Using OpenMP under Android

Vikas[1], Travis Scott[2], Nasser Giacaman[3], and Oliver Sinnen[4]

The University of Auckland, New Zealand
{vik609,tsco033}@aucklanduni.ac.nz,
{n.giacaman,o.sinnen}@auckland.ac.nz

Abstract. The majority of software authored for the mobile platforms are GUI-based applications. With the advent of multi-core processors for the mobile platforms, these interactive applications need to employ sophisticated programming constructs for parallelism-concurrency, in order to leverage the potential of these platforms. An OpenMP-like, easy to use programming construct, can be an ideal way to add productivity. However, such as environment needs to be adapted to object-oriented needs and should be designed with an awareness of the interactive applications. Also, OpenMP does not provide a binding that target these platforms. This paper presents a compiler-runtime system for Android that presents OpenMP-like directives and GUI-aware enhancements.

Keywords: OpenMP, Android, GUI applications.

1 Introduction

For several years now (since 2006), the number of cores available in mainstream desktops started increasing; only half a decade later (in 2011), this trend started emerging in the mobile space, including smartphones and tablets. All flagship mobile devices are now multi-core, most of which are running Android. Much like the dilemma that faced software developers when mainstream desktops went multi-core [15], the same dilemma now faces mobile application developers; in order to harness the computing power provided by these additional cores, mobile "apps" must be parallelised since traditional sequential code is unable to utilise the increased resources provided by multiple cores.

Java, the language used by Android [3], provides a suite of native constructs for writing concurrent code, in the form of threads, thread pools, handlers, runnables and so forth. However, successfully using these tools can be difficult, and may not be immediately obvious to developers with a sequential programming background. This is especially important as the appeal of application development is being embraced by non-experienced programmers, whereas parallel programming was traditionally the domain of experienced programmers targeting large scientific and engineering problems. But even for experienced users, managing a large number of threading constructs can be time consuming and leads to the introduction of additional bugs [9].

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 15–29, 2013.

1.1 Motivation

Keeping the above in mind, it can be seen that the rapid advancement of multi-core processors for mobile devices necessitate the GUI applications (existing and new ones) to be parallel and concurrent. The mismatch between the pace of hardware advancement and software development can be bridged by using the principles and methodologies of directive-based constructs. The directive-based approach can decrease the general complexities of parallel-concurrent programming. Furthermore, with the ever increasing importance of the mobile domain, and in the absence of any directive-based implementation or binding, an OpenMP-like environment can be helpful to a great extent.

1.2 Contributions

This paper acknowledges OpenMP's expressive power and ease of use in the domain of shared memory parallel programming. However, should we wish to extend OpenMP's coverage to encompass mobile application development, there are concepts vital to the development of object-oriented GUI-based applications that are absent in the OpenMP model. To address these shortcomings, this paper makes the following contributions:

Study of GUI application: We explore the basic nature of GUI-based applications, particularly on mobile platforms. This leads to formalising the general requirements for such applications.

OpenMP-like environment for Android: The most dominant mobile platforms (Android and iOS) use object-oriented languages for the development of interactive applications. This paper presents OpenMP-like[1] directives and runtime support for Android.

GUI aware enhancements: Assisted with the exploration of the nature of GUI-based applications, this paper proposes GUI-aware extensions.

Implementation exploration for Android: Android supports Java as an application development language (and C/C++ through NDK), but it omits some standard Java libraries used in standard GUI application development (such as Swing and AWT). In this light, the paper presents implementation details of the proposed compiler, demonstrating a sample Android-based OpenMP compiler implementation.

Performance evaluation on mobile domain: We evaluate the proposed system using a set of interactive GUI applications for Android.

