Architecture optimisation

Doctor of Philosophy
School of Computer Science and Engineering
The University of New South Wales

Nicholas FitzRoy-Dale

March 2010

Abstract

This dissertation describes *architecture optimisation*, a novel optimisation technique.Architecture optimisation improves the performance of software components or applications by modifying the way in which they communicate with other components, or with the operating system. This is a significantly different focus to traditional compiler optimisations, which typically operate on a single application and do not attempt to change the way it interacts with the rest of the system. To perform an architecture optimisation, the author of a programming interface writes a small, domain-specific *optimisation specification* which describes both the conditions necessary for the architecture optimisation to be valid, and the way in which such an optimisation should be performend. This specification is then used as input to an architecture optimiser, which applies the optimisation to a particular application. Architecture optimisation does not require application source code, effectively decoupling optimisation from compilation.

To demonstrate its usefulness, an implementation of architecture optimisation, named Currawong, is described. Currawong is a complete architecture optimiser, supporting two languages (Java and C) and two completely different software platforms (the Android smartphone operating system, and CAmkES, a research-focused component-based system). Currawong is applied to several optimisable applications on both platforms, and achieves significant performance improvements.

The two major contributions of the work are a concise specification language for architecture optimisations, and early proof that the technique is useful for real-world applications, in the form of benchmark results demonstrating significant (up to 2x) performance improvements.

ORIGINALITY STATEMENT

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

.....
Nicholas FitzRoy-Dale
March 31, 2010

Acknowledgements

Thank you to my partner, Catie Flick, for her love, insight, and surprising willingness to learn about component system arcana. Thank you to my parents, Robyn FitzRoy and Chris Dale, for their unconditional love and support.

Thank you to my supervisor, Gernot Heiser, for his encouragement, and for his enviable ability to consistently provide on-target, insightful comments. Thank you to to my co-supervisor, Ihor Kuz, for his expertise, attention to detail, and unflagging dedication to the cause. Thank you also to Charles Gray and Ben Leslie, whose ideas and feedback were vital to the genesis of this work.

Thank you to the members of the ERTOS group, both past and present, for your friendship and support over the years. It's been a privilege to work with such cheerful, resourceful, and intelligent people.

Finally, thank you to the anonymous reviewers of this dissertation. Your comments improved it significantly.

Contents

List of Figures

List of Tables

1. Introduction

> By what course of calculation can these results be arrived at by the machine in the shortest time?
>
> – Charles Babbage [Babbage 1864]

This dissertation describes *system software architecture optimisation*, a technique to improve the performance of applications by modifying the way in which they communicate with the rest of the system.

1.1. Why optimise?

It is difficult to design efficient system software. This is particularly true for system software that forms the operating system on a device which is tightly resource-constrained, such as a mobile Internet device. Three factors contribute to the problem. The first factor is support for third-party applications: the system must support applications written by programmers other than the system designer, the exact requirements of which are unknown at design time. This means that the system designer cannot take performance shortcuts based on total knowledge of the system. The second factor is a large code base: modern operating systems must support a wide variety of functionality, with the result that they tend to be rather large. This means that keeping track of "corner cases", or undesirable interactions between portions of the software under certain conditions, is harder than it used to be when the average system was smaller. The third factor is unknown or moving hardware targets: architectures which were designed around the requirements and capabilities of one hardware platform are frequently ported to other platforms, often taking a lowest-common-denominator approach to features in the process.

Even though these three design issues are quite distinct, they all nonetheless tend to result in the same kind of application performance problem. Many applications for resource-constrained devices are IO-bound (the case for most applications), or they do a lot of IO but also perform a lot of computation (the case for games). As a result, the efficiency of data transfer between application and *system stack* (constituting all non-application parts of the system, such as system libraries) has a significant impact on the overall performance of the application. This efficiency is directly impacted by the three design issues above.

We can, therefore, generalise a little about these three design issues: they can all affect performance when one portion of the system attempts to communicate with another portion of the system across an interface. In the first case (support for unknown applications) the system designer must anticipate the kinds of things application programmers may

Design issue	Resultant performance issue
Support for third-party applications	Performance shortcuts must be avoided, because they could compromise system security
Large code base	Applications may use the system in unexpected ways, highlighting inefficient corner cases
Unknown hardware	System may be unintentionally optimised for a particular device

Table 1.1. Three design challenges for smartphone system software stacks

want to do, so that interfaces between applications, system libraries, and the operating system are efficient. The second case (overall size complexity of the stack) is similar to the first: designers must anticipate ways in which different portions of the system will be used together by an application. The third case (changing hardware) requires the designer to anticipate the nature of future hardware, so that applications using interfaces designed for the current hardware generation are also efficient on the next generation. These issues are summarised in Table 1.1.

Although the performance issues described above apply to a wide variety of resource-constrained systems, this dissertation focuses on smartphones. Smartphones represent a new but rapidly-growing portion of the computing market. Compared with laptops and netbooks, these devices have lower-powered processors, less capable graphical hardware, and significantly smaller batteries. Nonetheless, their users expect them to perform the same demanding and varied tasks that they would of more expensive systems. The typical modern smartphone can render Web pages, run graphically-intensive games, play video, and work with complex documents. Unlike older "dumb phones", where all the software functionality was determined by the manufacturer, smartphones can run third-party applications. They behave, essentially, like miniature general-purpose computers. The high performance requirements placed on these devices, combined with their modest hardware resources, present a unique challenge for the design of their system software.

The easiest way to improve software performance is to apply standard compiler-style optimisations, such as strength reduction and loop unrolling. These certainly help: with these kinds of optimisations enabled at compilation, code runs approximately twice as fast as equivalent unoptimised code, without any special effort required on the part of the programmer. Standard compiler-style optimisation is, however, usually limited to a single *module*—typically represented by a single text file in popular languages such as C, C++, and Java. This limitation means that standard compiler-style optimisations do not address the particular problems of smartphone system software stack design.

Another simple response to the problems posed is that if the system is designed correctly, the performance problems are avoided. This is certainly true to an extent: system designers have a good idea of the ways in which a system will probably be used, based on knowledge of previous systems, knowledge of the hardware that will be in the device, and so on. However, intuitively, it seems unlikely that such an optimal system can be de-

signed to cater to all unknown applications and future hardware. For example, the API for accessing camera image data in Android phones uses the YUV colour space by default. Future phones may use the RGB colours space instead, requiring applications either to support both formats or suffer a performance hit.

The three system design challenges described above all arise when control or data flows across an API boundary between an application and system software (or between two applications, via the system). This dissertation introduces an optimisation technique, *system software architecture optimisation*, that specifically (and uniquely) addresses performance issues which arise when applications interact with their environment. The name "system software architecture optimisation" is very specific but rather long, so it is usually referred to as "architecture optimisation" in the rest of this dissertation.

Architecture optimisation is a form of context-sensitive optimisation: performing an architecture optimisation requires some semantic knowledge of the code. In this sense it is a specific example of *high-level optimisation*, defined by Veldhuizen and Gannon as "optimizations which require some understanding of the operation being performed" [Veldhuizen and Gannon 1998].

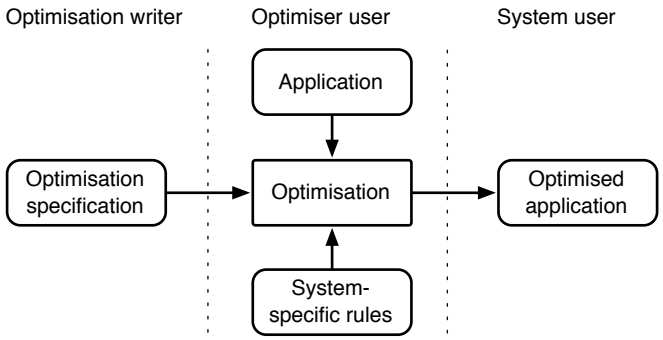

Figure 1.1. System software architecture optimisation process

Architecture optimisation involves the following roles:

The system user: This is the end user of the optimised system.

The optimiser user: This is the person or process using the optimisation tool to produce an optimised application. This may be the system user.

The optimisation writer: This may be the system designer, writing an optimisation for a system sub-system; or it may be a hardware manufacturer, writing an optimisation for a particular piece of hardware.

In general, an architecture optimisation involves the following operations:

1. The optimisation writer thinks of a way in which cross-interface interaction could be improved, or identifies a particular type of cross-interface interaction which causes problems. The problem might exist only for a very specific type of application, or it may only arise on a particular combination of hardware and software.

2. The optimisation writer formalises the optimisation. This formalisation includes a description of the exact conditions under which the optimisation is valid, plus the optimisation itself. Both the conditions for optimisation and the optimisation itself are written in a domain-specific language, and refer to specific features of the system. For example, the optimisation criteria may state that the optimisation is only valid when a specific API call is performed by the application, and the optimisation itself may specify a different API call for the application to make.

3. The optimiser user selects an unoptimised application.

4. The optimiser user runs the architecture optimiser, or an automatic process causes the architecture optimiser to be run at an appropriate time. This optimiser takes as input an optimisation description, or set of optimisation descriptions, and an unoptimised application. The optimiser determines whether the optimisation is valid, and, if it is, modifies the application according to the optimisation description.

5. The output of the optimiser is a new application, which is installed on the system, ready for use by the system user.

In step 4, the optimiser verifies the optimisation. To facilitate this process, the optimisation begins with zero or more statements of fact about the application. The optimiser verifies that each statement of fact is correct about the particular application being optimised. The statements of fact fall into two categories:

- Statements about the *path-insensitive data flow* of an application. These are statements regarding structural features of the application code. For example, "The application makes a call to library function x" is a statement of this form.

- Statements about the *path-sensitive data flow* of an application. These are statements which require the optimiser to make statements about the values of data in the application relevant to the optimisation. The statement "When the application calls x(a), a is always equal to 1" is a statement of this type.

Once the optimisation has been verified, the application is modified by applying one or more *remedies*. A remedy is a modification to the application (for example, by changing the class that application makes use of). These are covered in more detail in Chapters 4 and 5.

The language in which the optimisation is written is customisable and extensible, both to add support for new types of verification, and to add support for modifying applications in a particular system. A graphical representation of the whole optimisation process is given in Figure 1.1.

The above sequence identifies the *optimiser writer*, *optimiser user*, and *system user* roles, but does not specify whether these roles should be filled by the same person, or by separate people. The system is designed so that the optimiser could run at application installation stage, possibly without knowledge of either the application writer or the end user. Architecture optimisation is completely independent of other forms of optimisation,

and can thus, for example, be used in conjunction with compiler-style optimisation to synergistic effect.

1.2. Currawong: an architecture optimiser

The design presented in this dissertation is implemented in Currawong, an architecture optimiser named after the distinctive Australasian bird. Currawong is an architecture optimisation testbed, capable of performing a variety of architectural optimisations. Currawong rigidly enforces separation of the roles illustrated in Figure 1.1: the optimisation writer, optimiser user, and system user can be completely separate entities. Currawong is used by the Optimisation Writer and Optimiser User roles: by the former in order to develop and appropriate optimisation, and by the latter to apply the optimisation to a particular application. Importantly, the final role, System User, does not involve Currawong at all: because Currawong optimisation is a pre-runtime process, optimised applications do not rely on Currawong at run-time.

A key component of Currawong is a method by which optimisations are specified. Currawong focuses on keeping specifications concise and high-level, so it uses a declarative and extensible specification language, but hides the details of the process from the optimisation writer. For example, an optimisation writer can specify a particular class or function call to optimise, but need not specify exactly how to find that piece of code, nor the details of making changes to the system.

Currawong supports multiple methods for verifying that an optimisation is correct, some of which are built in to the syntax of the specification language. It is capable of performing architecture optimisation on two systems: the Android smartphone platform, as well as a microkernel-based research component system.

An important difference between Currawong and many other optimisation tools is that this technique does not require application source code. This is a highly practical choice, because it means that application developers do not need to be involved in the process of optimising their own applications. Obviously, it is preferable for developers to support their own applications, but there are many reasons why a particular developer may not apply a particular architecture optimisation:

- The optimisation may be specific to hardware or software that the developer does not have. For example, an architecture optimisation may only work with the latest version of the phone's operating system, or may only support a particular manufacturer's touchscreen.
- The developer may have discontinued support for the application.
- The developer may simply not realise that their application can be optimised in this way.

End users, rather than application developers, can thus apply their own optimisations. Alternatively, suitable system-specific architecture optimisations could be applied to applications without any user intervention required.

Non-reliance on application source code is particularly useful for API evolution, which is the practise of updating the API of a component or library to assist with maintenance of that library, to add support for new features, or to improve the library's performance. Because API modifications are a core motivation for architecture optimisation, and architecture optimisation techniques encompass API evolution techniques, API evolution may be considered to be a subset of architecture optimisation.

Currawong's effectiveness is demonstrated by variety of optimisations for several applications on two different systems.

1.3. Related approaches

Architecture optimisation builds upon two large areas of related work: library optimisation, and component-based software engineering. Work on domain-specific and library-level optimisation is motivated by some of the same goals that motivated this work, and implements similar solutions. The general idea is that a library author supplies a number of domain-specific optimisations along with the library. The application programmer then uses a custom-written optimising compiler to apply the optimisations to application code. As with architecture optimisation, the advantage to the application programmer is that he or she need not know the best way to make use of the library, because that information is effectively supplied with the library in the form of annotations. This approach provides some, but not all, of the advantages of architecture optimisation. Because library-level optimisations involve compiler modifications, they are restricted to a single source language. Additionally, library-level optimisations require application source code, making them unacceptable for API evolution purposes. Because of these limitations, the optimisation specification language is irrevocably linked to the target language, making the specifications difficult to generalise.

Optimisation of communication across an interface boundary is a core area of research in the general area of component-based systems, and several approaches attack the same general problem by providing a flexible component system: either one that offers applications a choice in the way they communicate with other components (or the operating system) or that decides, through simulation or other heuristic, which communications method would be best. Architecture optimisation builds on that approach. These techniques are typically only applicable at system creation time, when the source code is available. Additionally, this work supports performance analysis of a system for which only binary code is available, and thus delivers more flexibility with respect to when the optimisation can be performed.

1.4. Scope

The variety of use cases presented above show that architecture optimisation is a rather general concept. Architecture optimisation could apply to application-to-application, rather than application-to-operating system, optimisation; it may also apply to systems other than smartphones, such as laptops and netbooks; and the same techniques may be

usefully applied in distributed systems. This is too broad a scope on which to build an experimental framework, so some degree of scope narrowing is required.

As discussed above, smartphones are an attractive optimisation platform. To recapitulate: smartphones have significantly slower processors and slower graphics accelerators than desktop devices, but are expected to provide desktop-like performance in areas such as gaming and Web browsing. Smartphones provide rich APIs, but, because the genre is relatively new, it is not necessarily clear to developers which APIs are most performant for a given application. Compounding this confusion is the heterogeneity of hardware available, resulting in varying performance capabilities between phones. Finally, smartphone operating systems are evolving rapidly, resulting in a heterogeneity of installed operating system versions. Smartphones also have the practical advantage of large and easily-searchable ecosystems of third-party applications, in the form of "application stores" for the various platforms. This dissertation presents performance benchmarks taken from real applications taken from such a store.

I selected applications which moved large amounts of data between application and operating system as domain applications to optimise. I chose these sorts of applications specifically because they rely on efficient application-to-system data transfer in order to perform acceptably, and thus stood to benefit from architectural optimisation.

Architecture optimisation is aimed at the middle ground between individual program modules and the entire system. It is not concerned with techniques that can be performed just as effectively by existing optimising compilers. It is also not concerned with exhaustive whole-system analysis techniques (such as model checking). The state of the art in the area of techniques such as model checking is not at the point where such computationally-intensive techniques can be aimed at the very large amount of code that makes up a smartphone system software stack plus its applications.

For similar performance-related reasons, the examples presented in this dissertation involve only modest amounts of static analysis. There are two reasons for this. A simple and pleasant reason is that for the class of optimisation for which Currawong is intended, a large amount of static analysis turned out not to be necessary. A second reason is that because architecture optimisation involves whole-program analysis, potentially with many hundreds of optimisation descriptions, each resulting in one or more program modifications, it is important to keep processing time to a minimum, to minimise the state explosion problem.

The case studies presented here are limited to two systems: the Android commercial smartphone operating system, and a simple demonstration operating system based on the OKL4 microkernel. These choices are largely arbitrary, but they are both useful for demonstration purposes as both are particularly open systems.

Some discussion should be given to the matter of energy conservation. This is often supplied as a justification for reducing the CPU workload on mobile platforms. However, power management is outside the scope of this dissertation, and energy consumption is mentioned only indirectly—I made no attempt to quantitate energy saving as a result of optimisation, for example. Efficient energy usage is simply another benefit of reducing overall CPU and peripheral power consumption through architecture optimisation techniques.

1.5. Contributions

The major contributions of this work are:

- the concept and description of binary-only, interface-level optimisation, and case studies demonstrating, via implementation of the technique, that it is viable and useful;
- a domain-specific and target-language-independent language for the architecture optimisation description;
- an implementation of the technique described, named Currawong; and
- a binary-only component specification for microkernel-based embedded systems.

1.6. Overview

The dissertation proceeds with a study of related work in Chapter 2. Chapter 3 (Background) provides a brief introduction to the software stacks referenced in the dissertation—the OKL4 microkernel and the Android software stack in particular. Chapter 4 (Performance anti-patterns) gives a detailed motivation for the work and introduces a set of running examples. The motivation is used as a basis for design of the system, which is given in Chapter 5. An implementation of an architecture optimiser, named Currawong, is described in Chapter 6 (Implementation). An evaluation of Currawong is then presented in Chapter 7 (Evaluation). Finally, a summary and overall evaluation of the work is given in Chapter 8 (Conclusions).

2. Related work

> Machines take me by surprise with great frequency.
>
> – Alan Turing [Turing 1956]

Before the major optimisation methods are discussed, a short introduction to *traditional compiler optimisation* is given, to serve as a point of reference for other optimisation techniques. I then identify three major optimisation methods, and discuss each one in sequence.

The first method, *dynamic optimisation*, concerns rewriting that occurs in real time (that is, at the same time as the execution of the code). The second method, *source-based rewriting*, works with source code. This type of rewriting either involves a separate optimiser, or makes use of a special type of compiler capable of performing both high-level optimisation and compilation. The final method, *pre-execution binary rewriting*, is a compromise between dynamic and source-based approaches.

Currawong also builds on work in the fields of component-system optimisation, dynamic upgrade, and program transformation metalanguages. Each of these areas is discussed separately, after the three major optimisation methods.

2.1. Traditional compiler optimisation

Traditional compiler optimisation refers to the set of techniques employed by modern *optimising compilers* to improve the compiler's output in some way, most commonly to make the compiled code faster. Architecture optimisation does not rely on optimising compiler techniques. They are, however, often used as a point of reference, so it is worth discussing them here.

Optimising compilers have a long history. The second compiler in the world, released in 1957 for the FORTRAN language, was capable of simple optimisations. For example, this compiler performed *common subexpression elimination*, in which sub-expressions of an arithmetic expression were only evaluated once, even if they appeared multiple times [IBM 1956].

Modern optimising compilers perform a wide variety of optimisations, which may be classified in various ways. For example, *intra-procedural* optimisations work only with a single function, whereas *inter-procedural* optimisations make use of multiple functions. In the former category are optimisations that operate on loops, such as *loop unrolling*, in which the body of a loop is copied one or more times (with a corresponding reduction in the number of loop iterations) to reduce loop overhead and increase instruction parallelism. In the latter category are optimisations such as *function inlining*, in which the

body of a called function is copied verbatim into the body of the calling function, in order to eliminate function-call overhead [Bacon et al. 1994].

Some optimisations require information about the run-time behaviour of the application. For example, if the compiler knows that the probability of a particular branch instruction being taken is very likely, it can reorganise the code so that the cost (in terms of cache misses) of taking the branch is low. Historically, compilers have not had access to run-time performance profiles of the application, so information on the probability of various branches had to be provided by programmers, in the form of compiler-specific instructions (*annotations*) embedded in the source code. However, some modern optimising compilers can compile code in a special mode to enable *trace-based profiling*. This causes the application to generate a log file containing run-time performance information, such as the number of times various branches were taken. After the application has been run (perhaps multiple times), the compiler re-compiles the application, taking advantage of the supplied performance information [Gatlin 2010].

