570168	81728691.5	17.78393375
ITLB hit rate	99.926689	0.001064349	99.9127272	0.018959045
Dcache accesses	123954264.5	1.247898705	142818703.5	17.47440747
Dcache hit rate	97.7849383	0.009245604	97.7625718	0.014500901
Icache accesses	70743488	1.205336722	81583367.5	17.76954899
Icache hit rate	99.0477347	0.035601329	99.0634296	0.035848446

Table A.13: GLSurfaceView – Detailed – Profile

	Mean	**Co. of Var.**
Runtime (sec)	56.7071455	0.495669544
Cycles	113413651249	0.49558621
Instructions	9761405726	4.984808126
DTLB accesses	4027482050	5.041953521
DTLB hit rate	99.89654881	0.022040119
ITLB accesses	9943508533	4.998294654
ITLB hit rate	99.99229389	0.001651655
Dcache accesses	3908616500	5.018261443
Dcache hit rate	99.55096760	0.061568786
Icache accesses	9943507327	4.998287526
Icache hit rate	99.68972167	0.007476754

BIBLIOGRAPHY

[1] Android. Activity Lifecycle.
http://developer.android.com/reference/android/app/Activity.html.

[2] Android. Platform Versions.
http://developer.android.com/about/dashboards/index.html.

[3] Android. Testing Fundamentals.
http://developer.android.com/tools/testing/testing_android.html.

[4] AnTuTu Labs. http://www.antutulabs.com/antutu-benchmark.

[5] ARM. ARMv7A reference manual. http://www.arm.com/index.php.

[6] Aurora Softworks. Quadrant.
http://www.aurorasoftworks.com/products/quadrant.

[7] S. J. Barbeau. GPSBenchmark. http://www.gpsbenchmark.com/.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX track*, pages 41 – 46, April 2005.

[9] F. Bellard. QEMU Accelerator (KQEMU). http://bellard.org/qemu/kqemu-tech.html, 2009.

[10] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[12] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, July 2006.

[13] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[14] BYTE Magazine. BYTEmark benchmark program.
http://www.tux.org/ mayer/linux/byte/bdoc.pdf.

[15] Canalys. Majority of smart phones now have touch screens. http://www.canalys.com/newsroom/, February 2010.

[16] Canalys. Google's Android becomes the world's leading smart phone platform. http://www.canalys.com/newsroom/, January 2011.

[17] Canalys. Smartphones overtake client PCs in 2011. http://www.canalys.com/newsroom/, February 2012.

[18] H. J. Curnow, B. A. Wichmann, and T. Si. A Synthetic Benchmark. *The Computer Journal*, 19:43–49, 1976.

[19] J. Dongarra. The LINPACK Benchmark: An Explanation. In E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, editors, *Supercomputing, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings*, volume 297 of *Lecture Notes in Computer Science*, pages 456–474. Springer, 1987.

[20] EEMBC. http://www.eembc.org/.

[21] J. Elder and M. D. Hill. Trace-Driven Uniprocessor Cache Simulator. http://pages.cs.wisc.edu/ markhill/DineroIV/.

[22] Futuremark Corporation. Audit Report: Alleged NVIDIA Driver Cheating on 3DMark03. http://www.futuremark.com/pressroom/.

[23] Gartner. Gartner Says Worldwide Smartphone Sales Reached Its Lowest Growth Rate With 3.7 Per Cent Increase in Fourth Quarter of 2008 [Press Release]. http://www.gartner.com/it/page.jsp?id=910112, March 2009.

[24] Gartner. Gartner Says Worldwide Mobile Phone Sales to End Users Grew 8 Per Cent in Fourth Quarter 2009; Market Remained Flat in 2009 [Press Release]. http://www.gartner.com/it/page.jsp?id=1306513, February 2010.

[25] Gartner. Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012 [Press Release]. http://www.gartner.com/it/page.jsp?id=1622614, April 2011.

[26] Gartner. Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010 [Press Release]. http://www.gartner.com/it/page.jsp?id=1543014, February 2011.

[27] Google. V8 Benchmark Suite. http://v8.googlecode.com/svn/data/benchmarks/v7/run.html.

[28] C. Guillon. Program Instrumentation with QEMU. In *1st International QEMU Users' Forum*, page 15, 2011.

[29] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite.

In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[30] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. *IEEE Workload Characterization Symposium*, 0:81–90, 2011.

[31] V. Hirvisalo and J. Knuuttila. Profiling for QEMU. Technical report, Aalto University, 2010.

[32] IDC. Android and iOS Powered Smartphones Expand Their Share of the Market in the First Quarter, According to IDC [Press Release]. http://www.idc.com/getdoc.jsp?containerId=prUS23503312, May 2012.

[33] ITU World Telecommunication/ITC Indicators database. Mobile telephony statistics, December 2011.

[34] Kishonti Informatics. GLBenchmark. http://www.glbenchmark.com/.

[35] Kishonti Informatics. JBenchmark. http://www.jbenchmark.com/.

[36] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[37] C. Lee, E. Kim, and H. Kim. The AM-Bench: An Android Multimedia Benchmark Suite. Technical report, GIT-CERCS-12-04, Georgia Institute of Technology, 2012.

[38] S. Lee and J. Jeon. Evaluating performance of Android platform using native C for embedded systems. In *Control Automation and Systems (ICCAS), 2010 International Conference on*, pages 1160–1163. IEEE, 2010.

[39] Y.-H. Lee, P. Chandrian, and B. Li. Efficient Java Native Interface for Android based mobile devices. In *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '11, pages 1202–1209, Washington, DC, USA, 2011. IEEE Computer Society.

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

[41] A. Miettinen, V. Hirvisalo, and J. Knuuttila. Using QEMU in timing estimation for mobile software development. In *1st International QEMU Users' Forum*, page 19, 2011.

[42] NVidia. Benefits of Multicore CPUs in Mobile Devices. http://www.nvidia.com/.

[43] NVidia. Tegra 2. http://www.nvidia.com/object/tegra.html.

[44] Passmark Software. http://www.passmark.com/.

[45] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, 29(5):18–29, Sept. 2009.

[46] SPEC. CPU2006 benchmark suite. http://www.spec.org/cpu2006/.

[47] Texas Instruments. OMAP Mobile Processors. http://www.ti.com/omap.

[48] J. Tromp. Fhourstones. http://homepages.cwi.nl/ tromp/c4/fhour.html.

[49] V. M. Weaver. *Using Dynamic Binary Instrumentation to Create Faster, Validated, Multi-Core Simulations*. PhD thesis, Cornell University, May 2010.

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

[51] V. Zivojnovic, C. S. J. Martinez, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proc. of ICSPAT'94 - Dallas*, oct 1994.

BIOGRAPHICAL SKETCH

Ira Ray Jenkins was born in Fort Sam Houston, Texas, in 1987. He was raised in Bonifay, Florida, and graduated as valedictorian of Holmes County High School in 2006. In 2010, he received a Bachelor of Science degree in Computer Science, summa cum laude, from The Florida State University. He will matriculate at Dartmouth College in September 2012 to pursue a doctorate in Computer Science.