2 Background

2.1 Distinct Structure of GUI-Based Applications

In many ways, the execution flow in a GUI application is different from that of a conventional batch-type application. Firstly, the interactive applications have

[1] The directives, specially the GUI directives, that we present here are not official directives in the OpenMP standard; they are however designed with the intention of promoting the spirit of OpenMP, and as such, we refer to them as OpenMP-like directives.

their execution flow determined by the various inputs from the user. Secondly, the control flow is largely guided by the framework code of which the program is built on; this distinguishing aspect is known as *inversion of control* [7], when the flow of control is dictated by a framework rather than the application code.

The primary concept is that *events* are generated from within the framework code in response to user actions. The programmer only needs to implement specific routines, known as *event handlers*, in the application code in response to those events. The control returns back to the framework upon completion of the respective event handler, namely to the *event loop*. In most GUI frameworks, this is performed by a dedicated thread known as the GUI thread. In comparison, batch-type programs have the control flow determined by the programmer (i.e. the application code) and deal with serial input-output (even though the processing may be parallelised).

From the requirements perspective, in addition to the generic software requirements, an important criterion for GUI applications is their responsiveness. The application should always remain interactive and responsive to user actions. From the execution semantics, programmers must ensure that the GUI thread minimises its execution in the application code, therefore remaining largely in the event loop to respond to other potential events. In addition to off-loading the main computation away from the GUI thread, regular intermittent progress updates frequently need to be communicated back to the user to be perceived as having a positive user experience.

2.2 Mobile Devices and GUI Application Development

The mobile environment differs from the desktop environment on both the hardware and the software levels. From the hardware aspect, the mobile environment is largely constrained in memory and processing power (the dominant ARM processors focus on efficiency of power consumption). On the software side, the GUI frameworks support high level languages (such as Java in Android, Objective-C in iOS); however, not all the libraries or APIs are supported in a mobile application. Some possible reasons might be that these frameworks cater to the specific needs of the mobile devices and do not need to provide support for general purpose libraries. Also, many libraries and tools have been part of the desktop environment for legacy purposes. In many cases, they were designed and developed for systems with larger memory and faster processing power, and thus do not find a place on mobile platform. Nevertheless, mobile-based GUI applications do follow a similar architecture to desktop-based GUI applications, but the general consideration for memory and processing efficiency is always present.

2.3 Distinctions in Android Application Development Environment

There are some mentionable factors that make Android GUI application development different from that of the desktop environment. First, Android's execution environment runs every application in a separate sandboxed process and only a single application is displayed on the screen at a time (under the hood, Android uses a Linux kernel and every application is treated as an individual user).

This restricts the applications to have shared access to the file system, the application data, and more. The applications are executed in separate instances of the virtual machine, thus each application has a separate instance to itself. This enforces a design requirement over the applications. Moreover, Android's virtual machine, known as the Dalvik Virtual Machine (DVM), is not like a regular Java Virtual Machine (JVM). It is a slimmed down virtual machine that is designed to run on low memory. Importantly, DVM uses its own byte code and runs .dex (Dalvik Executable File) instead of the .class files that the standard JVM runs. In effect, the legacy Java libraries or Java code need a recompilation in order to be used for the Android.

On the framework side, Java is only supported as a language; libraries like Swing, AWT and SWT are not supported [13]. Nevertheless, it provides extra concurrency features, as will be discussed in section 3. Thus, even an experienced Java programmer needs to become acquainted with alternative constructs and the legacy programs cannot be directly ported.

Furthermore, owing to its recent development, Android has incorporated the generic requirements of a GUI-based application into the framework. For example, to counter the responsiveness related application freeze, Android throws an Application Not Responding (ANR) [2] error when an application remains unresponsive for more than a certain amount of time (around 5 seconds for Jelly Bean). In effect, this enforces a rule on the GUI applications to avoid the time consuming computations on the GUI thread and the applications should offload them. Another example is that of the `CalledFromWrongThreadException` exception; like most of the GUI framework, the Android framework is single threaded and the thread safety is maintained by throwing this exception if any non-GUI thread tries to update the GUI. Therefore, the programmer always needs to use specific constructs in concurrent programs for updating the GUI.

