ABSTRACT
We describe Currawong, a tool to perform system software architecture optimisation. Currawong is an extensible tool which applies optimisations at the point where an application invokes framework or library code. Currawong does not require source code to perform optimisations, effectively decoupling the relationship between compilation and optimisation. We show, through examples written for the popular Android smartphone platform, that Currawong is capable of significant performance improvement to existing applications.

1. INTRODUCTION
Modern operating systems have large and complex APIs, and writing software that uses these APIs efficiently can be challenging. There may be multiple ways to accomplish the same task, in which the only differences between two or more alternatives are performance characteristics. Sometimes the right choice for one device is the wrong choice for another device. Even correct and efficient API usage can become inefficient as the API evolves, making the right choice a moving target.

API-related inefficiencies are particularly important for mobile devices, such as smartphones. Their relatively-low-powered processors, slow buses, constrained graphics hardware, and limited RAM make inefficiencies more obvious than they would be on more powerful hardware. Unnecessary use of the CPU or of devices also impacts power consumption, an important factor for hand-held devices.

We propose a novel optimisation technique that addresses API inefficiencies: system software architecture optimisation, or architecture optimisation for short. Architecture optimisation performs work at API boundaries: that is, at the point where application code interacts with the rest of the system. This interaction usually involves system frameworks or libraries, so Currawong improves performance by modifying the way in which framework invocations or library calls take place. Architecture optimisation does not require application source code. This is important, because it means that a system designer or end user could perform API-level optimisation on an application, instead of waiting for the developer to produce a faster version for their hardware. This gives users more control over their applications, particularly if the developer is unable or unwilling to support them.

In this paper we describe the principles behind architecture optimisation through reference to our experimental optimisation tool, Currawong (named after the distinctive Australian bird). Currawong performs architecture-level optimisation of Java programs on the Android platform. Section 2 gives a brief overview of Android and introduces two motivating examples which are used throughout the paper. Section 3 discusses the design of Currawong with reference to the running examples. Section 4 shows the results of running Currawong on the running examples. Section 5 discusses related work, and Section 6 concludes.

2. ANDROID
Android is a mobile operating system developed by Google which runs on smartphones, netbooks, and similar devices [3]. In Android, applications are reliant on a system framework for services, some of which run in a separate process. Applications communicate across processes using a custom IPC mechanism. For example, applications communicate with an external component to draw graphics to the display.

![Android application communication](image)

Figure 1: Android application communication

The framework services that cannot be implemented as application-local libraries are implemented in the System Server, a privileged application which runs in a separate process. An example interaction with the System Server is shown in Figure 1. The application transmits bitmaps to the System Server for display using shared memory (represented by small squares in the diagram). It communicates via function-call-based IPC to instruct the drawing server

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

APSys 2010 August 30, 2010, New Delhi, India.
Copyright 2010 ACM 978-1-4503-0195-4/10/08 ...$10.00.
to update the display (the "Drawing ctl." connector). The System Server communicates via the same IPC mechanism to notify the application of input events, such as a finger moving on the touch screen.

Android applications are written in Java and execute on Dalvik, a custom virtual machine.

2.1 Case studies

We illustrate the design of Currawong with two Android-based examples of architecture optimisation.

The touch events optimisation modifies the way applications are notified of touch events—finger-presses on a touch-sensitive screen. In the standard model, shown in Figure 2 (A), touch events are handled by the System Server component. Data representing the event are marshalled and sent to the foreground application using IPC. In the optimised version, shown in Figure 2 (B), the application reads directly from the relevant device node representing the touch screen hardware. This eliminates the cost of the IPC. IPC in Android is surprisingly slow, so this optimisation can have a noticeable effect.

The redraw optimisation modifies the way applications re-draw their displays. Android offers multiple ways to draw 2D graphics. The recommended method for relatively-static graphics is to use an API called onDraw(). This method is called whenever the display should be updated. This arrangement is shown in Figure 3 (A). Despite it being recommended only for relatively-static displays, this approach is sometimes also used for high-frame-rate games. Unfortunately, the onDraw() method is very CPU-intensive: for every frame, a Surface object, which contains bitmap data, is re-initialised and cleared.

When the optimisation is applied, as shown in Figure 3 (B), several changes are made to the application. The application is modified to start a new thread, labelled “Currawong thread” in the diagram. This thread then calls the appropriate application code to re-draw the surface. Importantly, the Currawong thread maintains a persistent Surface object, rather than re-creating one each time.