2.2. Dynamic optimisation

Dynamic optimisation is the technique of optimising code as it runs, rather than *before* it is run. Dynamic optimisation can apply to both native code and bytecode. Dynamic optimisation is most well-known as one of a number of techniques used by just-in-time compilers, or JITs, to produce efficient native code from bytecode. JITing is, itself, most well-known for optimisation of Java bytecode. In this setting, a lot of useful information is known to the JIT at run time, so there is potential for significant performance improvement. For example, the entire application and all dependent libraries are available to the optimiser, making multi-function optimisation feasible; all data types are known (if it is a Java virtual machine), allowing for domain-specific optimisation; and commonly-taken code paths ("hot paths") are known, acting as a guide to direct the optimiser. JIT is not at all a new idea: despite the recent popularity, JIT systems have been available for various languages since the 1960s [Aycock 2003].

The challenge with JIT compilers has always been in balancing advanced optimisation against increased application latency: when a portion of the application is to be optimised, the compiler must run before the application can continue. Perversely, the better the compiler is at identifying (and, therefore, being required to just-in-time compile) critical portions of code which must run quickly, the more acute this problem becomes. This issue is so important that recent versions of Sun's Java HotSpot virtual machine actually contain two JIT implementations: a *server compiler*, which sacrifices some latency to achieve better optimisation; and a *client compiler*, which makes the sacrifice in the opposite direction, discarding optimisations that would significantly increase perceived latency [Kotzmann et al. 2008].

HotSpot can perform traditional multifunction optimisations, such as function inlining; and can co-locate frequently-referenced objects to improve cache performance (i.e. using temporal locality to optimise spatial locality). This latter example, which requires run-time locality information, demonstrates a benefit of JIT compilers in general over tra-

ditional before-execution compilers ("pre-compilers"). This advantage is, however, being slowly eroded by modern profile-guided optimisation [Gatlin 2010].

Given the inherent performance challenge described above, dynamic optimisers are at their worst when optimising native code. In this situation, the dynamic optimiser starts at a disadvantage: running any dynamic optimiser code means not running application code, so the act of optimising itself reduces the performance of the application. This is not so noticeable with JIT compilers for virtual machines, because the JIT activity can take place concurrently with other activities necessary to execute the non-native code (such as creation of data structures, or even during interpretation).

Mojo is a dynamic optimising compiler for native code on the Windows platform [Chen et al. 2000]. The Mojo compiler provides a good reference point for this sort of dynamic optimiser. Mojo follows other dynamic optimisation systems, such as Dynamo [Bala et al. 2000], by attempting to find "hot traces" in executed code, where a hot trace is a commonly-executed sequence of instructions. Mojo then atttempts to rewrite the code implementing the hot trace to eliminate jump instructions on the critical path. Off-path jumps execute as normal. In their paper, Mojo's authors describe their experience applying Mojo to a number of large commercial applications for the Microsoft Windows operating system—a tough challenge. Unfortunately, Mojo's performance was lacklustre: application execution times generally increased significantly under Mojo. The authors attribute this to the overhead imposed by the optimisation engine itself.

One commonality with dynamic optimisation systems is their relative immaturity compared with source-code-based optimisation techniques, discussed in more detail below. The systems are generally not extensible, for example, and do not implement domain-specific optimisations. Further, it seems that implementing good domain-specific optimisations while not slowing the system down is a significant, and perhaps impossible, technical challenge. This is particularly relevant on embedded systems, where processor and memory resources are tight.

2.3. Source-based rewriting

Source-based rewrite engines represent the largest and most diverse class of all optimisation types examined. Unlike dynamic rewriting, which is commercially successful, most source-based rewrite engines remain highly specialised research systems. The two major sub-categories discussed below are those that only modify the structure of the code (*refactoring engines*) and those that also modify the code's behaviour.

2.3.1. Refactoring

Refactoring represents the simplest type of source-based transformation. It is a program transformation that changes the structure of a program but does not modify its behaviour [Fowler et al. 1999]. A simple example of refactoring involves renaming a class. Once the class itself has been renamed, all references to it are updated to use the new name. Although refactoring is not directly used to perform optimisation, refactor-

ing can frequently assist with *API evolution*, by helping programmers modify their source code in response to API changes. One study of several large Java programs which support plug-ins showed that 80% of the API changes to the plug-in interfaces could be modelled as refactorings, and the authors conclude that better support for automated refactoring tools would help programmers deal with changing APIs [Dig and Johnson 2005].

API evolution is an important and difficult problem, and is of direct interest to this thesis. A recent study showed that API developers deal with API evolution in a variety of ad hoc ways. One de facto standard is to use *deprecation*. A class or method that has been deprecated is still fully functional, and its behaviour does not change. It is, however, marked as "deprecated" in API documentation. Depending on the programming language, the compiler may also display a warning if a programmer uses deprecated functionality. Deprecation places the onus on application programmers to update their applications, an expectation that sometimes backfires: deprecated classes in the Java API, for example, have sometime been "un-deprecated" because of their continued use; or, if they are eventually removed, application developers may use older versions of the API, with the result that the library developer ends up supporting multiple versions of the library [Henkel and Diwan 2005].

Another ineffective solution is simply to expand the API, supporting both the legacy interface and the new one. This takes the onus of support off the application programmer—the correct thing to do—but results in ever-larger APIs, with corresponding support and comprehensibility problems, particularly for new programmers ("Which of these two similar solutions should I use?"). The authors of Catchup!, an experimental automated refactoring tool [Henkel and Diwan 2005], suggest that the difficulties associated with API evolution act to strongly discourage API developers from making incompatible changes, and speculate that reducing the cost to upgrade client code may ultimately result in better APIs. The Catchup! authors also suggest that support for more advanced static analysis would make their tool more useful. In particular, they identify support for simple temporal constraints, such as "calls method X, then calls method Y" as a useful addition.

Despite Catchup!'s age (it is 10 years old, at time of writing), there has been little work in the area since then. One more recent project, ReBA [Dig et al. 2008], presents an alternative approach: a client-specific "compatibility library" is generated. This stub library supports the old API interface, but transparently translates calls to use the new interface. This approach has limited optimisation potential—it was not designed for optimisation—and requires source code for the application.

2.3.2. Beyond structural modification

Aspect-oriented programming

Aspect-oriented programming (AOP) resides somewhere between refactoring, as discussed above, and active libraries, discussed below. It is a software engineering technique in which the code implementing an application is divided into several *aspects*—where an "aspect" represents a distinct functionality of the application. Aspects are separated out in this manner even if—in fact, especially if—the functionality they encompass cannot

cleanly be encapsulated in a class or module using traditional modular design techniques. The goal of AOP is to enable better "separation of concerns", by representing the concerns as aspects, rather than as traditional modules, classes, and functions [Kiczales et al. 1996].

AOP attempts to address what has become known as the "library scaling problem", after Biggerstaff [Biggerstaff 1994]—the observation that as components become larger, they become more specialised: less generally useful, but more useful in their particular domain; and, conversely, as components become smaller, they become more generally useful, but less useful for any particular purpose. AOP attempts to deal with this quandry by providing another alternative to the traditional module-based system that defines libraries, addressing what Biggerstaff refers to as "limits of representation".

Canonical examples of aspects include code which ensures that locks are taken and removed at appropriate times; calls to log the progress of the application for debugging purposes; and code to manage the memory allocation of the application. The theory is that once these aspects are removed from the core application, the remaining code, or *base aspect*, becomes much easier to understand. The application becomes less prone to bugs, because programmers maintaining one aspect need not worry about the other aspects: most significantly, programmers working on one aspect can focus entirely on it without being distracted by code implementing other aspects.

At build time, all the aspects are combined into a single application using a source-to-source compiler called an *aspect weaver*. Each aspect includes one or more *point-cut* specifications, which are essentially a declarative description of where code implementing the aspect should be included in the base aspect. The aspect weaver uses these point-cut descriptions to *weave* a complete application.

Notably, point-cuts can only insert code—they can not modify what is already there. This follows from the core concept of AOP as a program composition methodology.

AOP is interesting from a program transformation perspective, nonetheless, because the point-cut specifications must somehow refer to specific positions in the code at which the new functionality (known as *advice*) must be added. In effect, each AOP implementation (i.e. each implementation of an aspect weaver) defines a matching by which portions of code may be identified.

Point-cuts typically identify structural references, such as "class name starts with set_" or "within the com.example namespace".

Active libraries

Veldhuizen and Gannon coined the term *high-level optimization* to refer to optimisations that require "some knowledge of the operation being performed" (as mentioned above) [Veldhuizen and Gannon 1998]. They distinguish this from *low-level optimisation*, which can be performed "without any knowledge of what the code is supposed to do". Traditional compilers perform a known set of low-level optimisations over which developers have very little control, but a relatively-recent trend in research compilers is to support experimentation with (at least) low-level optimisations without requiring the experimenter to have expert knowledge of the compiler's code. These compilers, no-

tably SUIF [Wilson et al. 1994] and LLVM [Lattner and Adve 2004], add support for user-specifiable transformations within a larger compiler infrastructure, and have optimisation research as a specific goal. These compilers provide structured access to their intermediate representation(s) (IR) so that optimiser writers can experiement with custom "passes".

This approach works well for research into low-level optimisations: LLVM in particular boasts an impressive set of well-modularised optimisations. Access to the intermediate representation would seem, however, to be the wrong level to implement high-level optimisation. Related work in this area has tended to focus on expressing properties about the code to be optimised either in a domain-specific language devoted to specifying code properties, or in the implementation language itself. Both options are preferable to the intermediate representation as they more closely resemble the original code.

The most coherent expression of this idea was popularised by Veldhuizen and Gannon as *Active Libraries* [Veldhuizen and Gannon 1998]. The concept as originally described is quite general: any library that attempts to guide its compiler to produce domain-specific optimisations counts as "active". This definition includes certain types of C preprocessor usage, or C++ templating features, but also covers techniques requiring special compilers or preprocessors—in particular, partial evaluation (the technique of determining the values of one or more parameters to a function at compile-time, inlining the function, and then compiling as if the identified parameter were a constant).

Research on active libraries has tended to focus on libraries for scientific computing, because they are often structured as a set of functions operating on a small number of data structures. This format makes them particularly amenable to the wide application of a small number of optimisations. For example, a parallel matrix operation library contains a number of functions to perform various matrix operations. Each function takes one or more matrices as arguments. An active library can make use of the mathematical properties of matrix functions to produce application-specific optimisations. For example, the library could replace multiplication by the identity matrix with a no-op.

The techniques proposed by Veldhuizen and Gannon were refined and extended in the Broadway domain-specific compiler [Guyer and Lin 2005]. Broadway is a source-to-source compiler which uses dataflow analysis techniques to implement domain-specific optimisations. The compiler requires application source code plus an *annotation file* for a library used by the application. That annotation file consists of annotations, each of which apply to a particular function in the library (Broadway also supports global annotations which apply to all functions. They do not function significantly differently to per-function annotations, so are not discussed further here). These annotations specify a number of properties about the function: they comprise an extensible domain-specific language. In particular, data flow behaviour of the function in terms of its input and output parameters is described, so that Broadway can construct a dataflow lattice for the whole system comprising application and library.

Broadway annotations primarily perform optimisation by allowing inlining, and by replacing function calls with more domain-specific calls if certain properties of the input parameters can be shown to hold. One of the optimisations applied by Broadway is the identity-matrix example described above.

Broadway is flexible for two reasons. Firstly, the underlying analysis technique is generic and proven, in that the dataflow analysis technique, and its capabilities and constraints, are well-known and described in other literature. Secondly, annotations are supplied with libraries, rather than the compiler itself, allowing for an unbounded number of library-specific optimisations. The authors argue that while it may be difficult to come up with the annotation, that difficulty is only encountered once, by the library authors, and the benefit can then by enjoyed by all users of the library. Another benefit of this division of labour is that the optimisation is done by a domain expert—the library author—rather than an application programmer who may not be as familiar with the internals of the library.

It should be noted that the idiosyncratic structure of scientific-computing libraries—a large function set operating on a small data-structure set—is not common in other types of libraries, and there is no guarantee that the techniques which can be applied to validate optimisations in scientific libraries will apply equally well to other libraries. Indeed, it seems unlikely: Broadway shows its most promising results when applied to linear algebra and matrix operation libraries; when it was applied to various non-scientific libraries it was restricted to simple inlining optimisations and to error-checking. For example, Broadway could warn at run-time if an application attempted to read or write to a file that was known to be closed. This is not to imply that the actual *optimisations* implemented by Broadway (i.e. various types of specialisation) are unsuitable but, rather, that Broadway-style dataflow analsyis may not, by itself, be the best approach for optimisation of a heterogeneous collection of libraries.

Several projects address the problem of API evolution using methods that extend beyond function-call matching. Coccinelle [Brunel et al. 2009], for example, defines a domain-specific *semantic patch* language in order to support API evolution. In the Coccinelle system, a semantic patch describes a set of source-code level changes to code which must be made to support an API evolution. The "semantic" portion of the name refers to the way these patches are applied: rather than doing a simple textual match, Coccinelle parses both patch and code and uses static analysis to determine whether the meaning of the patch is present in the code. This allows Coccinelle to apply semantic patches even when the code being patched does not superficially resemble the text of the patch. Figure 2.1 shows a Coccinelle semantic patch to add a parameter to a function, replacing code which would generate the information as a local variable.

Currawong's templating system, discussed in Section 5.3, echoes the semantic patch ideas of Coccinelle.

2.4. Pre-execution binary rewriting

Pre-execution binary rewriting, also known as link-time binary rewriting, provides a compromise between the source-based and dynamic transformation methods. In some ways, it is the best of both worlds. Since the optimiser can run before the application, the speed of optimisation is less critical, which means that the optimiser can do more work and perhaps produce better optimisations. No dependence on source code means that authors

```
@@
function a_proc_info;
identifier x,y;
@@
   int a_proc_info(int x
+                    ,scsi *y
          ){
-    scsi *y;
     ...
-    y = scsi_get();
-    if(!y) { ... return -1; }
   ...
-    scsi_put(y);
   ...
   }
```

Figure 2.1. A Coccinelle semantic patch, replacing local generation of "y" with a parameter.

of frameworks or libraries can release updated optimisations without requiring application authors to recompile their code. The major problem with binary transformation is the lack of type information, a problem because type information is a useful way to specify constraints on optimisation. This issue is solved in various ways depending on the implementation language.

The refactoring systems discussed above operate on source code, but binary-only refactoring systems have also been developed. Keller and Hölzle describe a *binary component adaptation* system—their term for an extended refactoring system which works exclusively with binary code [Keller and Hölzle 1998]. Their system loads Java byte code and scans for structural features (such as class and method names). These features are then modified according to a small declarative language. The focus on binaries makes the system a little more flexible, because the refactorings which are necessary to deal with API evolution do not need to be implemented by the programmer. Instead, the system automatically refactors code at load time, so that it works with the installed API. This was taken one step further with PROSE [Popovici et al. 2002], which extends aspect-oriented programming techniques to compiled Java code.

Native binary rewriting is more challenging than rewriting of byte code. In a binary, the high-level control flow information is lost, as are the data types. These must be re-created or inferred. Some of these challenges are addressed by the DIABLO binary rewriting system [Put et al. 2005]. DIABLO is a "link-time rewriting framework", which can reduce the space used by binaries by removing unused code. It finds dead code by examining all jumps taken by the application to produce a "whole-program control flow graph". Dead code can be inferred by determining, for each byte of program code, whether it can be referenced from the graph. Unfortunately, most systems-oriented languages are known to be difficult to analyse in this regard due to their support for function pointers. There is no indication that DIABLO deals with these correctly.

2.5. Optimisation in component-based systems

The same techniques that have been applied to libraries can also be applied to component-based systems. The VEST system is an implementation of AOP for real-time component-based systems. In VEST, aspects can cross-cut component boundaries across control-flow paths (i.e. the criteria for code insertion can involve multiple components) [Stankovic et al. 2003]. Because it is designed to produce real-time systems, VEST also employs schedule checking tools to ensure the execution time of the resulting transformed system is within acceptable bounds.

Of the large selection of component-based systems for embedded systems [Friedrich et al. 2001], many support some degree of optimisation at the level of component description. A simple example is the Knit framework for the Flux OS Toolkit component system [Reid et al. 2000, Ford et al. 1997]. Knit acts as a component-aware linker, allowing inlining across component boundaries, as well as performing other non-optimisation-focused tasks. An alternative approach, the Pebble component-based operating system, supports hardware-mediated memory protection, but allows components to communicate via *portals*. Portal communication involves executing a small amount of code in the kernel. Normally this code is the same for all communicating processes in the system but, in Pebble, components can specify their own portal traversal code. This code executes with kernel privileges, so Pebble uses a number of mechanisms to make sure that safety properties are enforced: portal traversal code is written in a domain-specific language and compiled and checked by Pebble [Bruno et al. 1999].

2.6. Dynamic upgrade

Some problems addressed by Currawong resemble the problem of *dynamic update*, that is, that is, replacing one or more binary components in a running system through run-time binary modification. One such system, the K42 Dynamic Update system [Baumann et al. 2005], relies on details of K42's object system to implement dynamic update with minimal service interruption. In K42, upgradable parts of the system are implemented as *clustered objects*, which expose a functional interface. Other parts of K42 access clustered objects via indirection through an *object translation table* (OTT). Dynamic update is implemented by a *hot-swapper* making use of the object translation table. To begin an update, the hot-swapper interposes *mediatior* objects in the OTT. These mediators can at first track, and then delay, access to the clustered object being upgraded, allowing the hot-swapper to wait for quiescence in the system, prevent threads from accessing the clustered object whilst it is being upgraded, and finally to replace references to the mediator object in the OTT with references to the new object.

Upstart [Ajmani et al. 2006] also implements dynamic upgrade for distributed systems using a mediator-style approach. Upstart-upgradable systems communicate using an RPC protocol which is dispatched by a mediator running on each node. When performing an upgrade, nodes replace calls to functions representing the object with calls to mediator objects which perform one of a number of tasks related to the upgrade. These systems

and Currawong both perform updates on binary code through interposition of functions.

2.7. Metalanguages

This section discusses languages that describe program transformations. To avoid confusion, I follow Kleene's nomenclature by distinguishing between the *object language*—the language in which code to be transformed is written—and the *metalanguage*—the language which describes those transformations [Kleene 1967].

The most popular refactoring tools rely on Eclipse [Eclipse Foundation 2010a]. Eclipse is a large program-development system comprising an integrated development environment (IDE), complete with editor, language reference, and debugger; a refactoring engine; and a refactoring API supporting plug-ins (to enable third-party refactorings), as well as many other features.

Most fundamentally, however, Eclipse is an IDE. As a result, Eclipse-based refactoring tools tend to be focused on graphical user interaction and not automated—the programmer must indicate which refactorings to perform. The metalanguages of choice for programs which perform only refactoring—i.e., structural manipulation—tend to be rather similar: They are usually imperative, and follow a top-down parsing model based around a syntax tree (to some greater or lesser degree of abstraction).

A good example of this style of refactoring language is Eclipse itself [Eclipse Foundation 2010b]. Eclipse refactorings are written in Java and run as a sequence of callbacks—in Java terms, refactorings are classes implementing a Eclipse refactoring interface. A small portion of a complete refactoring to rename a property in Eclipse is shown in Figure 2.2.

```
private Change createRenameChange() {
  // create a change object for the file that contains the property
  // which the user has selected to rename
  IFile file = info.getSourceFile();
  TextFileChange result = new TextFileChange( file.getName(), file );
  // a file change contains a tree of edits. Add the root of them
  MultiTextEdit fileChangeRootEdit = new MultiTextEdit();
  result.setEdit( fileChangeRootEdit );
  // edit object for the text replacement in the file
  ReplaceEdit edit = new ReplaceEdit( info.getOffset(),
  info.getOldName().length(),
  info.getNewName() );
  fileChangeRootEdit.addChild( edit );
  return result;
}
```

Figure 2.2. Renaming a property in Eclipse (example from Eclipse documentation)

Because Eclipse refactorings must integrate with the rest of Eclipse, which is primarily a programmer's IDE, the refactoring language is more complex than necessary: in the ex-

ample, most of the code consists of creation and manipulation of objects that describe the change to Eclipse in terms of an edits tree, obscuring the code transformation being performed. Other required portions of the refactoring language include exception handling, and code related to keeping the user interface updated—all of which could be expressed separately. Obviously, there is room for improvement.

Several languages exist which address the lack-of-clarity issue common to Eclipse-style program transformation by replacing the imperative-style transformation language with a declarative one. The Tree Transformation Language (TXL [Cordy 2006]) provides one such implementation. TXL was not designed for refactoring, or indeed for program transformation at all *per se*; it was designed to help with rapid prototyping of programming languages. However, the metalanguage used by TXL is quite interesting from a program-transformation perspective, so it is worth discussing here.