3 Related Work

3.1 Android Concurrency

Android supports the prominent concurrency libraries of Java using the `java.util.- concurrent` package, thus supporting the `ExecutorService` framework. The native threading is also supported. Additionally, Android exposes `AsyncTask` and `Handler` [12] for advanced concurrency. `AsyncTask` enables to perform asynchronous processing on background worker threads and enables methods to update the results on the GUI. `Handler` enables the posting and processing of the `Runnables` to the thread's message queue.

3.2 OpenMP for Android

There is no official OpenMP specification for Java, so OpenMP is not supported on Android. The Android Native Development Kit (NDK) supports C/C++ [11], but it does not support an OpenMP distribution. Although there are no Android OpenMP implementations, there are however some respectable Java OpenMP

implementations, namely JOMP [6] and JaMP [8]. While these tools provide important contributions to OpenMP Java bindings, they do not specifically target GUI applications as is the focus of Pyjama. More specifically, these solutions were developed well before the time of Android application development, as is the focus of this research.

4 Android Pyjama Compiler-Runtime

Pyjama [16] is a compiler-runtime system that supports OpenMP-like directives for an object-oriented environment, specifically for Java. Where required, Pyjama adapts the principles and semantics of OpenMP to an object-oriented domain and, in addition, provides the GUI-aware enhancements. The research presented in this paper is based on the preliminary work done on Pyjama, and now extending it to mobile application development.

4.1 Standard Directive Syntax

In the absence of any Android specification for OpenMP, we propose a format that is close to the OpenMP specification. A program line beginning with `//#omp` is treated as a directive by the proposed compiler and ignored as inline comments by the other compilers. Generic syntax is as shown below:

```
//#omp directiveName[clause[[,]clause]..]
```

4.2 Conventional OpenMP Directives and Semantics

The conventional directives [16], are supported in Android Pyjama as well. The system also supports object-oriented semantics within the scope of these conventional directives. For instance, *for-each* loop construct is a way to traverse over a collection of objects in object-oriented programming and parallelising a *for-each* loop is permissible using the `parallel for` construct. Furthermore, the OpenMP synchronisation directives like `barrier`, `critical`, `atomic` and `ordered` are supported with identical semantics as that of OpenMP for C/C++.

4.3 GUI-Aware Extensions

To improve upon the usability of OpenMP for GUI applications, Pyjama on Android introduces the following GUI-aware constructs:

- `freeguithread` directive

Specifies a structured block that is executed asynchronously from the GUI thread, freeing the GUI thread to process events.

```
//#omp freeguithread
structured-block
```

- `gui` directive

Specifies a structured block that is executed by the GUI thread. An implicit barrier is placed after the structured block unless an optional `nowait` clause is used.

```
//#omp gui [nowait]
structured-block
```

`freeguithread` Construct. A prominent limitation of using OpenMP in a GUI application is that OpenMP's fork-join threading model effectively violates the responsiveness of a GUI application. Consider the scenario where the code inside an event handler encounters an OpenMP construct; the master thread (MT) would be the GUI thread and would therefore take part in the processing of the OpenMP region. But in a GUI application, this is a problem; the GUI thread will remain busy processing (the parallel region), effectively blocking the GUI. The GUI-aware thread model addresses this responsiveness related issue; the basic principle is that an application will not have the tendency to become unresponsive, or block, if the GUI thread is free to process user inputs.

To achieve this responsiveness, we need to relieve the GUI thread from the execution of a specified region; hence the `freeguithread` directive. The underlying mechanism determines if the thread encountering `freeguithread` is the GUI thread. If yes, a new thread is created, called the Substitute Thread (ST), which executes this region on behalf of the GUI thread. As a result, the GUI thread is free to return to the event loop and handle incoming events. The structured block of the `freeguithread` directive is executed asynchronously to the GUI thread. When the execution of the region is completed by the ST, the GUI thread is notified and returns to execute the region following the `freeguithread` directive. This approach keeps the OpenMP threading model with its fork-join structure intact. Any `parallel` region that is then encountered by the ST is handled in the usual manner, whereby the ST is the master thread of that region.