3. DESIGN

Figure 4 shows a high-level overview of Currawong. Currawong requires as input a specification and an application.

The specification is parsed and tokenised, and the application is disassembled. The specification is then evaluated. The specification is written in a logic programming language, so evaluation of the specification either fails (in which case the application cannot be optimised by this specification), or it results in a new, rewritten application. In the latter case, this new application is reassembled and written to disk. Each of these parts is discussed in more detail below.

3.1 Specification

Currawong decides how and what to optimise based on an optimisation specification. An optimisation specification consists of two parts: verification that an optimisation can be applied, and then implementation of that optimisation. Figure 5 shows the complete specification of the redraw optimisation. The source code for the touch events optimisation is omitted for space reasons, but is conceptually similar.

Optimisation specifications are written in a custom language named Currawong Specification Language (CSL). CSL is a templated, declarative, extensible logic language. Each of these features plays an important role in making CSL maximally expressive with minimal overhead.

Its logic-language roots mean that CSL is declarative: optimisations are specified in terms of what they should do, rather than how they should do it. In addition to hiding implementation details, declarative specifications are comparatively concise. The features of a logic language are rather well-suited to the type of optimisation that Currawong aims to perform. For example, CSL employs unification as its execution mechanism, as it is based on Prolog [5]. Unification and backtracking along an optimisation specification provide an approximation to the F (eventually) operator of linear temporal logic [10], and other operators can also be represented. It is common to use temporal logics for optimisation specification (see Section 5 for details) because of their expressive power. Currawong provides the same level of expressivity, but does so in a relatively simple way.

CSL is a complete programming language. This means that CSL is extensible—optimisation support libraries writ-
3.3 Finding optimisation candidates

Optimisation candidates are found through matching code templates against application code. This process is driven by match commands within the specification. When matching, Currawong steps through each class in the application, attempting to apply the code template to the class. A class matches a template if the inheritance chain in the class is the same as the inheritance chain in the template, and all methods in the template match those in the class. A method matches another method if its signature is exactly the same. Additionally, match templates may include the special name “–”, which matches any name; or a name starting with a dollar sign (such as “$ClassName”) which both matches any name and makes that name available to the optimiser for further reference.

3.4 Output

To apply an optimisation, changes must be made to the application’s binary code. In the case of Java code, those changes are made to the assembly language representation of the code. The assembly-language files are then re-assembled using the Android-specific Smali assembler [6].

After re-assembly of source code, Currawong re-builds the APK. Because the code has changed, the file must be re-signed using a key created for Currawong. The resulting APK may be installed on the system via any standard method. Re-signing the application does not introduce any security issues by itself, because the unoptimised application must be downloaded and verified using its original signature. A minor annoyance is that updates to optimised applications must be downloaded manually and re-signed. This is not a fundamental issue and could be solved through updates to the application installation mechanism.

3.4.1 Security properties of the optimised system

Applications’ security properties may change after optimisation. For example, code added by the touch events optimisation reads data from the normally-inaccessible device node /dev/input/event0. We propose that Currawong re-use the existing Android security framework to manage this change. Android supports fine-grained security control mediated by a set of permission strings in the application’s manifest. For example, an application may require direct access to the graphics frame buffer. The user is presented with a list of the application’s security requirements prior to installation—thus, the user is informed of, and must explicitly allow, non-standard resource access. Additional fine-grained security requirements (such as access to /event0) could thus be added to Android’s security mechanism to accommodate Currawong. This proposal has not yet been implemented.

4. RESULTS

We applied currawong to several applications resembling existing applications available publicly for Android.

For the touch events optimisation, we replayed a simulated input sequence multiple times (by writing data to the Linux input device /dev/input/event0).
Veldhuizen and Gannon describe an active library as any library that attempts to guide its compiler to produce domain-specific optimisations [12]. In the simple case, this covers any library that makes use of the C preprocessor, or C++ templates, to generate domain-specific code. However, the definition also applies to those libraries which make use of a custom compiler. For example, the author of a matrix manipulation library may wish to include special-case code for the case when an identity matrix is involved in a multiplication. She could do this by writing a special check in the matrix-multiplication routine, but this slows down the routine in the general case. Instead, she may opt to use a precompilation tool which performs partial evaluation. In the cases where it can be discovered at compile time that the identity matrix is passed as a parameter, the active library may instruct the precompilation tool to remove the call entirely.