TXL is a programming language transformer. A complete TXL specification effectively defines two object languages: the original object language O_o, and the transformed object language O_t. O_o is specified using an EBNF-like grammar as used by parser generators such as `yacc`. O_t is specified in one or more *replacement rules*, which are written in the metalanguage. A simple replacement rule is shown in Figure 2.3. This rule replaces statements of the form `V := V + E` with a statement of the form `V += E` (in other words, it translates standard assignments to augmented assignments). TXL uses a simple extensible metalanguage capable of including embedded portions of O_o and O_t. This feature, known as "pattern matching" within the functional-language community, provides a very concise, expressive method to represent code changes. Also notable is TXL's notion of types, which are derived from the parse tree. This type concept is notable because it means that TXL's concept of types may differ from the object language's concept.

```
rule simplifyAssignments
  replace [statement]
    V [reference] := V + E [term]
  by
    V += E
end rule
```

Figure 2.3. A TXL replacement rule.

TXL is by design language-agnostic, with the result that it is incapable of performing most context-sensitive transformations (except when the context refers to type information derivable from the parse tree). Keller and Hölzle's language for refactoring of Java bytecode [Keller and Hölzle 1998](discussed above) also resembles a simple pattern-matching language. This language is, however, Java-specific, and by virtue of being so specialised it is even more concise than TXL. The language consists of the Java grammar plus a number of refactoring-specific keywords. The writer of the refactoring constructs a file describing the refactorings to perform, known as a *delta file*, an example of which is shown in Figure 2.4. The delta is Java-like, apart from the keyword "delta", and the prefix "add method". This delta concisely expresses the modification to perform, in a language which Java programmers already understand.

```
delta interface Enumeration {
  add method public Object lastElement() {
    /* Java code omitted */
  }
}
```

Figure 2.4. A binary delta to add a method to all classes implementing an interface

Metalanguages for programming-language-independent specification have been developed, with the justification that presenting a consistent language-independent interface will aid in the development of higher-level refactoring tools. An example is FAMIX, a metalanguage which supports transformations in Java, C++, and Smalltalk [Tichelaar et al. 2000]. This approach seems rather self-limiting. A language to describe multiple object languages must either limit itself to the language features shared by every object language—the lowest-common-denominator approach—or it must support all features of every object language—the inclusive approach. Neither option is appealing. In the lowest-common-denominator case, not all possible object language transformations will be possible in the metalanguage. In the inclusive case, the metalanguage is more complicated than necessary for any particular object language, and some transformations that can be expressed in the metalanguage cannot be implemented in object languages which do not support a particular metalanguage feature (such as multiple inheritance).

2.8. Conclusion

There is a lot of work in the area of program analysis for bug finding or security checking [Engler et al. 2001, Yang et al. 2006, Ball et al. 2006, Ball and Rajamani 2001, Holzmann 1997], but there is relatively little dealing with optimisation or API evolution as a general problem. Perhaps the most promising area of work is in the field of active libraries, but these tend to be limited for two major reasons: firstly, they generally require source code; and, secondly, they tend to focus on a single analysis technique.

Requiring program source code for optimisation makes providing advanced analysis techniques, such as data-flow analysis, relatively easy. However, the reliance on source code heavily restricts the applicability of the techniques. Deployed software is almost always distributed in binary form. Even if the most advanced analsis techniques cannot be applied to binary code, there is room for an optimiser that supports a simpler technique but remains binary-only.

There is no categorical best approach to determining information about a program for transformation purposes. Rather, the best approach depends on the nature of the transformation. Candidates discussed above include control-flow matching, data-flow analysis, and structural analysis. An ideal program transformer would support multiple analysis methods, and allow them to be used interchangeably. There is little related work in this area, though preliminary documentation on the Cake transformation system (developed concurrently with, but independently of, this thesis), indicate that others have identified

the same problems: Cake supports multiple methods for determining information about a program [Kell 2009].

Finally, optimisation and transformation systems tend to be restricted to a single language. This is unrealistic in a world where almost all systems are multi-language. For example, applications for the Android system are written in Java but frequently extended with C and C++; applications for the iPhone are typically (but not necessarily) written in Objective-C, but may also make use of C and C++ libraries (these two systems are discussed in more detail below). It seems that a multi-language binary-only optimisation would provide an interesting way to explore optimisations not possible with current work.

In summary, Currawong is novel in two major ways:

- It demonstrates the novel concept of binary-only, interface-level optimisation; and
- It uses a simplified optimisation specification language capable of representing a variety of optimisations in a consistent way.

3. Background

> I am rarely happier than when spending an entire day programming my computer to perform automatically a task that it would otherwise take me a good ten seconds to do by hand.
>
> – Douglas Adams

This work grew from research work on optimal component selection in an experimental microkernel-based component system. The core ideas were then applied to a real-world system in order to demonstrate their applicability to complex systems. Consequently, reference is made throughout this document both to the research component system, named CAmkES, and to the real-world system—the Android mobile operating system. This chapter gives a brief introduction to both of these systems and introduces some motivating examples which are used throughout the thesis.

3.1. The CAmkES component system

The Component Architecture for Microkernel-based Embedded Systems, normally referred to by the rather intimidating abbreviation CAmkES [Kuz et al. 2007], is a tool for rapid development of component-based embedded systems. It facilitates the component-based software engineering (CBSE) software development technique, in which systems are modelled as a set of interacting components. In CAmkES, components may be separated from each other using hardware-mediated memory protection: an implementation restriction currently requires one component per *protection domain* (see below), which is essentially equivalent to a single component per process.

The microkernel used by CAmkES is OKL4 [Heiser et al. 2007], a modern variant of the high-performance L4 family originally described by Liedtke [Liedtke 1995]. L4 does not have an explicit process (or address space) concept, but instead maintains only the concepts of *threads* and *protection domains*, where a thread is a schedulable flow of control, and a protection domain represents all the regions of virtual address space accessible to the thread. This arrangement encourages memory sharing. In the rest of this dissertation, all three terms “thread”, “process”, and “protection domain” are used when appropriate.

Components interact using *connectors*, which are small pieces of code, usually automatically generated, which perform the system-specific operations necessary to facilitate interaction between components. This interaction is defined statically at both a component level and at a whole-system level. At the component level, possible interactions with the component are defined by one or more *interfaces*, written in a subset of CORBA

IDL [Object Management Group 2004]. At the whole-system level, all component-component interactions are specified in a CAmkES-specific *architecture definition language* (ADL).

A system description written in ADL is transformed into a bootable system by the *ADL compiler*. This compiler runs when the system is being built and generates system-specific connector code from connector templates, which describe the connector's behaviour generically. It also generates a program to initialise the system by performing kernel-specific tasks, such as creating threads and initialising shared memory regions.

Figure 3.1 shows three diagrammatic representations of CAmkES systems. These diagrams are a graphical representation of the ADL-based system description. In the figure, the system architecture labelled A represents a three-component system, where each component is represented by a large rounded rectangle. Components communicate via the FS and Block connectors, which are *functional connectors*—they simulate a function-call interface. The Client and FAT File System components are also connected via a shared memory region (shown as small squares), as are the FAT File System and Block Device components. Note that connectors are implemented using one of a number of system-level communication primitives: for example, functional connectors can be implemented using IPC if the communicating components are in separate protection domains, but may use direct function calls if the components are in the same protection domain.

In the same figure, the system architectures labelled B and C show different ways that the basic architecture can be modified using CAmkES. System architecture B shows component replacement. Here, the file system component has been changed, from FAT File System to EXT2 File System. In CAmkES, components can be interchanged as long as they support the same interfaces. System architecture C demonstrates the *interposition* pattern to adapt old components to new interfaces. Here, the client uses the OldFS interface, but the FAT Filesystem uses the FS interface. To solve the problem, a new component, named Shim, is interposed between the two, to transparently convert uses of the old interface to the new interface and vice-versa.

Component-based systems make a good testbed for optimisation systems precisely because of this well-defined separation: some classes of optimisation can be implemented simply by inserting shim components, replacing components with alternatives which implement the same interface, or by changing the code which implements connectors between components.

3.1.1. Binary CAmkES

Several features were added to CAmkES over the course of this work. The new system is named Binary CAmkES, but is usually referred to simply as CAmkES in this dissertation. The most important new feature is support for composition of binary-only modules. CAmkES requires source code for all components when building the system, because it generates system-specific C / C++ header files from a template which the components must use. Binary CAmkES solves the same problem using a linker. In this regard it was inspired by, and extends, Knit [Reid et al. 2000].

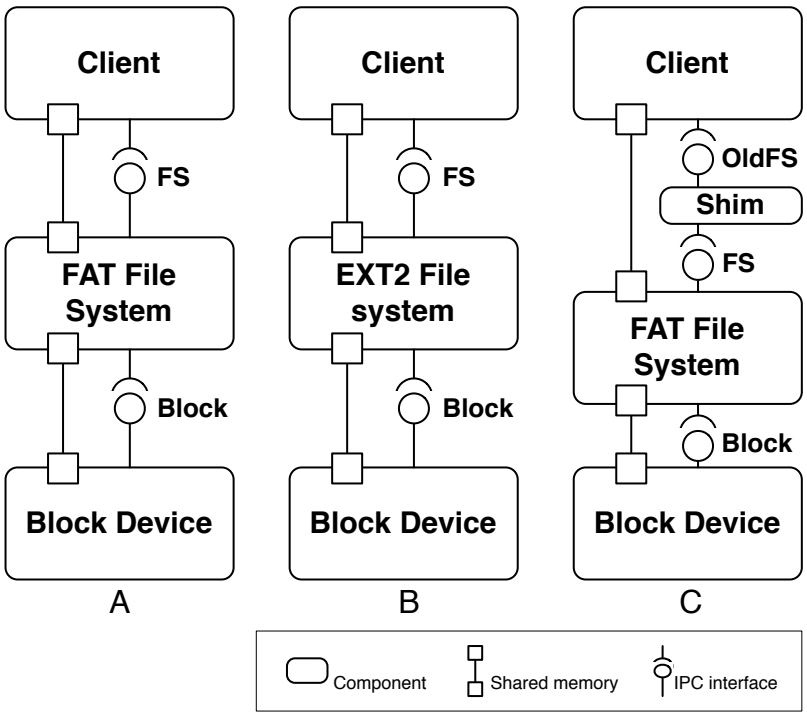

Figure 3.1. System design with CAmkES

In Binary CAmkES, each component consists of a single object file on disk. The component file is named using a naming scheme which guarantees uniqueness.

3.2. Android

Android is a complete environment for running code on mobile phones and other mobile devices. Google describes Android as [Google Inc. 2010]

> [A] software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.

Android's architecture makes it an interesting target for optimisation research:

- It is a real system: Android is based on industry-standard components such as the Linux kernel and Java, so optimisations applied to the system will be applied to code that has already been optimised using traditional techniques—there should be few "low-hanging fruit";

- It has a large API: Architecture optimisation operates at the API level, so a large API provides more opportunities for optimisation;
- It is testable: A large number of free applications are available for Android, including applications which require as much performance as possible, such as games; and
- It is componentised: Android represents a practical compromise between performance and a highly componentised system architecture.

Nobody could claim that it is a textbook example of a componentised system (for example, almost all user-level services run within a single process), but Android was nonetheless obviously influenced by component-based software engineering principles. Android supports installation of third-party software. Separate programs normally run as separate Linux processes, but can communicate with other processes using *Binder*, an Android-specific inter-process communication mechanism. Like CAmkES interfaces, Binder interfaces are well-defined (using Java as a specification language). Android's core functionality is split between system libraries (written in Java and C), the System Server, which manages access to devices, and launches applications, and a POSIX-like layer. This highly-layered approach provides multiple optimisation opportunities, although the examples shown later in this work focus on the system library-to-application interface.

Android was designed to run on a wide variety of hardware. However, the three most popular Android-based smartphones available at the time of writing, the HTC Dream, the Motorola Droid, and the Nexus One, share the following characteristics:

- Capacitative touchscreen (suited to finger-based, rather than stylus-based, input);
- Screen resolution of at least 480x320 pixels; and
- Hardware-accelerated graphics using (at least) OpenGL ES 1.1.

This de facto hardware standard has acted as a set of minimum hardware specifications for developers, and many assume that these characteristics will be present for all Android-capable devices. For example, all of the applications discussed below rely on the above screen resolution, most rely on a touchscreen, and some require OpenGL. Even relatively-small hardware details drive application design: for example, some applications assume that fingers (instead of a stylus) will be used for touchscreen input, thus assuming that the touchscreen is capacitative rather than resistive.

The Android architecture as provided by Google is shown in Figure 3.2 [Google Inc. 2010]. This diagram shows the layering described above and, in particular, emphasises applications' dependence on the Android application framework. This diagram does not, however, do a particularly good job of showing the amount of communication performed by a single application: all application input and output makes use of the System Server, which is essentially just an Android application itself (albeit one with special privileges). This communication takes place using Android's IPC mechanism, Binder. This means

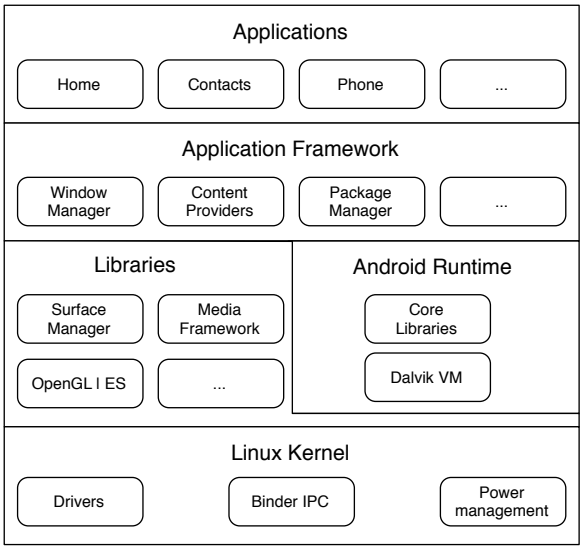

Figure 3.2. Android architecture

that, for example, every time the user touches the screen, a Java class representing the event is generated in the System Server and transmitted, via Binder, to the application. Similarly, whenever the application wishes to refresh the display, it must communicate with the System Server.

An application's interaction with the System Server is shown in Figure 3.3. The application transmits bitmaps to the System Server for display using shared memory (represented by small squares in the diagram). The application communicates via function-call-based IPC to instruct the drawing server 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. Figure 3.3 is somewhat simplified and omits many other interactions between the application and the System Server.

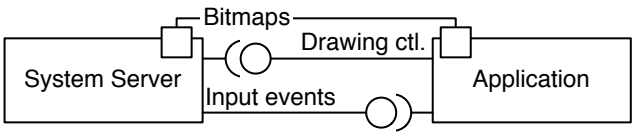

Figure 3.3. Interaction with the System Server in Android

3.2.1. Android is written in multiple languages

An important characteristic of Android is that most of the system is written in Java, but portions are written in C. In fact, all Android applications contain at least some Java, and

many are completely Java. It is very common for smartphone applications to be multi-language. For example, iPhone applications are written in a combination of Objective-C and C; and applications written for Windows Mobile 7 Series phones will run in Microsoft's Common Language Runtime, which supports many languages (the most common being VB.NET, C#, and Managed C++).

3.3. Running examples

A single running example is used to illustrate architecture optimisation in a research component system, and several Android applications are used to demonstrate some of the same techniques in Android.

3.3.1. The componentised video player

Component-system-based optimisations will be demonstrated on the video player shown in Figure 3.4. This figure is a graphical representation of the underlying CAmkES architecture description (in ADL).

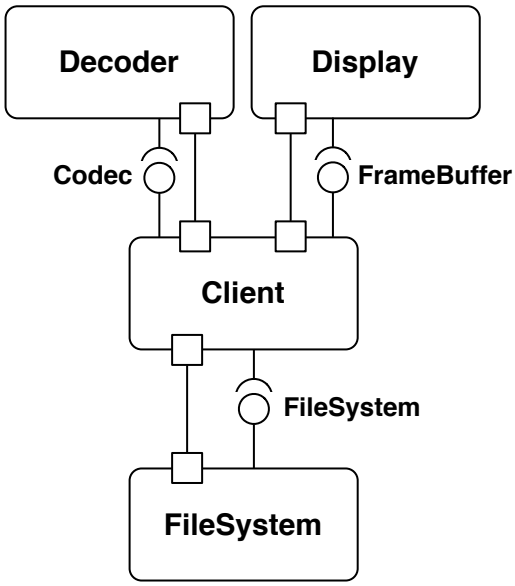

Figure 3.4. Componentised video player

In the video player, the Client component continuously requests blocks of encoded video data from the FileSystem component. This data is passed to the Decoder component. The decoded frames are eventually sent to the Display component. Each component is connected with two connectors: one remote-procedure-call connector (which provides a function-call interface); and one dataport connector (which provides a shared memory region).

The ultimate outcome of any optimisation involving this system results in a change to the ADL, resulting in a component being changed, a component being added (interposed) between two components, or a component being removed.

3.3.2. Examples in Android

Android optimisation was demonstrated using several applications written specifically to test Currawong, as well as two Android games. The following criteria were used to select the games:

- Popular: the game should be in the "popular" list in the Australian version of "Android Market", Google's online application store, as of February, 2010.
- Graphically intensive: the game should have a high rate of screen updates, i.e. it should be an action game. This requirement ensures that the application performs a large number of application-to-system interactions.
- Optimisable: the game should be able to be optimised by one of the two high-level optimisations written for this dissertation. Chapter 7 describes the two high-level optimisations in detail.

The first two games which met all the above criteria were selected from the "Popular" list of the Android Market. The games are:

- Bonsai Blast: A puzzle game, in which the player is required to connect three or more bubbles of the same colour to eliminate all bubbles on the screen.
- Space War: A top-down shoot-em-up.

Android evaluation is discussed in more detail in Chapter 7.

4. Performance anti-patterns

> Error is endlessly diversified. It has no reality but is the pure and simple creation of the mind that invents it.
>
> – Benjamin Franklin

No two software systems are the same, so, when a performance problem is encountered, it is tempting to conclude that it is unique to the system under examination. This is not necessarily the case. Often, a performance issue can be viewed as an example of well-defined class of similar problems. In this chapter, I first look at examples of architectural fixes to particular system architecture optimisation problems—in effect, instances of manually-implemented and domain-specific system software architecture optimisation. I briefly discuss the relevance of each example. I then discuss the general problems motivating this related work, to produce a set of *performance anti-patterns*. Here I use *pattern* in the sense described by Gamma et al. in "Design Patterns" [Gamma et al. 1995]—that is, a general solution to a design problem. A performance anti-pattern, then, is the opposite: a design-level cause of peformance problems.

In the final section of this chapter, I discuss potential remedies for the identified performance anti-patterns, where a *remedy* for a performance problem is either a description of how the problem may be fixed, or an implementation, in code, of that fix. The anti-patterns and their remedies form a motivation for a general-purpose architecture optimiser, the design of which is discussed in the next chapter.

4.1. Performance improvement examples

4.1.1. `sendfile()`

Many operating systems provide a function to send the contents of a file over a TCP socket (some implementations are more generic, but this is the core functionality). This function, often called `sendfile` or similar, provides highly-specialised support for Internet file servers (such as Web servers). The `sendfile` function provides a significant performance improvement over the traditional method of sending a file through a socket: one study claims an improvement of 51% [Nahum et al. 2002].

`sendfile`'s impressive performance is the result of a large number of improvements on older methods of file transmission. Figure 4.1 is a UML activity diagram showing the traditional method by which network servers send files on UNIX-like systems, using the `read()` and `write()` system calls. The file is transmitted in blocks. Sending each block requires two system calls, resulting in four context switches per transmission.

More significantly, each block is copied four times: from the disk into the kernel's file system cache, from the cache into the buffer provided by the application, from the application's buffer into a region of memory accessible to the network card, and finally onto the network itself.

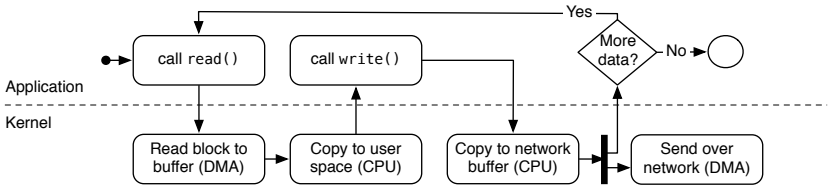

Figure 4.1. File transmission using read() and write()

By contrast, the `sendfile()` API attempts to minimise both context switches and copies. Figure 4.2 shows the behaviour of `sendfile()`. Each block of data is copied only twice, rather than four times, and only two context switches are performed for the entire file, rather than four per block.

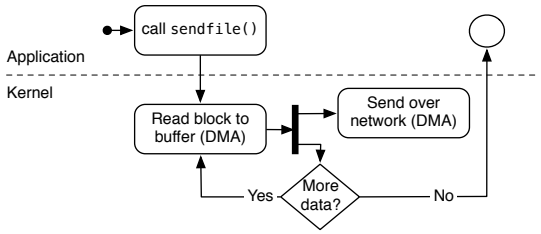

Figure 4.2. File transmission using sendfile()

On Android smartphones, which are based on Linux, `sendfile()` exists as an API for applications to use, and is thus directly relevant as a motivating optimisation.