`gui` Construct. For GUI applications, there is a need to update the GUI with intermittent results when the application code is still busy in the background processing. This may be related to conveying the partial results of the background processing or it may be a GUI update to convey the completion of background processing. In GUI application development, this is achieved by implementing a way to provide periodic updates to the GUI. Generally, it involves careful synchronisation methods or shared global flags. A programmer needs to introduce major code restructuring to spawn the computational work to other thread(s) and then again to execute GUI code, commonly encapsulated within `Runnable` instances and posted to the GUI thread. These methods have their own limitations and complexities and make it difficult to involve any OpenMP-like programming. It also opposes the the OpenMP philosophy of maintaining the program's original sequential structure when the OpenMP directives are ignored. For an elegant and easy solution of these issues, Pyjama introduces the new `gui` directive. Using it, a program can execute part of the code on the GUI thread from a background-processing region. This eliminates the need to maintain

complexities of synchronisation. The addition of the `freeguithread` and `gui` directives enable programmers to achieve responsive application development by obeying the single-thread rule of most GUI toolkits.

Syntactically, `freeguithread` and `parallel` can be combined in one directive statement. The further combination with the worksharing directives `for` and `sections` is also possible.

4.4 Runtime

The runtime component provides execution environment and timing routines, conforming to OpenMP 2.5 [10]. Additionally, the runtime provides a set of utility methods for the benefit of a programmer.

5 Implementation

5.1 Construction of Compiler

The parser for the compiler was created using Java Compiler Compiler (JavaCC). JavaCC is an open source parser generator for Java and the Java 1.5 grammar is provided as a part of the JavaCC distribution. It should be noted that JavaCC is not a lexical analyser or parser by itself. It needs to be provided with regular expressions and grammar and it generates a lexical analyser and a parser. The Java 1.5 grammar file was used as the base and was augmented with extended Backus Naur form (EBNF) -like[2] grammar notations for the OpenMP-like directives, to generate the lexical analyser and the parser.

5.2 Code Generation

The elementary process involves lexical analysis and parsing of the input code to generate an intermediate representation of the code (in this case, it is an AST); the AST is then traversed, using the visitor design pattern [14], and the directive specific nodes are translated to the respective parallel or concurrent version of the code.

From the software engineering perspective, the implementation is modularised by dividing the generation into two broad passes of *normalisation* and *translation*. Normalisation operations help simplify the process of target code generation. A wider range of directives are supported by actually implementing only the basic directives and normalising other directives to those implemented ones. The next pass is the translation pass, where an explicit parallel and concurrent version of the code is generated.

Translation of Conventional Directives and Clauses. The semantics of multi-threaded and GUI-aware translation adheres to the OpenMP's threading model. While introducing parallelism through the directives, the execution semantics follows the *fork-join* threading model and identifies a distinct master thread, like the OpenMP model.

[2] JavaCC provides the notation for defining grammar in a manner similar to EBNF.

Implementation wise, code outlining is the elementary approach employed by the translation pass. As figure 1 illustrates, the structured block from the `onClick()` method is outlined to form a new method (steps 1 and 2) and then executed using the runtime native queue and the native task-pool (steps 3, 4 and 5). Reflection is used to retrieve and execute the respective user code, thus implementing a *fork* and a barrier is placed after the region, thus implementing the *join*.

Also, the compiler achieves data passing by creating a new class to hold the variables from the encountering thread. Here, based on the data clauses used, an object of this class is instantiated and values from the encountering thread are assigned to it.