The Broadway domain-specific compiler is a implementation of many concepts behind active libraries [4]. Broadway’s compilation process is directed by an annotation file describing additional data-flow properties of each function in an active library. Broadway constructs a data-flow lattice for the entire system, which it then uses to make optimisation decisions. For example, Broadway can perform the identity-matrix optimisation described above.

Broadway’s focus on a data-flow matrix for static analysis makes it a very powerful choice for certain classes of libraries, particularly those related to scientific computing. However, the most powerful of Broadway’s optimisations as described in the literature are all for scientific libraries, and it is less obvious that this choice applies to libraries more generally. The data-flow model used by Broadway also means that Broadway requires source code both for the active library and its client application.

By contrast, Currawong’s other influence, refactoring, is a very simple source-to-source transformation technique. Refactorings modify program structure, but do not modify program behaviour [2]. A typical refactoring changes a method’s name, and updates all references to that method to make use of the new name. Thus refactoring is less focused on optimisation than it is on API evolution—adapting old code to new APIs [1]. Refactoring implementations typically require source code, but some Keller and Hölzle’s binary component adaptation scheme can work with Java class files [8]. Currawong extends this concept by allow code modification as well as structural changes.

Currawong is not the only optimiser to work directly with Java bytecode. The Soot optimiser, for example, can perform constant propagation, branch elimination, and copy propagation [11]. Rosser [13] builds on Soot with the inclusion of a specification language based on a temporal logic. Rosser’s specification language. This language, however, focuses on Soot-like optimisations, such as constant propagation and strength reduction; the specification language, with its focus on data flow, is unsuited to the specification of large-scale code transformations. However, Currawong’s and Rosser’s optimisations are orthogonal: a single application could benefit from both approaches.

6. DISCUSSION

We have demonstrated a novel optimisation tool, Currawong, which is driven by a complete general-purpose logic programming language. Currawong distinguishes itself with
its specification language, which provides programmers with a concise way to specify complex high-level optimisations. In combination with the unusual traits of not requiring source code and by working on existing systems, this characteristic makes Currawong a manifestly practical system: we have shown that Currawong can produce significant performance improvements, from small specifications, on production code.

In this paper two APIs have been optimised: touch input, and graphical output. We suspect that other APIs may also be optimisable, particularly those related to other forms of input and output.

We would like to extend Currawong in two directions. The first direction is to improve Currawong’s ability to find optimisation candidates. The examples presented here make use of pattern matching, a relatively simple technique. Pattern matching is ideal for optimisations focused on individual API functions, or a set of calls to API functions, because the specification closely resembles the affected code syntactically. However, pattern matching by itself is unsuited to finding data-sensitive optimisation candidates. A data-sensitive task might involve detecting that the same object is supplied as a parameter to two consecutive API calls, forming the basis of an optimisation which coalesces multiple API calls into one. Other forms of data-sensitive optimisation involve specialisation based on data values. For example, an optimisation for a matrix multiplication library might attempt to detect whether one of the matrices passed to a particular multiplication operation is the identity matrix.

We have added preliminary support for the detection of data-sensitive optimisation candidates to Currawong. Currently Currawong can detect if the same object is passed to two consecutive API functions. This data-sensitive requirement is specified as an additional predicate within the optimise clause, via reference to variables within the template. Internally, Currawong uses symbolic execution to determine whether two parameters refer to the same object. We plan to adding incremental support for additional data-sensitive analyses.

The second direction in which we would like to extend Currawong is towards supporting languages other than Java. We are currently working on adding support for C. Extracting information from compiled C is significantly harder than extracting the equivalent information from bytecode. Currawong builds a model of compiled code in a manner similar to that used by the Cake binary adaptation language. An internal representation is built consisting of each function in the library, as well as all calls it makes to external functions. Android uses industry-standard shared object files for JNI. These are supplied in the ELF format. The information required by Currawong is easy to retrieve, because ELF requires that libraries include information about external functions, as well as their locations, in the ELF header. This produces a list of functions and their outgoing calls. To encapsulate these functions into classes, Currawong makes use of a feature of the JNI specification that specifies a special naming system for JNI-accessible functions. This support is enough to support API modification by replacing API calls. We are currently adding support for static analysis of the kind discussed above, as well as support for adding code to existing binaries, so that a wider range of optimisations becomes available.

References