4.1.2. API call specialisation

Several experimental systems provide a method to fine-tune their APIs on a per-application basis. This concept as applied to a traditional UNIX-like kernel was popularised by the Synthesis kernel [Pu and Massalin 1988]. Rather than expose a traditional system call API, Synthesis instead exposes an interface to produce domain-specific system calls as required by the application. The actual system call is generated using partial evaluation based on the arguments supplied to the generator.

The biggest performance improvement in Synthesis relates to file access. Figures 4.3 and 4.4 illustrate the differences between Synthesis and a traditional UNIX system—the authors use 4.3BSD in their paper. In Figure 4.3, the `open()` system call (line 2) performs a number of security checks and creates entries in various data structures to manage the

```
1 char data[BUFSIZE];
2 int fd = open("/dev/mem", O_RDONLY);
3 read(fd, data, BUFSIZE);
4 close(fd);
```

Figure 4.3. File access using the POSIX API

```
1 char data[BUFSIZE];
2 struct SIO_if *file = IO_create(FILEACCESS, "/dev/mem", FA_RDONLY);
3 read(file, data, BUFSIZE);
4 SIO_terminate(file);
```

Figure 4.4. File access using Synthesis

file, returning an integer file descriptor to the application. The `read()` call (line 3) performs the same checks, accesses data structures to discover the kernel's data structure representing the file, and calls through several layers (system call interface, virtual file system interface, concrete file system, device driver) to retrieve the data. Finally, the `close()` call (line 4) causes the kernel to deallocate any memory allocated to the file descriptor.

By contrast, the Synthesis `IO_create` system call (Figure 4.4, line 2) causes the dynamic compilation of a specialised function to perform reading from the file. In this particular case, the specialisation routines notice that the file being referenced is the `/dev/mem` device, which presents a simple byte-for-byte view into memory. Because the procedure to access data from `/dev/mem` is so simple (for example, the file-index-to-device-location translation is trivial), the resulting specialised function bypasses both the virtual file system and concrete file system layers to interact directly with the device. The resulting function results in significantly less code being executed.

This is a contrived example, as the authors themselves acknowledge, but it serves to demonstrate the potential for highly-specialised system calls. This same optimisation (implemented a completely different way) was implemented by Gabber et al. in the Pebble operating system [Bruno et al. 1999]. In Pebble, as discussed in Chapter 2, communication between protection domains takes place via *portals*, small pieces of code (and, optionally, data), which run in kernel context. Portals are written in tiny declarative language, designed in such a way that only the application invoking the portal code must trust that portal.

Figure 4.5 shows the process of opening and then reading from a file in Pebble. The `open()` call results in a call to a library function, which communicates via a portal to the file server (a separate application). The server creates several new portals in the application's portal space: one each for reading, writing, seeking in, and closing the file. The call returns a "file descriptor" which is really nothing more than the index into the portal table of the first portal in the newly-created set. When the application calls `read()`, the associated library routine uses the file descriptor as a portal index, and invokes the appropriate portal. The portal code can contain within it a pointer to the file data structure

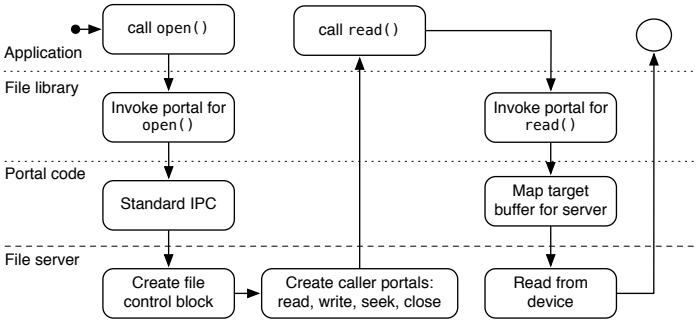

Figure 4.5. Reading from a file using Pebble portals

in the file server's address space. Thus, the creation of custom code avoids several table look-ups: the file server knows that the file data structure it is passed is correct, because the portal code is trusted.

Android does not contain an easily-identifiable example of API call specialisation. However, the CAmkES componentised video player, introduced in Section 3.3.1, contains a good example of the technique, in which standard file system read() and write() calls are specialised according to the requirements of the player so that memory can be shared between components. Section 5.2.3 contains a full treatment.

4.1.3. Integrated Layer Processing

Several authors have recognised the inefficiencies resulting from layered protocols. A common occurrence is that data are traversed more than once, resulting in suboptimal stack performance. For example, a network stack may copy data in one function, and compute a checksum in another (lower-level) function. Both these operations require examining every byte of the source data.

One remedy is to attempt to perform operations which would normally be performed in different functions, at the same time. This approach is known as Integrated Layer Processing, or ILP [Smith 1990, Abbott and Peterson 1993, Clark et al. 1989]. Abbott and Peterson provide a good treatment of the problem [Abbott and Peterson 1993]. They describe a generalised data-processing function which they call a *word filter*. A word filter accepts a single machine word as input, and outputs zero or more machine words. Multi-stage operations on data can thus be described using a chain of word filters: a relatively simple example is computing a checksum on a data packet while simultaneously copying it. Word filters are assembled at compile time and, for speed reasons, are written in a similar way to C-style macros. The authors achieve significant (17% to 36%) improvements by implementing their ideas.

Optimisation opportunities of this type occur in image processing. It is common to post-process images in some way in order to change their appearance. Simple transformations involve changing the image's contrast or brightness—such transformations involve applying a mathematical formula to each pixel in the image. More complex transforma-

tions are applied to groups of adjacent pixels. For example, *convolution filters* apply a 2D convolution in the spatial domain to an image by performing a 2D convolution operation on each pixel in the image, taking its neighbours into account. Common image convolutions include blurring, sharpening, and edge detection.

Figure 4.6 shows a filter chain applied to the componentised video player. Here, each filter acts as a framebuffer, accepting image data, processing it, and forwarding it to the next filter in the chain. This very straightforward arrangement results in each filter processing an entire image. This results in a streaming-data cache usage pattern, in which data for the first bytes of an image, as used by the first filter, has already left the cache by the time it is to be re-used for the second filter. This is the anti-pattern addressed by ILP.

Figure 4.7 shows a possible ILP-centric optimisation of the same filter chain. In this optimisation, an ILP-aware FilterControl component is interposed between the filter, Client, and Display components. Because it is connected to all filters, FilterControl can instruct each filter to process only a small portion of the framebuffer at one time. This means that the small portion of framebuffer loaded into the data cache for the first filter can also be used for the second filter, thus avoiding the streaming-cache antipattern.

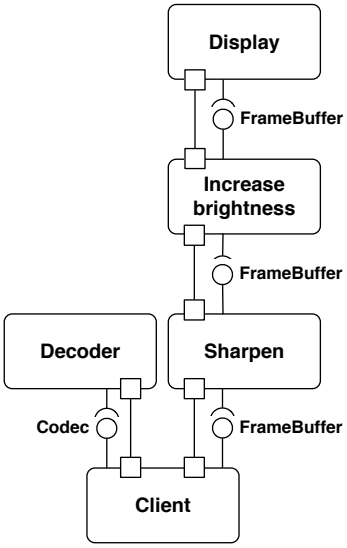

Figure 4.6. The componentised video player with a chain of image filters between client and display

4.1.4. Protocol header optimisation

In some multi-module systems, particularly networking stacks, a common pattern is for a module to accept a packet, add some data to that packet, and then pass the packet on to a lower-level function. A particularly common case is when headers are added to the packet. Figure 4.8 shows a typical TCP/IP packet ready to be transmitted over an Ethernet network. The packet contains three headers, each of which may have have been added

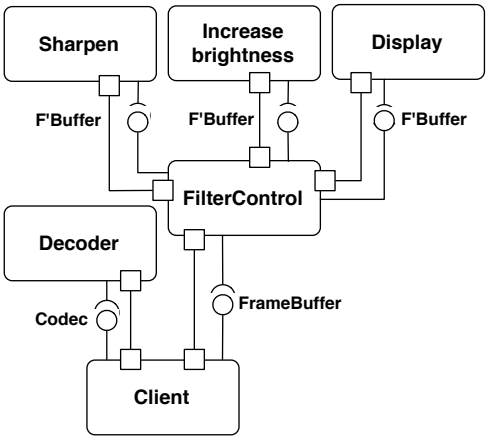

Figure 4.7. The componentised video player with a chain of image filters with data access mediated by a controller

by separate sub-systems. A naïve approach to adding these headers is to copy the packet multiple times, allocating new storage space each time. This is obviously inefficient. Another alternative is to over-allocate storage space for the packet, leaving empty space at the beginning for extra headers. This is the approach traditionally taken by Linux, with its `sk_buff` structure [Miller 2010].

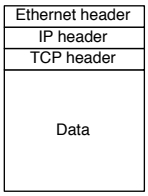

Figure 4.8. A network packet with multiple headers

A more generic approach is to represent the packet as a list of contiguous regions. When the packet is transmitted, the regions are either copied so that they are literally contiguous, or, if the network hardware supports *scatter-gather* I/O, the pieces are sent to the hardware as-is, relying on the scatter-gather hardware to assemble the pieces into a complete packet. This technique became popular with BSD 4.3's *mbufs* [Leffler and McKusick 1989]. Several other networking stacks support a similar structure. For example, the light-weight IP stack (LWIP [Dunkels 2001]) uses *pbufs*, which are based heavily on mbufs, as its core packet structure.

4.1.5. FBufs: high-bandwidth transfer

It is often desirable to transfer a large amount of data between protection domains. These sorts of communications occur more frequently in microkernel-based systems—a multi-server network stack is the canonical example—but may also occur in monolithic systems, particularly in systems dealing with video and audio data.

The traditional way to pass data between protection domains is to copy it. This forms the basis of UNIX pipes and fifos, and is also the only supported data-copying mechanism in early microkernels. Copying becomes inefficient when significant amounts of data must be transferred. Pipes are also rather difficult to customise—the buffer size of the pipe, in particular, can't generally be adjusted. A too-small buffer results in unnecessary context switches between data producer and data consumer; a too-large buffer wastes memory. Thus pipes pose problems in terms of both data flow and control flow: data flow, because of the imposed copy; and control flow, because of the lack of control over the extent and frequency of context switches between communicating processes.

The solution is to use shared memory—in particular, a circular buffer. This is a more complicated data structure and its use raises a number of design issues. Which process allocates the memory? Which process "owns" the buffer (and must, presumably, free it)? What if two processes attempt to access the same data at the same time? What if more than two processes wish to access the same data? Druschel and Peterson proposed FBufs to address these problems [Druschel and Peterson 1993]. FBufs has been reimplemented and extended upon multiple times [Pai et al. 2000, Mosberger and Peterson 1996, Chu 1996, de Bruijn and Bos 2008], and implementations exist in entirely user-level applications for non-research operating systems (for example, the `vo` video output library, part of the MPlayer video player, uses an implementation of the technique [MPlayer authors 2010]). Druschel and Peterson demonstrate an eight-fold performance increase over data copying in their paper. Interestingly, they propose a number of optimisations to make FBufs even faster, at the expense of a certain amount of flexibility (for example, they require that FBufs be mapped at the same virtual address in all participating processes). These optimisations produce a further twelve-fold increase over their original FBufs implementation, for a total one hundred times improvement over copying.

FBufs builds on many other concepts, including the BSD mbufs described above.

4.2. Summary of performance anti-patterns

Each of the performance problems described above is caused by at least one of the following five performance anti-patterns. Frequently, a particular performance problem will involve more than one such anti-pattern.

1. Context switching
2. Copying
3. Overly-generic or inflexible APIs

4. Unsuitable data structures

5. Reprocessing data

Each anti-pattern is discussed with reference to the network video player described in Chapter 3. For this discussion, it is necessary to extend the original video player diagram, Figure 3.4, to explicitly show each component's protection domain. The new video player is shown in Figure 4.9. In this figure, the dashed regions represent protection domains. In this version of the video player componentised system each component resides in one, and only one, protection domain.

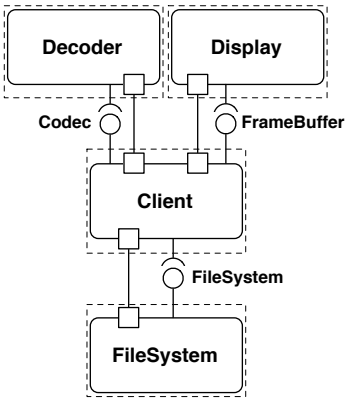

Figure 4.9. Componentised video player showing protection domains

4.2.1. Context switching

Both `sendfile()` and FBufs reduce context switch frequency. Context switches reduce application performance both directly and indirectly. The direct cost is due to the kernel code which must be run to implement the switch, and is usually relatively small. The indirect cost is due to the cache impact of the switch: if the total working set size of all programs exceeds the cache size, programs will suffer a performance hit as code and data which was evicted from the cache are re-loaded. On some architectures the entire cache is flushed during a context switch, causing performance impact potentially 100 times greater than the direct cost. The best modern coverage of this phenomenon, for IA-32 architectures, is given by Li et al. [Li et al. 2007].

However, when processes being context-switched are interacting, we can expect a less pessimistic result. For example, consider the interaction between the Client and Decoder components in Figure 4.9. In this system, a context switch occurs from Client to Decoder whenever the client requests that some data be decoded. When the decoded data are ready, another context switch occurs, from Decoder to Client. However, because Client relies on Decoder to perform a service, the code in Decoder would have to be run even if Decoder were in the same protection domain as Client—so there is little unnecessary code cache

pollution. Because the same code is being run, the impact on the data cache will also be minimal (assuming the two components use shared memory when running in separate protection domains). Therefore we can expect that the context-switching cost between two interacting components is, in a well-designed system, mostly due to the direct cost of a context switch.

In most cases, the direct cost of a context switch is not significant [FitzRoy-Dale and Kuz 2009, Li et al. 2007]. Nonetheless, a large amount of context switching tends to be symptomatic of other performance issues. The general solution to the problem of excessive context switching due to IPC between two components is to *decouple* the control flow between the two components, so that, to take an example from the componentised video player, the Decoder and the Client can both work independently.

4.2.2. Copying

`sendfile()`, FBufs, and protocol header optimisation all reduce the amount of copying performed in the system. Unnecessary data copying has an obvious performance impact: it consumes processor cycles and fills data caches. It also has some impact on other caches, such as instruction caches and the translation lookaside buffer, due to the execution of the copying code. There are three options to avoid copying:

1. Remove one component's involvement in the process. The `sendfile()` optimisation essentially relieves the application of any involvement in the task of sending a file. This automatically has the effect of removing extraneous copies, because the API responsible for the copying (i.e. the API comprising the POSIX `read()` and `write()` functions) is no longer required.

2. Use shared memory. FBufs takes this approach. Switching to shared memory means addressing considerations such as memory ownership and concurrent access.

3. Use better data structures. This is discussed in Section 4.2.4.

4.2.3. Overly-generic or inflexible APIs

Overly-generic APIs can result in multiple performance issues. Both Synthesis and Pebble aim to eliminate one of the most common: parameter validation and data lookup. They both achieve this by pre-validating the parameters and performing any necessary look-ups ahead of time. For example, Synthesis custom-compiles a `read` function with file-specific information compiled into the function.

Both Synthesis and Pebble achieve their performance goals by providing a single, application-customised API. An alternative is simply to provide a large variety of APIs for the same task, each with differing performance benefits. The significant disadvantages to this approach were covered in Chapter 1. To summarise: multiple similar APIs makes application development more confusing for the developer, and makes supporting the API more difficult for the API writer.

4.2.4. Unsuitable data structures

Unsuitable data structures result in excess data manipulation. In the protocol header optimisation example, this resulted in data copying when a header was added to a packet. The network stack was simply using the wrong data structure for the job, so the data structure had to be changed. The two alternatives presented (`sk_buffs` with additional space at the beginning, or `mbufs`, which can be chained) both address the problem by using a more capable data structure.

4.2.5. Reprocessing data

Reprocessing of data is similar to data copying in terms of performance loss, but without the associated cache issues: the data are not copied, but portions of data are accessed multiple times, unnecessarily. The Integrated Layer Processing example in Section 4.1.3 avoids data reprocessing through architectural means, by performing multiple operations for each word when processing a network packet.

4.3. Remedies

Once a performance anti-pattern has been identified, an appropriate remedy can be applied. Remedies describe the actual changes that must be made to the system in order to effect an optimisation.

It is sometimes possible to optimise a component-based system without changing any component's implementation. Instead, changes are only made to the connections between components. This style of *inter-component optimisation* bypasses the potentially-complex task of verifying component source code, because the source code is not modified. This feature also makes inter-component optimisations easier to reason about.

There is a small and well-defined set of inter-component modifications. An inter-component modification changes the protection domains in which components reside; or modifies the component graph by adding, removing, or changing components. Only some of these modifications are optimisations; others may be used to enforce security or correctness requirements. For example, it is sometimes useful to place two components into separate protection domains. A typical reason to do this is to isolate a sensitive component from untrusted code. However, communication between the two components must now cross a protection-domain boundary, reducing performance.

In this section, the following set of component-system modifications are discussed as remedies:

1. Combine protection domains
2. Replace component or library
3. Interpose component

Inter-component optimisations are attractively simple, but not all optimisations can be expressed this way. Sometimes architecture optimisations require modifying a component's implementation. Component code is significantly more complicated than a component architecture, so correctly modifying code is a more significant challenge than correctly modifying a component architecture. This dissertation does not attempt to provide a general code-verifcation system. Instead, it focuses on a single relatively-safe form of code modification:

4. Modify component-to-component APIs

The important distinction between the first three remedies and this one is that the first three remedies are *structural remedies*: they do not rely on modification to component code, but instead focus on recognising and modifying the connections between components. The fourth remedy is a *code remedy*: it requires analysis, and, possibly, modification of source code. The details of this fourth intra-component optimisation, and a discussion of its relatively safety, are provided in later chapters.

In general, the modifications required to implement a remedy are small. However, as described in the introduction, some types of optimisations must be verified. The exact nature of the verification depends on both the intent of the optimisation and the particular remedy being applied.

In this chapter, I distinguish between verification that requires knowledge of data in the system, *data-sensitive verification*, and verification that does not require knowledge of data, *data-insensitive verification*. An optimisation that only requires knowing that an application calls a particular API function can be verified using data-insensitive verification, because no knowledge of the API parameters, or the rest of the program's state, is required. If, however, performing the optimisation safely requires knowing, for example, that the first parameter is always a non-null pointer, then data-sensitive verification is required: the optimiser must prove a fact about the parameter before optimisation can take place. This distinction is covered in more detail in Chapter 5.

Each remedy is described below. For each description, an example of a system optimisation making use of that remedy is given. The verification conditions for the optimisation are also described, informally. A complete formalisation is given in the next chapter.

For the sake of clarity, the supplied examples are illustrated with reference to the running example of the componentised video player, shown with protection domains in Figure 4.9. However, each remedy's application in a real-world, less-idealised system, such as Android, is also discussed.

It is important to note that there is not a one-to-one correlation between optimisation remedies and performance anti-patterns: the examples of performance anti-patterns shown in this chapter could be addressed in various different ways. Currawong is, however, limited to the four remedies described above.

4.3.1. Remedy 1: Combining protection domains

A simple way to improve performance in a system involving two isolated components is to place them both into the same protection domain.

Figure 4.10 shows the video player with the video decoder (Decoder component) and the user interface (Client component) occupying the same protection domain. In Android or another Unix-based system, this scenario is analogous to replacing a separate decoding process with a shared library loaded by the Client. Combining protection domains is a way to reduce context switching and copying.

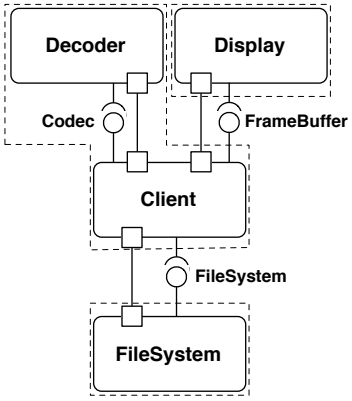

Figure 4.10. Video player: client and decoder occupy the same protection domain

Combining protection domains is a very generic and potentially highly-dangerous remedy. The hardware-enforced memory protection boundary between the components is now removed, and a buggy or malicious component can now crash its communication partner by overwriting its data. For example, the Decoder component could overwrite the Player component's private data and cause it to crash or behave unexpectedly. Along similar lines, if the Player component had specific privileges (such as being able to communicate with the Display component), and if these privileges are managed on a per-protection-domain basis, the Decoder component could now also make use of those privileges. A component written with malicious intent could exploit any of these vulnerabilities for its own gain.