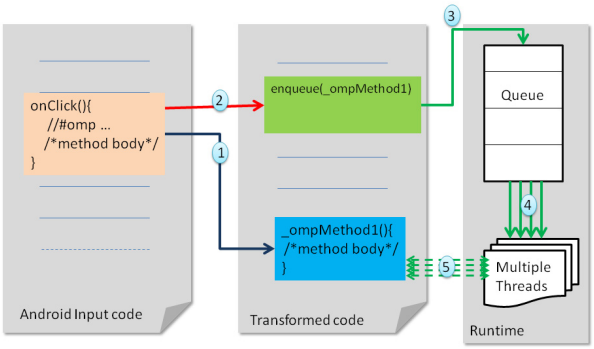

Fig. 1. The code outlining and queuing approach

Translation of GUI Directives. Translation of GUI directives forms an interesting part of this research. Semantically, the `freeguithread` region is treated as a task and moved to a new method which is enqueued and executed in the same way as a conventional directive. Like the semantics for the conventional directive, the enqueuing serves as the *fork*. The exception being that the encountering thread is not assigned as the `master`. Also, no barrier is placed in the original code, but a callback method is created using the code that appears after the `freeguithread` region in the original code. This callback is processed only after the execution of the task. This point serves as a continuation point and effectively as the *join*.

Concerning the specifics of implementation, the Android framework does not have an Event Dispatch Thread (EDT) and that is why the rudimentary implementation uses the application's *looper* instead. Every Android application has a main thread living inside the process in which the application is running. This thread contains a *looper*, which is called the main looper (instance of class `Looper`). For an *activity* or a *service* with a GUI, the main looper handles the GUI events. The main looper in Android is essentially analogous to the EDT in Java. With that knowledge, the GUI-aware directive verifies if the encountering thread is an event loop or not, using the method shown below:

```
private boolean isEventLoop() {
    boolean bELoop = false;
    if(Looper.myLooper()!= null) {
        bELoop = (Looper.myLooper() == Looper.getMainLooper())
    }
    return bELoop;
}
```

The `gui` directive translation searches for the main looper and posts the user code (as a `Runnable`) to it, as illustrated in the following code:

```
pHandler handler = new Handler(Looper.getMainLooper());
    handler.post(new Runnable(() {
        public void run() {
        ....
        }
    });
```

Here, `Handler` is another class that allows sending and processing the runnable objects associated with a thread's message queue. It binds to the threads message queue that created it. Also, as shown in the code, `Looper.myLooper()` returns the looper associated with the current thread, if it has one. If the returned one is not the application's main looper, then it is concluded that the current thread is not the main thread of the application.

6 Evaluations

In this section we present the preliminary evaluations of the conventional and GUI directives for Android. The overall strategy has been to evaluate the system on diverse devices and using diverse applications (non-conventional application, conventional mainstream applications and pure performance measuring applications).

6.1 Evolution Strategy Algorithm

The first application we present is an implementation of the EvoLisa algorithm created by Roger Alsing [1]. It is an evolution strategy algorithm, and is a subset of the broader class of evolutionary algorithms [5]. For this evaluation, we used a Galaxy Nexus 7 tablet, running an ARM Cortex-A9 Nvidia Tegra 1.2 GHz quad core processor with 1GB of RAM.

Strategy. In order to accurately gauge the suitability of the system as a development tool, and also more accurately reflect a real design scenario, an algorithm was chosen that was not already parallelised. Without prior knowledge of the algorithm model it was ensured that selection bias would not result in an algorithm being selected that was naturally suited to a fork-join model. The challenge of parallelising the algorithm also provided the opportunity to explore

how the environment can be used to parallelise non trivial algorithms. In addition to being unknown, the algorithm was also required to be long running and computationally expensive, such that it would be completely impossible to implement purely on the GUI thread. It also needed to provide updates to the user interface and have some level of user interactivity, so that it cannot simply be passed to a background thread to execute in isolation from the rest of the application. In fact, below is the code snippet of this app using Android Pyjama:

```
performImageGeneration(){
initialise starting working set of polygons
....
    //#omp freeguithread parallel
    {
        while(continueToGenerate){
        // fetch portion of polygons for current thread
        ....
        // attempt 100 random mutations and select best one
        ....
            //#omp barrier
            ....
            //#omp single
            {
                //recombine polygons and create new parent
                // create display image
                //#omp gui
                {
                    // update GUI
                }
            }
        }
    }
}
```

This code snippet promotes the harmony of using standard OpenMP for performance and synchronisation (in green), in combination with the GUI-aware constructs of Pyjama (in blue) to adhere to GUI concurrency rules in promoting a responsive Android GUI application. Furthermore, this code demonstrates the elegance of using a directive-based solution (such as OpenMP) to a new class of user interactive applications, not just the batch-type scientific and engineering applications traditionally tackled by OpenMP.

Evaluations. The algorithm was run repeatedly using different numbers of threads on three versions of the application: sequential, multi-threaded using `ExecutorService` and Android Pyjama directives. Tests were all performed on a single boot, with tests interleaved (i.e. single thread test performed, multi-threaded test with 2 threads performed, multi-threaded test with 4 threads performed). To perform the test an image was left to generate for 900 seconds, with

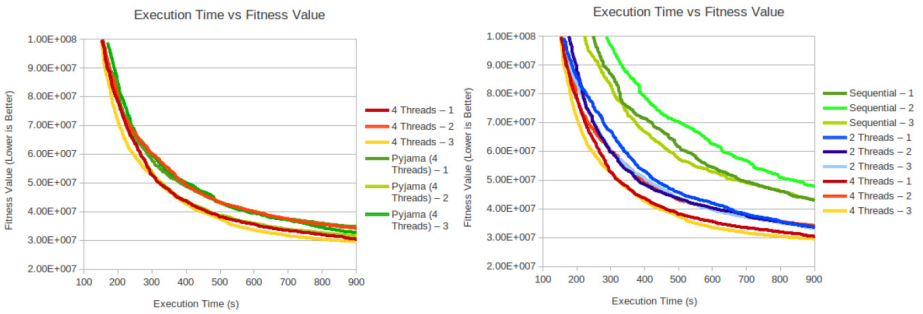

Fig. 2. Execution time vs fitness Values. Left:Pyjama vs native treads. Right:Improvement in fitness value with increase in number of threads.

the execution time and fitness level of the generated image recorded after each evolution cycle.

Results of the testing are shown in figure 2, displaying three runs of each of the manual threading (using `ExecutorService`) and Android Pyjama when each use four threads. It can be observed that the scalability and performance gain achieved is similar in both versions; the directive-base parallel-concurrent version (i.e. Pyjama) performs similar to the manually programmed multi-threaded version (lower fitness value is better). Due to the randomness aspect of the algorithm, there is sometimes a large degree of variation between runs. Figure 2 helps confirm though, that having increased parallelism does in fact speed up the progress of the algorithm on the mobile device, as increasing from 1 to 2 to 4 threads.

From the GUI aspect, intermittent GUI updates (i.e. results from the mutations) is an important part of this application. The code snippet showed how this was effectively handled by the GUI directive, embedded within a standard OpenMP `single` construct to ensure only one update request is made per cycle. In effect, the Android Pyjama implementation exhibited identical responsiveness to the manually threaded implementation using standard Android constructs; however, the Android Pyjama implementation clearly has the advantage of highly resembling the sequential version with minimal recoding. Irrespective of the ongoing background processing, the GUI remained responsive to the user actions. Figure 3, presents a small gallery of screenshots from the application.

6.2 Pattern Rendering Application

We developed another GUI application that creates psychedelic renderings on the screen. We used the Samsung GT-I9300 (Samsung Galaxy S-III) smartphone that has a 1.4 GHz quad-core ARM Cortex-A9 CPU and 1 GB of RAM.