In general, if we wish to ensure that the new system has exactly the same safety properties as the old, the optimiser must guarantee complete noninterference between components in the new protection domain: a complex path-dependent optimisation. However, in some cases shortcuts can be taken. For example, if one of the combined components is written in a high-level language and is run on a virtual machine (such as Java), the system optimiser can choose to trust the correctness of the virtual machine running the component, and focus instead on verifying the behaviour of the high-level-language code.

4.3.2. Remedy 2: Replacing components or libraries

Another simple way to improve system performance is to replace a component (in a componentised system) or a library (in a noncomponentised system) with a different one which performs the same function, but is optimised in some desirable way. The extent to which this can be done depends on the structure of the existing code. Consider the

protocol header optimisation case described in Section 4.1.4. This optimisation removed a copying requirement on network buffers by replacing the data structure representing the buffers. If all operations on the buffers were performed through a small functional API, then replacing this API is a simple matter. If, however, operations on the buffers are performed directly, then any non-source-code-level optimisation may be infeasible. Component replacement is a very generic remedy and could conceivably be used to reduce the impact of any of the performance anti-patterns listed. However, it is particularly well-suited to handling of overly-generic or inflexible APIs (anti-pattern 3) or unsuitable data structures (anti-pattern 4). Another simple example of this sort of optimisation is adding hardware-specific support to a system, replacing the Decoder component in the video player example with one which supports hardware-accelerated decoding, for example.

If the functionality of the replaced component is indeed exactly the same as that of the new component, then the verification conditions are very simple: only path-independent analysis is required. If, however, the new component imposes additional non-functional requirements on its clients, those requirements must be checked before the component can be said to be safe. For example, suppose the original Decoder and Client components communicated by copying data, but the new Decoder component uses shared memory. In this case, the Decoder may have additional requirements of Client—such as that of mutual exclusivity of access to the shared region. In this case, the optimiser performs the data-sensitive verification that the Client component does not access the shared region when it is in use by the Decoder component.

4.3.3. Remedy 3: Component interposition

Component interposition involves adding another component to the component graph, connecting two or more components to the new component, and then placing the new component in the protection domain of an existing component. In terms of high-level modifications to the system, it is thus more complex than the previous two optimisations (combining protection domains, and replacing components). However, it is conceptually simple: the new component acts as an interface between two or more components. An example of component interposition is shown in Figure 4.11.

The figure illustrates the following optimisation: some videos played by the video player do not require decoding, in the sense that they are already in a form that may be sent directly to the output hardware. In this case, it may be acceptable to include a very simple "null" decoder in the protection domain of the Client component, which forwards data to the Decoder component in another protection domain in the cases where the data actually does require decoding. If the video does not require decoding, the Null component recognises this fact and simply returns the original data to the client: no protection-domain crossing is required.

The component interposition pattern is particularly well-suited to the "Overly-generic or inflexible APIs" anti-pattern (number 3).

The verification requirements for component interposition are simple (path-independent) if the newly-interposed component is *trusted*, that is, if the optimiser user decides that the interposed component will not interfere negatively with other compo-

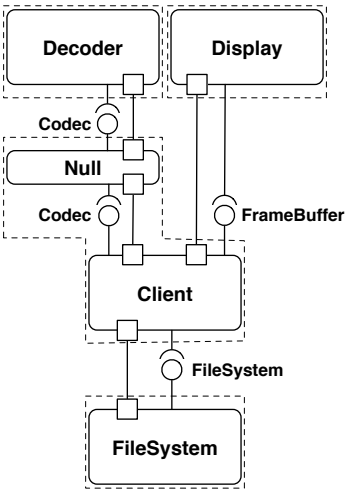

Figure 4.11. Component interposition

nents in its protection domain. This scenario is not as unlikely as it might be for the general-case protection-domain-combining remedy described above. This is because it is expected that interposed components be used to translate from one API to another (remedying anti-pattern 3, "Inflexibe APIs") or otherwise perform some small, well-contained task. A smaller amount of code translates directly to decreased likelihood of bugs in that code.

4.3.4. Remedy 4: Modifying component-to-component APIs

This remedy involves modifying the way components communicate. In our component system, this is equivalent to changing a *component connector*. In Android, this is instead equivalent to causing the application to make different calls to the Android API package.

The simplest way to perform this modification is to use component interposition. That is, a component supporting both the old and new API is *interposed* between two communicating components. The new component translates API calls from the old API to the new, and back again. This scenario is illustrated in Figure 4.12 (interposition is also shown in Figure 3.1, system C, in the previous chapter). To accomplish interface interposition in Android, one would write a class that mimicked the old API, and rewrite the application code to make use of the new class.

Component interposition to perform API rewriting is tempting because it requires minimal modification to the components themselves and, as described above, can be verified using efficient data-insensitive techniques, as long as the interposed component is trusted.

Interposition is, not, however, the most efficient solution. At the very least, additional function calls (to the interposed code) are required. More dramatically, simulating the old API in terms of the new API simply may not be the best solution. For example, the system in Figure 4.12 shows an interposed component, Shim, translating from the

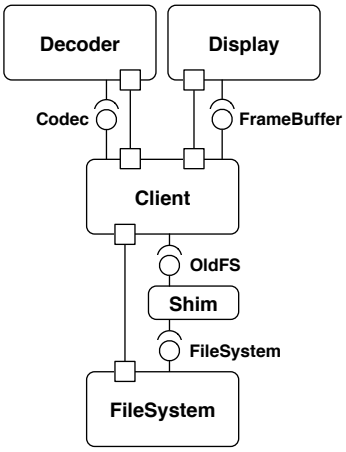

Figure 4.12. API modification via interposition

FileSystem API to the OldFS API. Imagine that the FileSystem API supports memory mapping of files, but the OldFS API does not. If the Client could take advantage of memory-mapped files, it may perform more efficiently. However, it cannot make use of memory mapping as long as it continues to use the OldFS API.

The more efficient alternative is for some portions of the Client code to support the new API. Doing so may be quite simple—the API evolution may simply involve additional parameters, or renamed functions. In the former case, the code modifications involved could simply supply default values. In the latter case, even less work is required: the function names could be identified within the code, and simply renamed in-place (this is, to some degree, a simplification. More detail is given in Chapter 6). In these cases, the verification requirements remain simple and data-insensitive.

However, more complicated API modification may require more complex, data-sensitive verification. For example, the memory mapping interface described above may only be supported if the client does not attempt to modify the memory-mapped file. Verifying this behaviour requires data-sensitive analysis.

4.4. Conclusions

This chapter analysed a large and wide-ranging set of optimisations. These were then generalised into a small set of general performance anti-patterns. The anti-patterns served as motivation for a number of optimisation remedies. For many optimisations, the actual system modification involved is relatively small—changing a component's protection domain, for example—but the verification effort to ensure the optimisation has not introduced bugs into the system may be non-trivial.

5. Design

> Before you start [designing], it is a good idea to stop and think about what your computer can and cannot do.
>
> – Computer Spacegames [Isaaman et al. 1982]

This chapter describes the design of Currawong, a tool to perform architecture optimisation. Given an application and a description of the optimisation, Currawong does two things: it finds suitable locations to apply an optimisation within the application, and then it applies the described optimisation at those locations.

There is no point in an optimisation system which cannot perform useful optimisations. In *Compilers: Principles, Techniques, and Tools* [Aho et al. 1986], Aho, Sethi, and Ullman identify three characteristics of optimisations:

1. They must preserve the meaning of programs, in the sense that the optimised program must give the same output for all inputs as the unoptimised program;

2. They must, on average, speed up programs by a measurable amount; and

3. They must be worth the effort, in the simple sense that the benefit gained from the optimisations offsets the time invested in implementing them.

Adding domain-specific optimisations to general-purpose compilers fails criterion 3 above: it is not worth the effort. Robinson makes a convincing economic argument: it is simply not economically viable for compiler writers to spend time writing and testing optimisations that only benefit a small percentage of their customers [Robison 2001]. He describes the general problem with the memorable statement that "Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers".

Architectural optimisation's answer to this economics argument follows the Active Libraries model discussed in Chapter 2: give the task of writing appropriate optimisations to the authors of the optimisable component. This strategy encourages those who will directly benefit from the optimised code to write their own optimisations, and thus directly addresses Robinson's economic argument. However, the more general concern remains—if architecture optimisations are too difficult to write, or too difficult to apply, they will not be used. A goal of Currawong's design, then, was to make optimisation specifications both easy to write, and easy to understand.

5.1. Currawong overview

Currawong locates anti-patterns, and applies remedies. Specifically, Currawong locates the anti-patterns summarised in Section 4.2. Once an anti-pattern has been found, Currawong then applies one of the remedies described in Section 4.3.

Two additional criteria guided Currawong's design: ability to work without application source code, and support for multiple languages. In Chapter 1, I argued that architecture optimisation is most useful when the architecture optimiser does not require source code for the application being optimised, because the optimisation can then be used even if the application developer is unwilling, or unable, to apply it. Support for binary-only optimisation is, therefore, a Currawong design criterion.

Although many of the transformation systems described in Chapter 2 are capable of non-trivial static analysis, most do not optimise binary-only systems. Even fewer recognise multiple implementation languages. This is not a failing of the related work, but it reflects a different focus: either on enriching a source-level application programming interface (API) for application programmers (as championed by the active libraries approach), or on helping programmers keep up-to-date with minor API changes (as used in refactoring-based approaches). In Chapter 1 I noted that in many cases the programmer will not, in fact, benefit significantly from optimising his or her application, but the end user will. Currawong, therefore, differs from the active-library and refactoring approaches in that it attempts to support the system designer or end user rather than the programmer.

In Section 3.2.1, I argued that it was common for applications to be be written in more than one programming language and that, consequently, architecture optimisation should support multiple languages also. Using a single architecture optimisation implementation capable of supporting multiple languages means that optimisations which span multiple languages could, in theory, be implemented. For example, one could apply an optimisation to an Android application, written in Java, which makes use of an extension library written in C. Unfortunately, time constraints meant that no examples of this optimisation type could be included in this dissertation. Currawong supports multiple languages, but cross-language optimisation was not implemented.

In summary, Currawong should:

1. Recognise anti-patterns;
2. Rewrite applications to apply remedies;
3. Perform its work without the target's source code; and
4. Support multiple languages.

Architecture optimisation is a program transformation technique. Therefore Currawong must, at the very least, behave like a program transformer. That is, it should accept a program as input; perform some transformation on that program; and produce a transformed program as output. Currawong's high-level design extends this simple model by adopting features from the program transformers surveyed in Chapter 2.

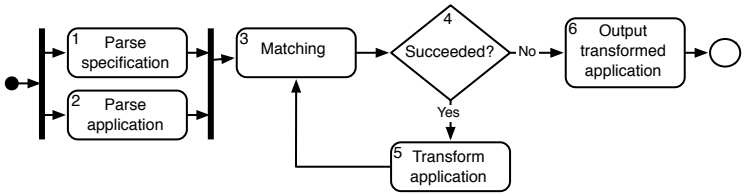

Figure 5.1. Currawong overview

The major components of Currawong are shown in Figure 5.1. The input to Currawong is an application and an optimisation specification. The output is an application that has had the optimisation described by the specification applied to it as many times as possible. It may not have been possible to apply the optimisation even once, in which case the output of Currawong is an application identical to the one supplied as input.

Currawong consists of four major stages. Each of these stages corresponds with one or two numbered steps in Figure 5.1.

Parsing (steps 1 and 2 in Figure 5.1): The application is parsed and Currawong generates an abstraction. Currawong also loads and parses the optimisation specification. The result of this process is a parsed optimisation along with an application representation.

Matching (3): The optimisation specification is used in conjunction with the application abstraction to locate a portion of the application code to optimise. The optimisation specifies one or more *match critera* which determine whether or not applying an optimisation at a given point is the correct thing to do. Currawong verifies that these match criteria actually apply.

Transformation (4 and 5): Currawong transforms the application according to the optimisation specification. The matching and transformation steps repeat until no more optimisations can be applied.

Output (6): Finally, Currawong writes the new application to disk, if any transformations were applied.

The optimisation examples in this chapter focus on performing optimisations for Java and C. These are the two languages supported by the implementation of Currawong as described in the next chapter (Chapter 6). They were selected because they are the major languages used for Android applications and for CAmkES applications, respectively.

5.1.1. Use of source code

A key design goal for Currawong is that it not rely on the target application's source code being available when optimising. The major advantage of this approach is that optimisation can be decoupled from program development—this advantage is described

in detail above. However, optimisers typically rely on information only present in the source code in order to perform safe code optimisation. What is the impact, if any, upon the expressive power of Currawong optimisations if source code is not available? Further, what about source code for other parts of the system, such as the system libraries? To understand the trade-offs, it is necessary to understand the information that is gained or lost when source code is compiled and linked. The discussion below focuses on C code first, and then discusses the differences between C and Java.

```
#define EFFECT_NONE 0
#define EFFECT_BLUR 1
#define EFFECT_SHARPEN 2

struct video_frame {
     int depth;
     int length;
     uint8_t *data;
};

struct video_renderer {
     int token;
};

int return_error(void);
int send_to_frame_buffer(struct video_frame
*data);

int render(struct video_renderer *self, struct
video_frame *data, int effect)
{
     switch(effect) {
     case EFFECT_NONE:
          break;
     case EFFECT_BLUR:
     render_blur(self, data);
          break;
     case EFFECT_SHARPEN:
          render_sharpen(self, data);
          break;
     default:
          return -1;
     }

     send_to_frame_buffer(data);

     return 0;
}
```

Figure 5.2. A small component (bold portions represent information preserved by the compiler)

A lot of information is discarded during the compilation process. Specifically, type information, control flow, language features, and meta-information is lost. The most important loss for programs like Currawong is that of type information. Figure 5.2 shows

a simple component to apply visual effects to a frame of video. The bolded portions of code in that figure represent information preserved after compilation. As the figure shows, all type information is discarded: both data types, such as the `video_frame` structure, and function type information, such as the parameters and return type of the `render` function.

Types are not the only detail discarded by the compiler. Language features disappear as well. In C, pre-processor macros are fully expanded, with the result that symbolic names (such as `EFFECT_BLUR` in Figure 5.2) are replaced with numbers. Control flow and program structure is obscured, either due to optimisation passes in the compiler, or simply as a result of reimplementing a high-level feature in binary code. In this example, the `switch` statement is replaced by a series of comparisons and jumps, and the `send_to_frame_buffer` call is duplicated. The compiler may choose to inline specific functions, obscuring the program structure.

Program meta-information is also discarded by the compiler. Variable names are removed, as are the names of C functions which do not have external linkage. Comments are removed, and the arrangement of code into files and directories is obscured.

However, the compilation and linking process also adds information. The most important gain for Currawong is that the overall program structure is revealed. It is difficult, in general, to determine exactly which functions are used within a program from its source code, without simulating the entire compilation and linking process. The problem can be appreciated by considering a program which implements multiple versions of the same function, with the linked version being selected by arguments passed to `make`. Using the application binary eliminates a tedious and error-prone stage required of source-based optimisers.

Additionally, timing information is more accessible in a binary than it is from source code: it is possible to compute best- and worst-case execution times from binary code. Doing the same thing using source code is much harder, as one must essentially reimplement the compiler so as to know the number of CPU clock cycles that each high-level instruction would take to execute.

Two properties unique to system software architecture optimisation mitigate the impact of information discarded by the compilation and linking process. The first property is that Currawong optimises around API boundaries. In terms of a compiled application, this amounts to performing optimisations around function calls made by an application, often to functions in an external library. Function calls to external library code are usually easy to detect in binary code (a well-known exception to this rule is discussed below). Furthermore, function calls to external libraries must preserve the name of the called function in the application binary, for linking purposes. Consequently, Currawong can identify both the name of the API function being called, and the point in code at which it is called.

The other property which improves Currawong's optimisation power is related to the expected author of high-level optimisations. For system software architecture optimisation, the expected optimisation writer is not the author of the optimisation tool, but is instead the system designer, hardware manufacturer, or other person familiar with the particular API being optimised. Because the optimisation writer is familiar with the API, she may bring domain-specific knowledge of that API to Currawong when writing optimisa-

tions. This knowledge comes both directly, by including parts of the API in a Currawong optimisation specification, and indirectly, through knowledge of the behaviour of a particular API.

5.1.2. Multiple-API limitations

As described above, Currawong does not infer optimisations by itself. Instead, it uses optimisation descriptions provided by an optimisation writer—either the system designer, or the author of an API for a set of optimisable components. This requirement extends the expressive power of Currawong significantly, because optimisations can make use of domain-specific knowledge known to the optimisation writer, but not easily discoverable from the code. However, reliance on an external source for optimisation specification makes some categories of optimisation infeasible.

To appreciate the limitations of the approach, consider the componentised video player first shown in Figure 3.4. This system comprises five components: decoder, display, client, and file system. In this system, data flows from the file system, to the client, to the decoder, and finally to the display. It may be useful to implement a whole-system optimisation that involves multiple components. For example, the decoder component may wish to modify the rate at which it receives data from the file system. However, doing so involves optimising two interfaces: the first between decoder and client, and the second between client and file system. At this point, we have an optimisation problem that involves three components. The optimisation writer would have to have knowledge of both the decoder-to-client and the client-to-file-system interfaces in order to write an optimisation specification. This is not always realistic. In general, optimisations which span multiple interfaces are not amenable to the single-external-author approach chosen by Currawong. Overcoming this multiple-API limitation in a reusable way is an interesting but difficult problem, and is outside the scope of this dissertation.

Nonetheless, the multiple-API problem is not necessarily as bad as it may appear. In some systems it may not even arise, because the only relevant API is known in its entirety to the optimiser writer. This is the case for Android-based phones, where a large API is maintained by a single vendor. In systems where it does arise, some optimisations can still be performed by taking advantage of the concept of connectors. The advantage of connectors with respect to optimisation was described in Chapter 3. In this context, connectors essentially provide a single system-wide API: as long as components communicate using known connectors, it doesn't matter if the optimisation writer is not familiar with the particular components.

5.1.3. A domain-specific programming language

Most of Currawong's processes are performed as a result of evaluating an optimisation specification. The optimisation specification determines which parts of an application should be examined, how they should be checked, and how the application should be transformed. Currawong provides a declarative programming language: optimisations

written for Currawong make use of an API provided by Currawong, concerned with application input, checking, transformation, and output.

Currawong's API must provide everything that an optimisation specification may require to analyse and transform an application. This chapter discusses the design in object-oriented terms. The major portion of the API is provided by a single object, the Application object, which is made available to the optimisation specification. This object provides a representation of the application being optimised, supports checking of an optimisation (by providing support for recognising anti-patterns), and is capable of transforming the application.

The rest of this chapter discusses the individual portions of the Application object: code representation (Section 5.3), matching and checking (Section 5.4), and transformation (Section 5.5). After the API has been established, the specification language design is motivated (Section 5.6) and described (Section 5.7).

5.2. Motivating examples

To give context to the design decisions made in this chapter, five complete examples are presented below. Each example demonstrates an anti-pattern and remedy from the set introduced in Chapter 4. These examples are revisited in more detail in the evaluation in Chapter 7

The first three examples demonstrate architecture optimisation of compiled C code and refer to the componentised video player example for the CAmkES demonstration component system, described in Figure 3.4.

The remaining examples demonstrate architecture optimisation of byte-compiled Java code and refer to the Android platform.

5.2.1. Example 1: CAmkES same-domain decoder

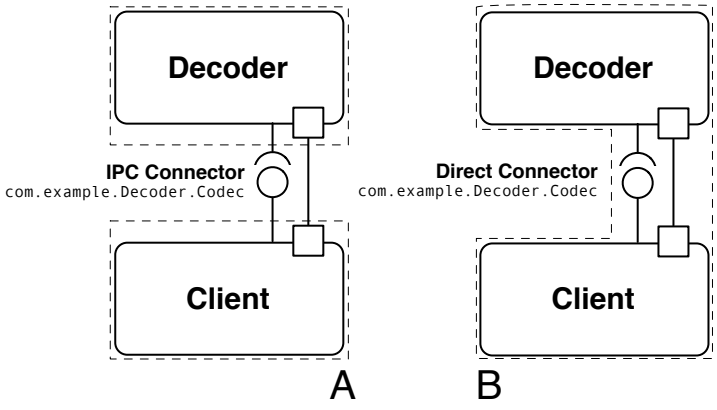

Figure 5.3. Protection domain merging in the componentised video player

This domain-specific componentised video player optimisation moves the Decoder component into the same protection domain as the Client component. This puts Decoder and Client into the same address space, in monolithic operating system terms, and has the effect of reducing the communication overhead between the two components. Figure 5.3 shows the process: the original system portion is shown in section A of the figure, and the rewritten system is shown in section B. In these diagrams, as per previous examples, large, rounded rectangles represent components; a dashed outline represents a protection domain; small squares represent shared memory; and the circle and semicircle icon represents communication via a functional interface.