Strategy. We designed the Android application in a way that it can serve as a representation of the major types of applications that are published for mobile

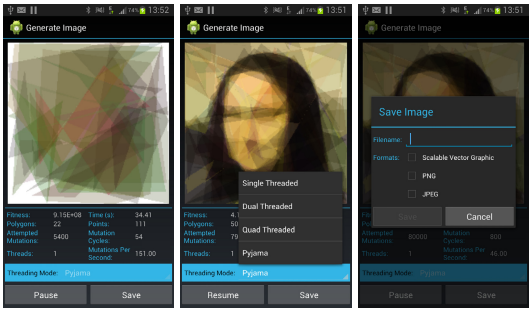

Fig. 3. Responsiveness while reconstruction of images in the EvoLisa Application

devices. Firstly, it creates a standard GUI for information screens and the help-screen using standard components of the Android toolkit, such as *layouts* and *widgets*. Secondly, the rendering screen of the application draws directly to the `canvas`, and controls all the drawings to it directly. In this way, it represents typical gaming and graphical applications, which are more compute intensive and richer in rendering.

The directives add incremental concurrency, performance and responsiveness to the application. Each screen (*activity*) uses the GUI-aware directive (`gui`), to render the screen components (*layouts* and *widgets*). The `parallel for` directive, along with GUI-aware directives (`freeguithread/gui`) are used to migrate the computations to the thread pool and to keep the screen responsive to gestures and touch inputs. For evaluations, we used two versions of the same applications. The first one using the directives and the second one using standard Android constructs, and compared the behaviours. This reflects on the completeness of the system on Android.

Evaluations. Considering the application responsiveness and behaviour, the two versions of the applications performed identically. The GUI display and pattern rendering on the screen is seen to be the same. Looking at the responsiveness, as figure 4 quantifies, the gesture and touch inputs were correctly registered.

6.3 Responsiveness Evaluation with Monkey Tool

The Monkey tool [4], distributed with Android SDK, generates random events (such as touch, motion, key events, clicks and more) which can be fired at the application. In figure 2, we observed that an increase in the number of threads improves the performance; we used the Monkey tool to test the responsiveness in the face of this improved performance. We evaluated different versions of the application by averaging over 20 runs and in each run we fired 5000 UI events at the applications. We measured the number of events that get dropped while the GUI thread is busy processing. We also measured the response time. The lesser the response time, the more responsive the application, and lesser are the chances of an application-freeze. We compared the Android Pyjama

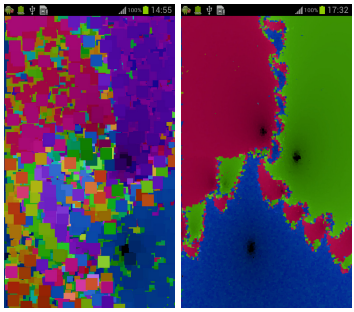

Fig. 4. Snapshots showing response to user's gestures. A "circle" gesture adds more colours to the fractal generation.

version of the application with that of another version developed using the native threads. The non-concurrent (single thread) version remains unresponsive to the events while processing the computationally intensive load and so could not be evaluated (Android throws an exception if workload is processed on the GUI thread). The results are shown in figure 5.

The results provide a fairly quantifiable measurement of the responsiveness in the Android Pyjama version. An overall responsiveness is achieved with both the Android concurrency and Pyjama, with comparable results. Here, it should be noted that Pyjama utilises a native task pool and therefore avoids the thread creation and destruction overheads. Also, as the number of threads increases and the application achieves more computation, the GUI thread gets smaller time slice to process the UI events. But this results in very small loss of responsiveness when compared to the performance gain that the application achieves.

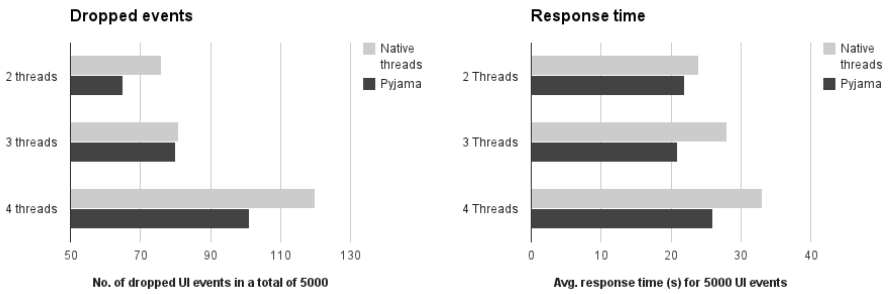

Fig. 5. Results of the responsiveness test with the Monkey tool

6.4 Mandelbrot Application

In order to measure the gross performance gain, by minimising the effects of GUI-based processing, we implemented an Android Mandelbrot application.