In order to perform this optimisation, Currawong must identify the two components (Decoder and Client), verify that the components are currently not in the same protection domain, and then place them both in the same protection domain. Communicating components in the same protection domain will probably use a different connector (the CAmkES feature allowing component system designers to explicity state the method of communication between components) compared with communicating components in separate protecton domains, so Currawong must take this into account when changing protection domains. In summary, the optimisation must:

1. Identify components by name;
2. Identify protection domains of components;
3. Add and remove components from protection domains; and
4. Ensure that an appropriate connector is used.

5.2.2. Example 2: CAmkES RGB conversion

Component replacement was identified in Chapter 4 as a general-purpose remedy for performance problems. For this example, we imagine that the video display hardware used by the video player supports two formats for display of information: RGB, a sixteen-bits-per-pixel format in which five bits represent the red component of a pixel, six bits represent the green component, and the final five represent blue; and YUV, an eight-bits-per-pixel format in which four bits represent the overall brightness of the pixel, and the remaining four bits represent the pixel's colour.

In this system, the native image format coming from the Decoder component is YUV. Before being transmitted to the client, the image is translated to RGB format. Performing such a transformation involves computing the result of a simple mathematical function for every pixel of the image. The client then sends the new RGB image to the display. If, however, it can be shown that the Client component does not care which format is being used (i.e., the Client component does not actually manipulate the image data), the Decoder component can be replaced with a component which produces output in the YUV format. Doing so would eliminate the memory overhead of processing each pixel, as well as the computational overhead of performing a YUV-to-RGB conversion.

The optimisation is shown graphically in Figure 5.4. The original player, shown in section A of the figure, is transformed to the player shown in section B. The Decoder

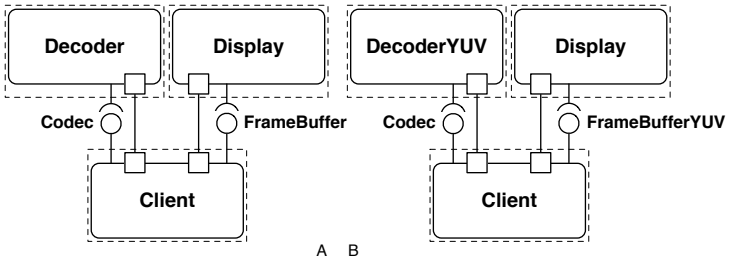

Figure 5.4. Component replacement in the componentised video player

component is replaced with a decoder which natively produces YUV, and the Client's connection to the Display is modified in order to identify the changed data format to the Display component. There is one major task that Currawong must perform to implement this optimisation:

1. Determine whether a particular region of memory is accessed by a given portion of code—in this case, whether the decoded image is accessed by the Client component.

5.2.3. Example 3: CAmkES protocol translation

The final CAmkES-oriented remedy, *component interposition*, is demonstrated through an improvement to the file-reading portion of the video player. The optimisation is represented graphically in Figure 5.5.

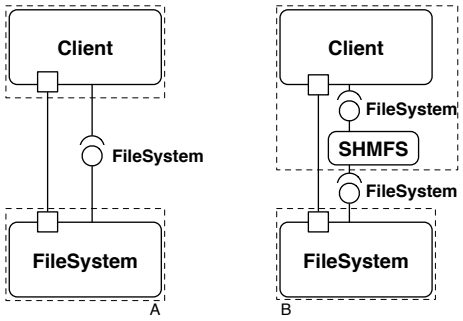

Figure 5.5. Component interposition in the componentised video player

In the original system, compressed video data are read from the FileSystem component at the request of the Client component. The data are copied to a shared memory buffer for use by the client. The Client, however, uses a standard POSIX-style `read()` call to get at the data, which means that the client supplies its own buffer. The data in the shared memory area must therefore be copied, again, to the client. Figure 5.6, section A, illustrates this process.

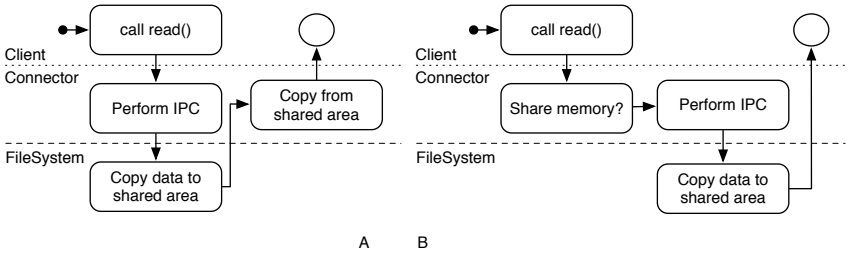

Figure 5.6. File system memory-sharing optimisation

A better approach is to be more like the process shown in section B of Figure 5.6, in which the buffer supplied to the Client's `read` call is shared with the FileSystem component. The simple approach is unsafe, because the starting and ending addresses of shared regions must be multiples of the system's page size (4096 bytes on the systems evaluated in this dissertation). If this is not the case, we must either share more memory than necessary, which is a security risk, or less than necessary, which would violate assumptions made by the component implementor (and would, consequently, almost certainly crash the component).

As with the second optimisation, Currawong must perform some basic static analysis here. It must:

1. Determine whether the memory used by the Client component is both page-aligned and equal in size to some multiple of the system page size;

2. Replace the connector between two components with a shared-memory connector.

5.2.4. Example 4: Android touch events

As described in Chapter 3, many services are provided to Android applications via IPC to a privileged process called the System Server. Among the services provided is notification of *touch events*. These are generated when the application user interacts with the touch-sensitive display by pressing or dragging one or more fingers on the display. Figure 5.7, section A, shows the activity performed by the standard system when a touch event is to be transmitted to an application.

When the user is interacting with the display, a lot of data is generated (compared with other input methods): the display hardware is capable of generating over 100 events per second, but in recent versions of Android the delivery rate is limited to 35 events per second [Hills 2009]. IPC in Android uses an Android-specific mechanism called Binder. Binder IPC is notoriously slow [Hills 2009], which both reduces application responsiveness (by a small amount) and contributes to overall CPU usage. On mobile devices, increased CPU usage results in decreased battery life, so reducing the performance overhead of delivering touch events could increase the per-charge longevity of the device.

For this example, as well as the next Android example, Currawong can rely on structural features of the application, without having to resort to code analysis, in order to

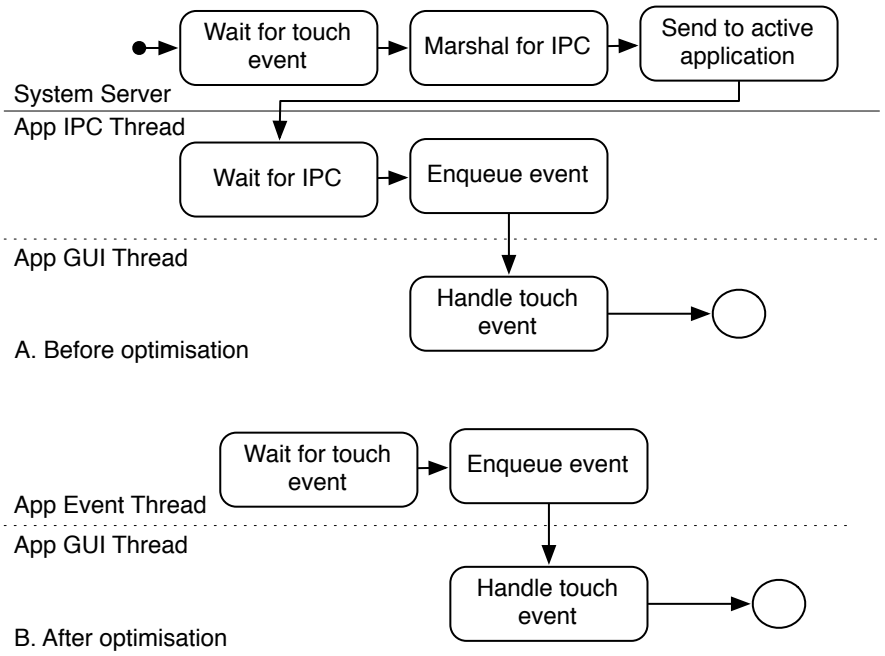

Figure 5.7. Touch events optimisation before (A) and after (B)

implement an optimisation. It must:

1. Determine that an application uses touch events; and

2. Add code to the application to support custom decoding of touch events.

5.2.5. Example 5: Android redraw

The second architecture optimisation applied to Android deals with the way applications draw graphics to the display. Some applications, particularly games, require high-frequency updates to large portions of the display. Android offers a variety of methods to accomplish this, and the best technique varies with the nature of the application (for example, certain kinds of 2D games are best served by the 3D API).

The simplest and most general-purpose way to do arbitrary 2D drawing in Android is to use the *Surface* technique. To use this approach, the application declares a class which inherits from the `Surface` API class. This class declares a method which the application class can inherit, named `onDraw`. The application performs its drawing when `onDraw` is called. When the application wishes to update its display, it calls the `invalidate` method, also defined in the `Surface` class. The advantage of the Surface approach is the simplicity of the API: the two functions described are literally all that is necessary to manage the display. The disadvantage is performance: the Surface class does not allocate memory from the pool directly accessible to the display, so it must be copied to display-

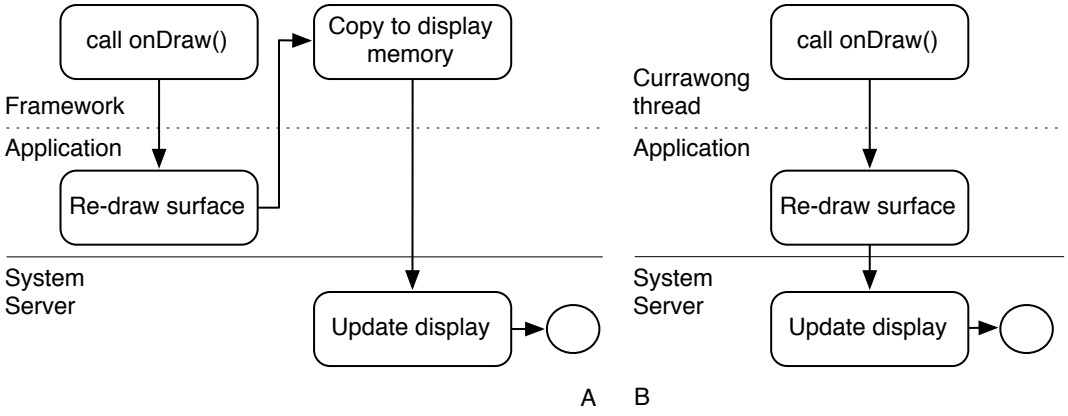

Figure 5.8. Redraw pathway (A) before optimisation and (B) after.

accessible memory on each redraw. This is expensive. The whole process is shown in Figure 5.8, section A.

A slightly more complex approach is to use the *SurfaceView* technique. This technique centres around a different class, SurfaceView, which directly allocates display-accessible memory. Applications using this technique are slightly more complex than those using the Surface class (SurfaceView is best implemented using another thread, for example), but they are more efficient, because they avoid the copy associated with Surface objects.

The difference between Surface and SurfaceView objects is technical, non-obvious, and somewhat hardware-specific. Consequently, some Android games which should be using the SurfaceView technique use the Surface technique instead.

The redraw optimisation rewrites applications using the Surface technique to use what is effectively the SurfaceView technique. A new package is added to the application which implements the requirements for interacting with a SurfaceView. This code starts a new thread in the application. That new thread calls `onDraw`, which allows the application to effectively behave as if the application were still using the Surface method. Thus the changes required to the application are minimal. The optimisation is shown in graphical form in Figure 5.8, section B.

As with the previous Android optimisation, this optimisation can be performed by analysis of structural features of the application. Currawong must:

1. Determine that an application is using the slower drawing style; and

2. Add code to the application to use the faster drawing style.

5.2.6. Summary of example requirements

The examples were chosen as representative implementations of the design remedies outlined in Section 4.3. Recall that these remedies are:

1. Combine protection domains;
2. Replace component or library;
3. Interpose component; and
4. Modify component-to-component APIs.

The three examples (the CAmkES examples) are remedied, respectively, by the first three remedies in the above list. The two Android examples are both remedied by the "Modify component-to-component APIs" remedy.

5.3. Providing an application representation

All remedies require some way to reference portions of the application. Recall that the four remedies may be divided into structural remedies and code remedies, where the former category makes use of structural features of the application and its component system, and the latter category also relies on code analysis and modification. Currawong should therefore provide a way to examine and modify application structure (for the structural remedies), as well as a way to reference portions of an application as a starting-point for further code analysis (for code remedies).

In fact, the way applications are represented is mostly kept hidden from the optimisation specification. Instead, a high-level approach was chosen: a portion of the application is *matched*, and then this match is transformed. The specifics of exactly how the application is stored in memory, and even the method by which it is analysed during the checking stage, are details of the implementation.

```
class _ extends View {
  protected void onDraw(Canvas _)
  { }
}
```

Figure 5.9. An example template

Because Currawong acts on binary code rather than source code, this choice—to use a high-level representation—entails some process for converting binary code to something equivalent to application source code. Currawong does this in different ways depending upon the language.

For Java applications, Currawong uses open-source tools to convert the compiled bytecode to and from an assembly-language representation. This process is discussed in more detail in Section 6.5.1. Briefly, however, all structural information is retained during the Java compilation process, so the original division of a Java-based application into modules, classes, inner classes, methods, and fields can be perfectly replicated in Currawong's application representation. Each method is stored along with an assembly-language representation of its implementation. This code is supplied as input to an instruction-level

simulator for data-flow search, discussed in more detail in the implementation chapter, Section 6.5.3.

For C applications, Currawong cannot perfectly replicate the structure of the original code, because it is not preserved by the compiler. Instead, Currawong extracts as much information as possible from the *symbol information* present in all executable programs. More information is presented in Section 6.6.1.

Optimisation specifications do not refer to application code directly. Instead, they refer to a matched portion. Application matching is achieved by presenting a *template* to the application representation. This results in the generation of zero or more *Match objects*, which represent portions of the application matching the template.

A sample template is shown in Figure 5.9. In fact, this is a simplified portion of a Java template from Example 5: Android redraw. It specifies a match on a class which inherits from ("extends") `View`. This class must contain a method called `onDraw` of type `void`, which accepts a single `Canvas` object as a parameter. Note, however, that neither the name of the class, nor the name of the `Canvas` parameter, is specified in the template: instead, the special variable `_` is used, to indicate that it does not matter what actually occupies that syntactic position. In other words, classes will match this template regardless of they are called (if they match the other criteria). The detailed design of Currawong templates is described in Section 5.7.2, because the design relies on the types of checking that Currawong can perform.

The advantage of the templating approach is that complicated specifications (as seen above: matching a function of a certain name, within a named class, taking a particular type of parameter, and so on) can be expressed both concisely and readably.

5.4. Matching and checking

Before it can optimise an application, an architecture optimiser must recognise a performance problem. Chapter 4 codified five generic types of performance problem as *performance anti-patterns*: context switching, copying, overly-generic or inflexible APIs, unsuitable data structures, and reprocessing of data. This section discusses how these anti-patterns might be specified and identified.

When designing a program that will recognise anti-pattern implementations, the obvious strategy is to create a program that simply recognises each anti-pattern. Such a program would detect any instance of context switching, copying, and so on. This is a good solution, perhaps, for a specialised error-detecting program, but it is not ideal for an architectural optimiser: for reasons discussed in more detail below, recognising such generic patterns is both difficult and slow. Even if generic anti-patterns could be recognised efficiently, it is difficult to imagine what could be done with them. The logical thing to do would be to apply an anti-pattern remedy, but anti-pattern remedies are very domain-specific and tend to be the result of human ingenuity rather than mechanical application of a template. Therefore, humans must be involved in the entire process of anti-pattern recognition and remedy application. Currawong allows optimisation writers to direct the optimisation process through the use of specification.

5.4.1. The importance of specification

A key idea of architecture optimisation is to make use of domain-specific knowledge to reduce analysis requirements. In Currawong, domain-specific knowledege is expressed via a specification language. The importance of a specification language is described here by first considering the alternative.

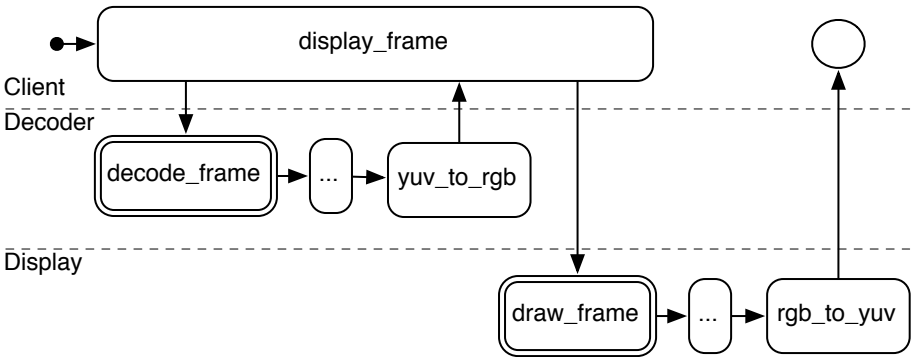

Figure 5.10. Unnecessary data manipulation in a video player (API calls have a double border)

Figure 5.10 shows the activity between three components in the componentised video player system (this is a portion of the complete system which was introduced in Figure 3.4). In this example, the Client calls `decode_frame`, which is part of the API provided by the Codec component. Frames are decoded into the YUV colour space, but converted to the more programmer-friendly RGB colour space before being returned to the Client. The client then calls `draw_frame`, which is part of the Display component's API. However, the Display component requires data in the YUV colour space, so the newly-decoded data is converted again before being displayed. This is an example either of API mis-use, or a poorly-designed API.

This figure describes an example of a data-reprocessing anti-pattern of the kind discussed in Chapter 4. Suppose we were to attempt to locate this type of problem automatically, without any domain-specific knowledge of the API provided by the Decoder and Display components. Doing so would be a difficult problem. At the very least, it would require pointer analysis of the entire application—a prospect that is at best time consuming, and at worst impossible without source code (because type information is lost).

In fact, in this particular example, the analysis requirement doesn't stop at the boundaries of the Client component, but continues into the Decoder and Display components, as the Client makes calls to those components. An analogy could be drawn between this component system and a standard monolithic system in which Client would represent the application, and Decoder and Display represent libraries used by the application. In the worst case, the generic analyser would have to perform a static analysis of all code in the system to detect this type of problem.

The example of figure 5.10 shows that a generic anti-pattern detector would be difficult to write, but even if they were easy to write and executed quickly, a generic anti-pattern detector would be difficult to use effectively. This is because finding anti-patterns is only half of Currawong's task. The other half of the work involves creating a remedy for the anti-pattern and applying it. However, all of the optimisations discussed in Chapter 4 are domain-specific remedies, custom-written for specific problems. None of them could be derived automatically from a set of remedies. For example, FBufs (discussed in Section 4.1.5) is an optimisation which reduces context switching and copying overhead. However, coming up with FBufs required domain-specific knowledge and ingenuity. It is not something that could have been produced mechanically from a program written to apply simple rules (such as "reduce copying overhead").

Currawong's solution to this dilemma is to incorporate the process that domain experts already apply when implementing optimisations in code: to recognise the *characteristics* of a particular anti-pattern, rather than recognising the anti-pattern itself. The optimisation writer (who is, presumably, a domain expert) specifies the characteristics of the anti-pattern using a specification language.

Allowing custom specification means that Currawong allows optimisation writers to make use of their own domain-specific knowledge of the system when writing optimisation specifications. Exploiting this knowledge can significantly reduce the amount of analysis required by Currawong.

The benefit of specification is best illustrated by referring again to Figure 5.10. Automatically analysing the system to detect the data reprocessing without any API-specific knowledge would be very difficult. It seems likely that the analyser would have to determine that `yuv_to_rgb` and `rgb_to_yuv` are inverses, for example.

If, however, we assume that this performance issue is known to the author of the Decoder and Display components, the optimisation author can write an architectural optimisation which detects usage of the Decoder and Display API functions `decode_frame` and `draw_frame`, respectively. Safely detecting that this optimisation is possible is then reduced to determining that the Client calls the appropriate API functions, and ensuring that the `draw_frame` is always called with the output of `decode_frame`. By incorporating domain-specific knowledge of the Decoder and Display APIs within a specification, the optimisation writer significantly simplifies the optimisation process. In the nomenclature of the previous chapter, specification has significantly reduced the data-sensitive requirements of the matching for this example.

5.4.2. Recognising anti-patterns with Currawong

The complexity of the matching and checking being performed depends on the nature of the program transformer. Refactoring engines, as described in Section 2.3.1, simply examine the structure of the code. Active library implementations, however, must do more work to ensure that a portion of code is suitable. Broadway, an active library implementation discussed in Section 2.3.2, performs data-flow analysis in addition to the syntactic matching performed by refactoring engines.

Some of the surveyed program transformers, then, implicitly perform several kinds of

checking. In these systems, however, the checking being performed tends to be intertwined with the rest of the system: Broadway, for example, is designed around data-flow analysis of expressions related to function arguments, and this design decision is reflected in the syntax of Broadway's specification language.

In Currawong, one may wish to only apply an optimisation to a particular class, or to a particular type of application-to-system interface. This desire would result in a specification that says something about the *structure* of the application—"only apply the optimisation to portions of the application that call these functions, or that implement this class", for example. However, the optimisation writer may also wish to make *control-flow-sensitive* statements about where to apply optimisations, such as "only apply this optimisation if function A has been called before function B". The criteria for applying the optimisation may even be more complex, and may include *data-flow-sensitive* statements, such as "only apply this optimisation if the return value of function A is passed as the first parameter to function B".

In Currawong terms, all these criteria—structural, control-flow, and data-flow—are called *matches*, and the overall process of *matching and checking* is what ultimately results in an optimisation being applied to an application. Matching is shown as step 3 in Figure 5.1. Currawong's design attempts to strike a balance between keeping these match types separable, and making them easy to express and use. The first type of match, *structure search*, is supported by the syntax of Currawong's specification language (complete details are given in Section 5.6). The second type, *control-flow search*, as well as the third type, *data-flow search*, are explicitly separated from structure search, in order to improve specification readability.

The three match techniques are discussed below. The previous chapter introduced the concept of *data-insensitive* and *data-sensitive* checking. In these terms, the first two methods are data-insensitive, and the third method is data-sensitive.

Structure search

Structure search refers to locating a portion of code by reference to its structure, that is, by reference to the hierarchical division into modules, classes, and functions common to object-oriented programming. Structure search refers to definitions (such as "this is a class named X") and references (such as "create an instance of class X"). Because architectural optimisation deals with API usage, it is far more common for a structure search to indicate references, rather than definitions.

Normally, the terms *class*, *function*, and *module* refer to source code. However, compiled code (and bytecode) tends to retain references to its code structure as (essentially) a side effect of support for run-time features like dynamic linking, debugging, or reflection. It is, therefore, both meaningful and extremely convenient to refer to familiar structural features, even in compiled or tokenised code.

The following are examples of architectural optimisations which make use of structure search:

1. In the Decoder class, rename the function `decode_frame` to `_old_decode_frame`.

2. Replace all instantiations of the form `com.example.player.Decoder()` with instantiations of the form `com.example.player2.LegacyDecoder()`

3. Replace all calls to `com.example.player.Decoder.decode_frame` with calls to `com.example.player.Decoder.decode_frame_toyuv`.

The matching portion of the above examples resembles the type of activities performed by refactorings. This should not be surprising, since both activities refer to code structure (Section 2.3.1 defines refactorings as structural transformation). In fact, both the matching and transformation portions of refactorings are structural—structure search forms the match portion of a refactoring.

Despite the simplicity of the concept, structure search in Java and in component systems has interesting properties that make it more powerful than might be expected. One of these properties is ability to identify unique portions of code. Consider example 3 above: the Decoder class is referred to by its *fully qualified name* of `com.example.player.Decoder`. Following Java convention, this refers to a class named Decoder, in the package named `com.example.player`. Since Java package names are required to be unique, this is sufficient to uniquely identify the Decoder class. If the optimisation writer is also the author of the class, as is intended, then a positive identification of the class is all that is needed to make assumptions about the behaviour of the class and, thus, to write optimisation specifications involving that class. Without the ability to uniquely identify code portions, the optimisation writer would have to identify code by its behaviour, a much less straightforward process. This is another example of Currawong's emphasis on reducing analysis requirements by replacing data-sensitive checks with data-insensitive checks.

The optimisation problem introduced above (Figure 5.10) can be checked almost entirely using structure search. In this example, the data returned by one API call (`decode_frame`) is returned in the incorrect format for another API call (`draw_frame`). An optimisation writer who knows the behaviour of these two functions can simply recognise them using structure matching. Unfortunately a successful structure search is not sufficient to safely apply the optimisation, because we must also verify that the data returned by the first call is not used anywhere else (since it would be in the wrong format). This seems to be a common characteristic of structure search in optimisation problems: it serves as a basis for, and lowers the requirement for, other types of checking.

Simpler architectural optimisations can be solved entirely using structure matching. Dig and Johnson have shown that many types of API evolution can be represented as refactorings [Dig and Johnson 2005]. As discussed above, a successful structure search is all that is necessary to perform a refactoring, so the types of API evolution described by Dig and Johnson can be catered to by structure search.

Structure search operates at a purely textual level. Fundamentally, it is a text search technique, and standard text-searching operators can be utilised. For example, optimisation authors could make use of wildcard-style searching to identify all functions which include a common subexpression, such as `*display*` to identify all functions which include the word "display", or could even make use of regular expressions to support more

complicated specifications. This style of specification is common in aspect-oriented programming environments such as AspectJ [Kiczales et al. 2001]. Particularly popular is the special case `*`, which matches all functions (usually within a certain higher-level scope, such as a class). This type of specification suits a common AOP pattern, which is to add a small piece of code to the beginning or end of many functions (to perform logging, for example). It appears to be less useful for architecture optimisation, which focuses on an API in which the names of all functions are known to the specification author.

One real-world problem with structure matching as presented is the existence of multiple versions of classes, functions, or modules. In Java, for example, packages are versioned. Dealing with versioning information is an implementation detail and does not meaningfully change the design. One solution is to allow the version information to also be matched. The version information could be represented as an attribute attached to the versioned scope (package name in the Java case) which is also made available for matching purposes in all lower scopes (classes and methods in the Java case), making it an *inherited attribute* in attribute grammar terms [Aho et al. 1986].

Control-flow search

The second type of match supported by Currawong is *control-flow search*. Control-flow search refers to specifying a portion of code based on the way control flows through it. For example, control-flow search can identify a sequence of function invocations (such as "call function A, then call function B"). Because control-flow search refers to syntactic features of the code, it can be specified as a sequence of structure searches (e.g. "find the location where function A is called, and then function B is called subsequently"). Control-flow search is rarely used by itself, because control flow provides limited additional information beyond what can already be found through structure search. Architectural optimisations that require control-flow search tend also to require knowledge of data flow within the system.

The following are examples of architectural optimisations which make use of control-flow search:

1. If `memhog_1` and `memhog_2` are called in the same function, but `create_mempool` is not called, add `create_mempool()` before `memhog_1`, and add `destroy_mempool()` after `memhog_2`.

2. If there is a function-call sequence `decode_frame`; `draw_frame`, replace `decode_frame` with `decode_frame_yuv`.

3. If functions from the Decoder component and the Display component are used in the same function, place the Decoder component into the same protection domain as the calling function.

These examples make use of control flow information to make quite significant changes to code. Unfortunately, each of these examples is potentially unsafe:

1. Of the three, the first example probably has the greatest chance of being correct. However, the optimiser implementing this example must ensure that the function cannot exit in any unexpected ways. If it does, the cleanup action (`destroy_memory_pool`) may not be executed. In Java, this amounts to ensuring that the cleanup action is executed even if an exception is thrown. In C, this may be easy to ensure (if the function does not call any other functions), or it may be completely impossible (if, for example, the application calls another function, such as `longjmp`, which modifies the instruction pointer unpredictably).

2. The second example changes the target of a function call, implementing a solution to the optimisation problem shown in Figure 5.10. However, the example does not check that `decode_frame` and `draw_frame` refer to the same data (i.e., that the parameter passed to the second function refers to the object created by the first function): without this check we cannot guarantee that the optimisation will behave as intended.

3. The third example combines the protection domains of two components. If the newly-added component is malicious or buggy, it could corrupt the private data of the component whose protection domain it has joined.

In the second and third example above control-flow search is a necessary, but not sufficient, part of the matching process. These examples may become safe when combined with domain-specific knowledge. For example, it may be judged acceptable to add the Decoder component to the protection domain of its caller, even without verifying its correctness. In general, however, control-flow search is not used by itself, but is instead combined with data tracking to provide data-flow search, discussed below.

Structure search and control-flow search are rigidly defined here so that there is no overlap. However, it must be acknowledged that the distinction is somewhat artificial: a compelling argument could be made that method invocations count as control flow rather than structure. I have two responses to this: firstly, as far as architectural optimisations are concerned, method-invocation sites are treated *as if* they were a structural feature, rather than a control flow feature, and are thus described as such. Secondly, the separation here into three match types is primarily for ease of exposition and does not reflect any fundamental separation in Currawong's design. In other words, the categorisation of the match methods is far less important than what they do.

Data-flow search

Some optimisations require knowledge of the system beyond structure or control-flow information in order to guarantee correctness. That extra knowledge comprises statements about some, or all, of the data in the system—before, during, or after the optimised portion. The procedure for making this kind of statement is termed *data-flow search* in this dissertation.

Just like structure search and control-flow search, data-flow search may also be specified declaratively. For example, the Broadway domain-specific optimiser performs data-

flow analysis. Optimisations in Broadway can be predicated on data-sensitive properties of function parameters, and this is specified declaratively. Figure 5.11 shows an example included in Guyer and Lin's paper [Guyer and Lin 2005]. This declarative specification method is both concise and easy to comprehend.

```
procedure fgets(s, size, f) {
  when (size == 1)
    replace-with %{ (*${s}) = fgetc(f); }%
}
```

Figure 5.11. Verification as search in Broadway [Guyer and Lin 2005]

The following three architecture optimisations require data flow search:

1. If there is a function-call sequence `X = decode_frame(); draw_frame(X)`, replace `decode_frame` with `decode_frame_yuv`.

2. If data returned from a function are not accessed by the caller, replace the function call with a call to another function which does not return that data.

3. If functions from the Decoder component and the Display component are used in the same function, place the Decoder component into the same protection domain as the calling function. Verify that the Decoder component cannot interfere with the Display component in unexpected ways.

These examples all require tracking of data within the application being examined. The first example a proof that two variables refer to the same object: the result of `decode_frame` should be passed to `draw_frame` . In C, this could be achieved by showing that two pointers point to the same location. The equivalent in Java is to demonstrate that the object returned from the first function is the same as the object passed to the second function. The second example involves slightly more work: the returned data must not be *used* anywhere, but the variable referencing that data can be copied, supplied as an argument to functions, and so on. The third example is still more complex: the architecture optimiser must perform a pointer safety analysis of the entire component, to prove that it cannot inadvertently overwrite memory.

These analyses are relatively-simple forms of *symbolic execution*, a method of program analysis through execution in which actual data values are not tracked, but in which constraints applying to data of interested are accumulated as execution proceeds [King 1976]. Section 6.3.3 contains more information about this technique as used by Currawong.

The diversity of data-flow search techniques is handled through abstraction: an analysis API is exposed to optimisation specifications, through which data-flow analyses can be performed. In other words, Currawong attempts to keep the implementation details of its data-flow search hidden from the specification. Instead, specific analyses are exposed via language constructs. For example, the task of checking pointer identity between two variables within a function is indicated using the CSL syntax `var1 = var2`.

The content of the search API is explained in more detail below.

This section discussed the first part of the Currawong process: matching and checking. The second part, *transformation*, is discussed next.

5.5. Transformation

Section 4.3 described the transformation remedies that should be applied by Currawong in high-level terms. They are:

1. Combine protection domains
2. Replace component or library
3. Interpose component
4. Modify APIs

To actually implement these remedies, Currawong must make material changes to the application. Transforming an application involves both *reference*, that is, locating the portion of the application to be modified, and *application*: performing the modification on the referenced portion of the application.

This section discusses the requirements of each remedy in terms of what kind of API is required of the Currawong Application object. The specifics of the API are discussed during the discussion of the specification language in Section 5.8.4, because they are language-dependent.

The remedies are discussed in two broad categories: those that do not involve modifying application code (architecture-level transformation), and those that do (code transformation).

5.5.1. Application architecture transformation

The first three remedies—*combine protection domains*, *replace component or library*, and *interpose component*—involve no modifications to application code. Instead, these remedies, summarised in Figure 5.12, operate at the boundaries between portions of an application (or component system).

To accommodate this sort of transformation, the application representation should include a model of the component system, including components, connections between components, and protection domains. The optimisation specification should be able to use this model to combine protection domains, to replace one component with another one, and to interpose a component.

5.5.2. Application code transformation

Not all systems are as neatly componentised as the example above. Android, for example, does not use a distinct architecture definition language, and most services are provided by a single component. When dealing with real systems such as Android, it is often simpler

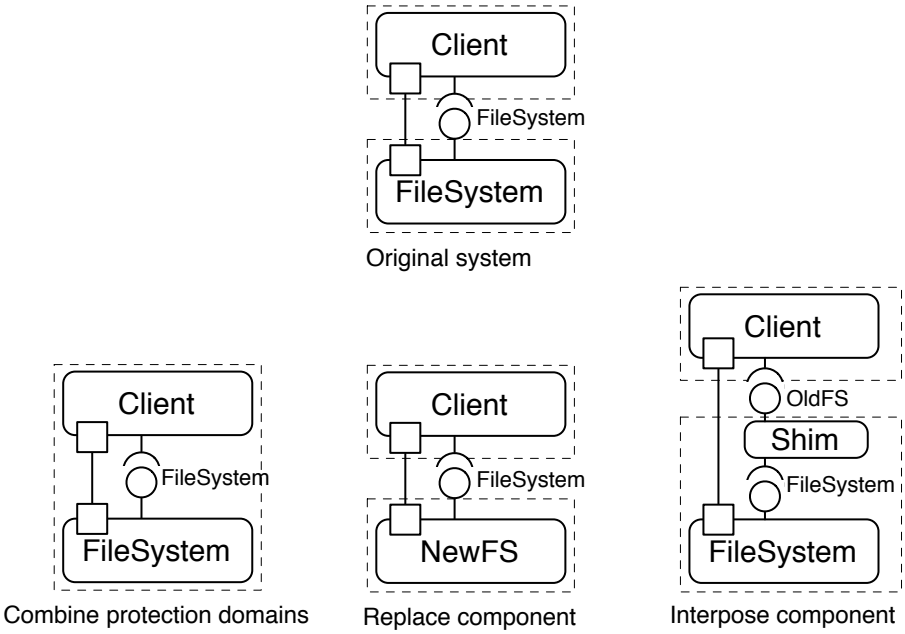

Figure 5.12. Application architecture transformation

to discuss architectural transformations in terms of minor code modifications. In Android, for example, it is difficult to combine protection domains of components, particularly when one of them provides system services—a simpler approach is to add additional code to the application and modify the application code to use the new addition rather than the existing system services. The fourth transformation method—*modify APIs*—relies on code modification, at least at a simplistic level. It recognises that code modification is sometimes convenient.

Unfortunately, the question "what sorts of code modification should Currawong support?" is difficult to answer, because code modification is, fundamentally, an open-ended programming task. However, the optimisations described in this dissertation tend to have only two simple code-modification requirements.

The first requirement is for *function-call renaming*, in which all calls to a certain method are changed so that a different method is called instead; and *function renaming*, in which a function's name is changed. Note that, given suitable component interposition, these two operations are equally powerful. Currawong includes both for the sake of convenience only.

The second requirement is to *add code* to the application. Code addition is limited to adding extra functions to existing bodies of code.

In summary, there are two major types of transformation: those which involve actually modifiying code, and those which work at a level above the code, such as at the component-system level for CAmkES. It is desirable, but not always practical, to avoid code modification.

5.6. Specification language requirements

The above sections provided a background, describing the requirements of Currawong's matching and transformation API. On the basis of that background, the requirements for the programming language that implements that API can now be described.

Optimisation writers describe both matching and transformation for a given optimisation using 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.

```
MatchOnDraw is Java {
    class $ClassName extends android.view.View {
        protected void onDraw(Canvas _)
        { }
    }
}
MergeOnDraw is Java {
    class $ClassName extends Android.view.SurfaceView
            implements Android.view.SurfaceHolder.Callback {
        private int _cw_tok;
        public void surfaceDestroyed(SurfaceHolder s) { }
        public void surfaceCreated(SurfaceHolder s) {}
        public void surfaceChanged(SurfaceHolder s,
            int fmt, int w, int h) {
            _cw_tok = au.com.nicta.cw.Draw2D.init(this);
        }
        protected void invalidate() {
            au.com.nicta.cw.Draw2D.invalidate(_cw_tok);
        }
    }
}
MergeOnDrawInit is Java {
    class $ClassName {
        $ClassName {
            getHolder().addCallback(this);
        }
    }
}
optimise(ondraw, App) is
    Match = App.match(MatchOnDraw),
    App.add_package('au.com.nicta.cw'),
    App.merge(Match, MergeOnDraw),
    App.merge_all(Match, MergeOnDrawInit),
    App.rename_method
        (Match.ClassName, 'onDraw', '_cw_onDraw'),
    App.merge(Match, MergeOnDrawInit).
```

Figure 5.13. An optimisation specification written in CSL.

A complete optimisation specification is given in Figure 5.13. This describes an Android optimisation, the *Android redraw optimisation*, that is discussed in more detail throughout this section. Evaluation of the effectiveness of this optimisation, as well as the motivation behind its application, is covered in Chapter 7.

Most of this specification's details can be ignored at this stage (a complete description is given in Section 5.2.5). It was included to give an idea of the concision of a declarative specification language such as CSL, simply in terms of the small number of code lines required to describe an optimisation. Note the basic format of the specification: Most of it is written in a Java-like templating language, but the final portion (which contains the transformation rules) is written in CSL.

Concision is a worthwhile goal by itself, but it takes on special significance in transformation languages. Transformation specifications tend to be small and rather simple to begin with, so any unnecessary code is particularly noticeable. Tree transformations written in TXL [Cordy 2006] are a good example of the size of these sorts of programs, although architecture optimisation specifications are slightly more complex than TXL programs.

Similarly, match criteria tend to be restricted to small, well-defined problem domains, and are amenable to compact representations. In particular, the match criteria described above can all be formulated as search problems. For these reasons, a logic programming language was selected as the basis of CSL. The fundamental characteristic of logic programming languages is that they are goal-directed: they use a proof mechanism to perform deductions in a goal-oriented manner, guided by rules [Kowalski 1988]. Logic languages are expressive, concise, and well-suited to search problems. Because they are declarative, programs written in logic languages express *what* rather than *how*, that is, they describe a particular problem, rather than explain in detail how to go about solving it. For this reason, CSL is based on a logic language. In particular, it is based on Prolog, and, like Prolog, CSL uses unification as its primary execution mechanism.

```
class Example {
   void $Y(int) {}
}
```

A

```
class('Example', Functions),
filter(Methods, (void, _, [int])).
```

B

```
for class in AllClasses:
    if class.name == 'Example':
        for method in class.methods:
            if len(method.params) == 1
                and method.params[0].data_type == 'int'
                and method.return_type == 'void':
                return function
```

C

Figure 5.14. Finding functions in a class through pattern-matching (A), unification (B), and iteration (C).

Structure search is a good example of a problem that is well-suited to declarative expression. Figure 5.14 shows three ways to perform structure search. The task is to find the methods taking a single int as a parameter and returning void, within a class named Example. Method A uses pattern-matching; method B, unification (for Currawong's purposes, a syntactic transformation of pattern-matching); and method, C, iteration. The iterative approach reveals details about the implementation of the data structure used: that the object representing a class contains a field named `methods`, that the classes objected can be

iterated over, and so on. The iterative version is also significantly longer than the other two versions.

By contrast, the unification version (B) is much smaller, making use of functional programming techniques, such as list filtering, to express intent more clearly. However, it is still rather difficult to understand the intent of the search, because the search terms ("Example", "int", and so on), are obscured by the syntax for pattern-matching and filtering. This approach also reveals details about the format of the data structure used for representing classes, i.e. that it is a two-element structure containing a class name and a list of functions.

The pattern-matching form, A, is an improvement over both alternatives in terms of both length and readability. This form uses a specification written in the language being searched (Java in this case), making it easy to read. It is immediately obvious that certain aspects of the template (such as the function name matched) are variable. Method A also completely hides the underlying data representation.

This example demonstrates that syntax is important. Methods A and B are similar (they are both declarative specifications), but they are syntactically very different. CSL supports method A using a templating approach: specifications in style of method A are translated to the method B form at run-time. The details of this approach, particularly the method by which the template is translated to CSL terms, are discussed in Section 5.7.2.

The above requirements motivate the design of the specific language proposed for Currawong: Currawong Specification Language.