Two versions of the application were created for this evaluation; a serial version which does not use any concurrency or parallelism constructs (although a background thread is necessary to be launched for time consuming computations, otherwise the Android operating system throws an exception) and the second one uses the directives for parallelism (the `parallel` construct and the `parallel for` construct). It was observed that Android Pyjama scaled well on the Galaxy S III; for 4 cores, a speedup of 2.7x was observed in comparison to the serial version.

6.5 Productivity Evaluation

Even though in its preliminary stages, the proposed system exhibits good reasons for bringing a directives-based approach to the Android platform. By replacing native threading constructs with compiler directives, we were able to remove much of the fragmentation introduced into the code by these constructs (a 75% reduction was achieved in the EvoLisa demonstration application), resulting in more sequential, readable code. For the GUI applications, the aim is to provide a fluid experience for the user, and achieving this aim is often hindered by a large number of small tasks which introduce slight delays. The proposed system makes it simple for developers to offload these tasks to background threads, and provides the necessary tools to manage the complex interactions between these background threads and the GUI thread, thus removing the burden of manually managing this process.

7 Conclusion

With the mobile domain being so relevant today, we presented a compiler-runtime system to support directives based parallelism for Android promoting the OpenMP philosophy. The evaluations demonstrated positive results using a set of Android applications that focused on the GUI aspect of these applications; here, traditional parallelism in the form of speedup is only one aspect of performance, the other vital measure of performance being that of ensuring a user-perceived positive experience. Code snippets for the used directives also helped illustrate the contribution such a tool can provide for the productivity of mobile application developers.

References

1. Alsing, R.: Genetic Programming: Evolution of Mona Lisa (December 2008)
2. Google Inc. Android. Keep your app responsive (April 2013)
3. Android, Google Inc., `http://developer.android.com/guide/basics/what-is-android.html`
4. Android, Google Inc., `http://developer.android.com/tools/help/monkey.html`
5. Brownlee, J.: Evolution Strategies

6. Bull, J.M., Kambites, M.E.: JOMP—an OpenMP-like interface for Java. In: JAVA 2000: Proceedings of the ACM 2000 Conference on Java Grande, pp. 44–53. ACM, New York (2000)
7. Fayad, M., Schmidt, D.C.: Object-oriented application framework. Communications of the ACM 40(10), 32–38 (1997)
8. Klemm, M., Bezold, M., Veldema, R., Philippsen, M.: JaMP: an implementation of OpenMP for a Java DSM. Concurrency & Computation: Practice & Experience 19(18), 2333–2352 (2007)
9. Lee, E.A.: The Problem With Threads. IEEE Computer 39(5), 33–42 (2006)
10. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 2.5 (2005)
11. Ratabouil, S.: Android NDK: discover the native side of Android and inject the power of C/C++ is your applications: begineer's guide. Packt Pub., Birmingham (2012)
12. Satya, K., Dave, M., Franchomme, E.: Pro Android 4. Apress, New York (2012)
13. Satya, K., Dave, M., Sayed, H.Y.: Pro Android 3. Apress, New York (2011)
14. Schordan, M.: The language of the visitor design pattern. Journal of Universal Computer Science 12(7), 849–867 (2006)
15. Sutter, H.: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (February 2005)
16. Vikas, Giacaman, N., Sinnen, O.: Pyjama: OpenMP-like implementation for Java, with GUI extensions. In: International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM) Held in Conjunction with 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013 (2013)