5.7. Currawong Specification Language

CSL is a variant of Prolog [International Standards Organisation 1995] with extensions for templating and object orientation. The base language is a minimal Prolog: a language using first-order unification as its execution mechanism, supporting lists, structures, terms, and atoms. CSL by itself supports strings, floating-point numbers, and arbitrary-precision integers as its base data types, but these are augmented with complex built-in data types representing the application under investigation, and search result matches.

5.7.1. Specification structure

The specification is defined with the Prolog rule named `optimise(+Name, +App).` The first argument is the specification name, as an atom. The second parameter is the application to be optimised. As a shortcut, this description follows the de facto method of referring to clauses by their name, a forward slash, and their arity (the number of parameters they accept). In this case, the `optimise` clause accepts two parameters, and is written `optimise/2`. The body of `optimise/2` is written, as is standard in Prolog, as a list of expressions separated by commas and terminated with a semi-colon. As evaluation proceeds, a number of rules within the Application object are evaluated, which have the side effect of modifying the Application object. If evaluation of `optimise/2` succeeds,

these modifications are made to the actual on-disk application.

5.7.2. Syntax

CSL is syntactically similar to Prolog. CSL atoms begin with a lower-case alphabetic character, and variables begin with an upper-case alphabetic character. A minor change is that clauses are defined using the keyword `is` rather than the syntax `:-`. As with Prolog, clauses consist of a sequence of expressions separated by commas and terminated with a dot (`.`).

The following sections describe the major differences between CSL and standard Prolog, as well as the two major built-in data types, Application and Match, which provide the API for specification-directed architecture optimisation.

Objects and data types

Encapsulated data types (i.e., objects) are a convenient addition to CSL, because they make specifications more concise. There are two different kinds of uses of encapsulated data types in Figure 5.13. The first kind, as exemplified by `App.match()`, `App.add_module()`, and so on, look like function calls in an object-oriented language. The second kind, as shown in `Match.ClassName`, looks like a field reference in an object-oriented language.

To add support for this kind of data type, the language is first extended to support *immediate evaluation*. Prolog already offers a limited form of immediate evaluation—the `is` keyword—but its semantics are inconvenient for CSL's purposes.

```
expr(Arg1) is                    expr(Arg1, Result) is
   expression,                      expression,
   expression,                      expression,
   expression.                      Result = expression.
A                                B
```

Figure 5.15. Implicit return values in CSL rules.

The rules for CSL immediate evaluation are as follows:

1. All rules are treated as if they have an implicit unbound additional parameter. This parameter is unified with the last successfully-evaluated expression in the rule. Figure 5.15 illustrates this behaviour. Rules of the form shown in part A of that figure are treated as if they were of the form shown in part B.

2. When an expression of the form `Expr((...))` is encountered, evaluate the expression immediately.

3. Continue evaluation as if `Expr((...))` was syntactically replaced by the return value from the expression after it has been evaluated.

Immediate evaluation provides a way for rules to act as *macros*, returning other expressions. This forms the basis for an object-oriented type system. The type system itself can then be implemented as *syntactic sugar*: in other words, it can be implemented through purely syntactic transformations on the code.

To support object field access of the form `Object.field`, proceed as follows:

1. CSL defines an object type to be the two-element list `[DataType, Value]`, where DataType must be bound to a string, and Value can be bound to anything at all, or can be unbound.

2. When an expression of the form `Object.field` is encountered, unify `Object` with the list `[DataType, Value]`. This unification must succeed exactly once.

3. Construct a structure name consisting of the string to which DataType is bound, an underscore, and `field`. For example, if the object access is `App.ClassName`, and `App` unifies with `[java_application, _]`, the structure name would be `java_application_ClassName`. Call this new name `StructureName`.

4. Continue the evaluation as if the object access were replaced by `StructureName ((Value))`.

This method easily extends to support structure access of the form `Object.structure (Arguments)`. To do this, the replacement in Step 4 becomes `StructureName (Value, Arguments)`.

Template support

A typical architecture optimisation includes structure search—the process of identifying a portion of code through reference to its structural features, such as class names, package names, and function names (covered in detail in Section 5.4.2). A convenient way to express a structure search term is to write the desired code that should be matched directly in the object language—that is, in the language that the application being optimised was written in. For example, to reference a particular class in Java, one would like to write `class ClassName`. For similar reasons, it would also be convenient to specify the data-flow conditions for data-flow search directly in the object language. Finally, some forms of transformation involve adding code; obviously the best way to describe the code to be added is simply to write it in the desired language.

CSL therefore has a unique need for portions of optimisation specifications to be written in the object language. This is achieved in CSL through templating.

CSL's templating language takes advantage of the immediate evaluation syntax described above. In place of any expression in CSL, the optimisation writer may use the syntax `TemplateName { Code }`. This is translated by the parser into `template_TemplateName "Code"`, i.e. everything inside the braces is supplied as a string to the rule `template_TemplateName`. This rule is then immediately evaluated, as per the steps described in Section 5.7.2, and the result is syntactically inserted into the specification code.

```
MatchOnDraw is Java {
  class $ClassName extends android.view.View {
    protected void onDraw(Canvas _);
  }
}
```

Figure 5.16. CSL templating example

Figure 5.13 shows two examples of templating. The first example, reproduced for convenience as Figure 5.16, specifies a structural match. This particular template matches any Java class which extends `android.view.View`, which contains a `protected` method named `onDraw`, accepting a single parameter of type `Canvas`, and returning `void`.

Using templates to perform structure search results in two problems. The first is a referencing issue: having matched a portion of the application, how can one refer to the matched portion? The second problem is a generality issue: how can we ensure that a template isn't overly-specific, not matching portions of an application that it should?

The referencing issue can be re-stated as the requirement that the optimisation specification should have some way to reference portions of the matched code. Matched portions of code deal with this problem by defining a hierarchy that matches the structure of the code, and is accessible using object attribute references (see below for more information on Match objects).

The generality issue is addressed through support for *variables* in the templated code. Any Java or C identifier in a template, if prefixed with a dollar sign, is treated as a variable. The example in Figure 5.13 uses this feature to identify the name of the class. Variables can be used in place of any identifier in the object language. Instead of specifying a class name to match, the variable `$ClassName` is supplied instead. These variables can be accessed by other portions of the optimisation specification. A special-case variable, named `_` can also appear in templates. This variable functions as a wildcard—it can be used multiple times within a template, but is never bound, and cannot be accessed from the optimisation specification.

The idea of using the object language as a specification system is not new: many other transformation systems support direct injection of object code into the specification language. Broadway supports specification of data-flow conditions in the object language, as can be seen in Figure 5.11.

The logic-language-based aspect-oriented programming system TyRuBa [De Volder 1999] also supports specification in the object language. TyRuBa's basic syntax for this type of reference, which CSL borrows, is to surround object language strings with brackets. TyRuBa's support for object-language references is quite rudimentary: it treats the target language snippet as simple text, and matches it against the code in this way (after normalising whitespace). By contrast, templates in CSL are a complete domain-specific specification language which must obey formal rules.

Real systems are built using multiple programming languages. Therefore, an architecture optimiser for real systems must support multiple programming languages, too. CSL's

templating system was designed to accommodate multiple programming languages by requiring explicit specification of the template name when creating templates. This means that different languages can be supplied to different templates—for example, the one implementation could support Java and C through appropriately-named templates.

Summary

The major features of CSL are:

CSL is **templated** so as to provide a domain-specific way for the code under examination to be checked and modified. Templates let optimisation authors write portions of the optimisation specification in the object language. In Figure 5.13, the clauses beginning with `MatchOnDraw` and `MergeOnDraw` are templates.

CSL is **declarative** both because it provides a concise way to express information, and because it resembles the natural way that programmers talk about program modification: "replace any call to method X with a call to method Y". This high-level approach also means that optimisations need not be aware of the implementation details relating to program analysis. This, in turn, means that the implementation details may change without affecting the optimisations making use of them: it insulates optimisation specifications from changes to Currawong's implementation.

CSL is **extensible** as it is a complete programming language. This means that vendor-specific optimisation support routines can be supplied externally to Currawong. In the current implementation of Currawong, the Android- and CAmkES-specific portions of CSL need not be, and in fact are not, built in.

Finally, CSL is a **logic language**, using first-order unification as its method of execution. Using unification for execution gives CSL some powerful properties. For example, an optimisation specification can make use of the back-tracking behaviour of unification to match two mutually-interdependent, but structurally separate, portions of the application. Doing so requires just two lines of specification code.

5.8. The Currawong API

CSL's abstraction of object language and application behaviour is handled by an API. Most of the API is provided by two objects: Application object and Match objects. CSL's Application object implements the API described earlier in this chapter, starting with Section 5.3.

An Application object represents the application being examined, and is system-specific: an Android Application object is different from a CAmkES Application object. As described above, Application objects abstract the details of the access to code; the layout of code on the disk; the process required to perform static analysis on that code; and the method by which code is transformed.

The following sections discuss the Currawong API with respect to the way it is used by an optimisation specification. Figure 5.17 shows this process: Matching starts with structure search(1), optionally followed by additional structure searches as part of a control-

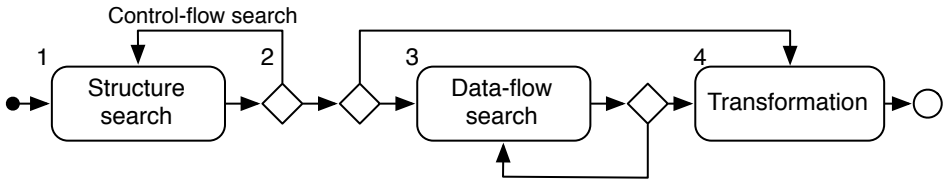

Figure 5.17. Matching in Currawong

flow search(2), optionally followed by one or more data-flow searches(3), followed by rewriting(4).

5.8.1. Structure search

Matching in Currawong starts with structure search. To perform a structure search, the optimisation specification supplies a template to the `Application.match` rule. The result of this rule is zero or more unifications resulting in `Match` objects. Match objects represent a portion of code within the Application which has matched a particular template.

Match objects are a very important part of Currawong. They perform three functions:

1. They are the basis of structural and control-flow search;
2. They provide information to aid data-flow search; and
3. They provide a reference for code transformation.

Match objects can serve these three roles by taking advantage of templating. The concept of templating as a way to represent applications was introduced in Section 5.3. Templating as a language extension was described in Section 5.7.2.

In the context of structure search, Match objects are quite simple. A template is applied to the application under examination. The resulting match object makes structural information from the match, as well as information about the location of any function calls within the match, available to the optimisation specification.

Structural information refers to the type of structural information described in Section 5.4.2, i.e., classes, functions, and modules. Other structural information may be present, depending on the system being analysed. For example, if the system is a component system, an additional structural level may be supplied describing the system architecture.

The **locations of function calls** (if the structure search matches any code-containing structure) is a special-case of structural information. However, rather than describing a scope, this information refers to a piece of application code.

Match objects also make any variables matched during template matching available to the optimisation specification. The content of this information (variable matches and structural information) is deliberately opaque to the optimisation specification—the only

thing the specification can do with it is pass it to APIs which perform further matching, or which perform transformation. This is discussed in more detail below.

5.8.2. Control-flow search

Control-flow search can be expressed as a series of structure searches (Section 5.4.2 covers this in detail). More specifically, control-flow search can be expressed as a sequence of increasingly-constrained structure searches. The first structure search is completely unconstrained; the second is constrained in that it must occur after the first one in a plausible program control flow; the third must occur after the second, which implies that it also occurs after the first; and so on. Currawong represents this scenario by supporting an additional parameter to the `match` rule through which the match which chronologically precedes the desired match is supplied. In this manner a chain of matches may be generated. Figure 5.18 shows an example of this API. In this example, the first `match` command produces a match object based on the template named `FirstTemplate`. The second `match` command refines this match object, reducing the matched code to that which matches `FirstTemplate` and, subsequently, `SecondTemplate`. The final `match` command further refines the match object: `Match3`.

```
Match1 = Application.match(FirstTemplate),
Match2 = Application.match(SecondTemplate, Match1),
Match3 = Application.match(ThirdTemplate, Match2).
```

Figure 5.18. Control-flow matching API

Control-flow search does not place any additional requirements on the resulting Match objects.

5.8.3. Data-flow search

Data-flow search involves making statements about data as control moves through an application. Currawong's data-flow search support focuses on tracking *object identity*. Compared with systems that were written specifically to perform data-flow search, object identity tracking is quite limiting. Systems like Yang's EXE, for example, can also keep track of the values of all the data within the system under analysis [Yang et al. 2006]. However, restricted data-flow search abilities are less limiting for Currawong than one might expect, due to the nature of Currawong. Firstly, many viable optimisations simply do not require extensive data-flow search—as Section 5.4.1 discusses, often the requirement for data-flow search can be ameliorated by structural search combined with domain-specific knowledge of the system. Secondly, object identity tracking seems to address many types of real-world optimisation: the first two examples given in Section 5.4.2 can be verified by object identity tracking.

To track object identity, Currawong adds *objects* to the list of elements tracked by Match objects. In Java, almost everything is an object (apart from basic types, such

as `int`). In C, *objects* refer to pointers. Because C discards type information, object tracking in C is limited compared with object tracking in Java, but it is sufficient to track parameters passed to API functions.

```
MatchMethods is Java {
  class $ClassName extends android.view.View {
    void redraw()
    {
      $arg1 = api.function1();
      api.function2($arg2);
    }
  }
}
optimise(dataflow, App) is
  Match = App.match(MatchMethods),
  Match.arg1 = Match.arg2.
```

Figure 5.19. Data-flow matching example

Two types of pointer tracking are supported: to check for equivalence, and to determine access.

Figure 5.19 illustrates the pointer-tracking concept for equivalence. In this example, the template portion produces two variables: `arg1`, which is the result of calling `function1`, and `arg2`, which is the only parameter passed to `function2`. These two variables are then unified in the `optimise` clause (with the line `Match.arg1 = Match.arg2`. Currawong implements a custom unification process for this data type which triggers a pointer analysis.

The example highlights another advantage of equivalence tracking as a data-flow search method: it integrates well into the language. In CSL, equivalence tracking can be initiated by attempting to unify two variables from a Match object.

Checking for object access is initiated using a rule defined on the Match object, `Match.access/3`. This function produces a set of objects which use a particular pointer, starting from the scope object supplied as the first argument, and unifies it with the third argument.

The most important aspect of any implementation of data-flow analysis is that it is *conservative*, that is, if the analysis gives an incorrect result, it is always that the property being checked does not hold, when in fact it does. This conservatism ensures that optimisations are never applied when they should not be (at least, on the basis of data-flow analysis). If the reverse were true, then optimised applications could perform unexpectedly.

5.8.4. Transformation

Matching is one portion of a CSL specification. The other portion specifies the transformation to perform. Unsurprisingly, transformation portions of optimisation specifications

tend to reference the same code that was referenced by the match portion of the specification.

Section 5.5 described application transformation in terms of reference (identifying a portion of the application to be transformed) and application (applying the transformation). That section also outlined the types of transformation to be supported: architecture-level transformation, function and function-call renaming, and adding code.

The API for architecture-level transformation and function and function-call renaming is straightforward—a single rule for each expected action (such as renaming a function definition). This API is provided in Appendix B However, the API for adding code does not follow the same convention. Instead, Currawong supports *code merging*.

```
MergeOnDraw is Java {
  class $ClassName {
    private int _cw_tok;
    public void surfaceChanged(SurfaceHolder _,
      int _, int _, int _) {
      _cw_tok = au.com.nicta.cw.Draw2D.init(this);
    }
  }
}
...
Match = App.match(MatchOnDraw),
App.merge(Match, MergeOnDraw).
```

Figure 5.20. Adding code with Currawong

Code merging is used in the example provided in Figure 5.13. The relevant portion is reproduced as Figure 5.20 for convenience. The syntax is the same as that required for specifying templates in Currawong, and the new code can incorporate templated variables. The new code is then applied to the application in the context of a Match object. When applying code, all variables (that is, all tokens beginning with `$`) in the new code are replaced by the appropriate matched variables. Any class-level variables in the new code are then added to the matching class in the Match object. New functions are inserted into the class if necessary. In the example, a Match object is created from the `MatchOnDraw` template. Then, code specified in a template, `MergeOnDraw`, is added to the application, using the Match object to guide placement. Variables present in the Match object, such as `ClassName` are applied to the merge template—so `ClassName` is filled with the appropriate class. The field `_cw_tok` and the method `surfaceChanged` are then added to the class named `ClassName`.

Currawong cannot currently replace or remove code directly. So far it has been sufficient, when code should be replaced, to rename the function implementing the unwanted code, and then insert another function of the same name as the original first function. However, it does not seem infeasible to extend Currawong to support removal of code, should the need arise.

5.8.5. Summary

A fundamental part of Currawong's design is to provide support to the optimisation specification via an API which abstracts the details of checking and transformation as much as possible. Currawong's API supports the optimisation specification by supporting three types of match: structure search, control-flow search, and data-flow search. The API also enables the optimisation specification to transform code through simple refactoring-style manipulations, as well as via more complex code modification.

5.9. Transformation and looping

Architecture optimisation specifications may match within an application zero times, once, or more than once. Handling the zero-match case is simple: Currawong can leave the application unchanged. Handling the single-match case is almost as simple: Currawong transforms the application according to the match, and exits.

However, problems arise if a specification matches multiple times. The major problem is that of interference between matches. Perhaps a specification matches twice on the original application, but the process of transforming the application according to the first match invalidates the second match.

There are two ways to deal with this problem. The first approach, *optimisation-guaranteed noninterference*, is to require that optimisation specification writers guarantee that matches do not interfere with each other. The advantage of this approach is that all potential optimisations can be extracted from a single evaluation of the optimisation specification (making use of backtracking). A single evaluation pass means a faster architectural optimiser. However, optimisation-guaranteed noninterference puts a significant burden on the optimisation writer—arguably an inappropriate burden, she must now divert her attention from writing the optimisation and instead focus on correctly using the architectural optimiser's API.

The second way to deal with potential conflicts is to perform matching and transformation sequentially: take the first match, perform the appropriate transformation, and then check for new matches. This approach, *sequential application*, simplifies optimisation specifications—optimisation writers do not need to worry about conflicting matches, because the transformations effected by the first match will ensure that the second match does not occur. This means that the only requirement of optimisation writers is that they ensure that optimisation specifications only match when a valid optimisation can be applied. Since this requirement is the basic requirement of an optimisation in the first place, sequential application is a better alternative for optimisation writers. It is also simple to implement within architecture optimisers: the core optimisation pass can be implemented as a loop which repeatedly evaluates the optimisation specification, transforms the code according to the first result, and repeats, until the optimisation specification does not produce any result.

Sequential application may be improved in various ways. One option is to apply the best transformations, rather than simply applying the first one, where "best" means "results in the largest performance improvement in the rewritten application". This approach

is outside the scope of this dissertation.

5.10. Output

The final component of the architectural optimisation is writing changes to disk by actually modifying the application. Optimisation writers have no control over this process: if the optimisation succeeds, it is applied automatically. This leaves a lot of flexibility up to the implementation, regarding the way in which application rewriting will occur.

The exact process by which output is performed is left as an implementation detail, because it is system-specific.

6. Implementation

> On two occassions I have been asked, – "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> – Charles Babbage [Babbage 1864]

Currawong's implementation follows the design described in Chapter 5. It is a multi-platform architectural optimiser, supporting both the CAmkES software stack and the Android mobile operating system software stack. In addition, Currawong supports two languages—C and Java—and satisfies the criteria for an architecture optimiser in that it does not rely on application source code, supports multiple verification methods, and implements Currawong Specification Language.

Chapter 5 presented the design of Currawong's API, and then used this API design to motivate the design of Currawong specification language. In this chapter, an overview of Currawong is given first (Section 6.1), followed the implementation of the language (Section 6.2), and then specifics of the API: verification (Section 6.3) and transformation (Section 6.4). Verification and transformation are first described in a system-agnostic way, after which system-specific extensions for Java and Android (Section 6.5), and C and CAmkES (Section 6.6) are described.

6.1. Overview of Currawong

Currawong is a program mutator controlled by a domain-specific programming language. The programs it runs are optimisation specifications; the input to the program is an un-optimised application; and the output (if the optimisation was successful) is an optimised application.

Figure 6.1 shows the workings of Currawong from the perspective of the optimisation specification. From this perspective, Currawong provides an API to the optimisation specification. The specification makes use of this API to find a good place to apply the optimisation and to transform the application after locating a suitable place. Most of the API provided by Currawong to the optimisation specification is embedded within the Application object, which represents the application being examined. Interaction with the Application object results in the generation of a number of ancillary objects (such as the Match object). This high-level approach to optimisation keeps the specification concise.

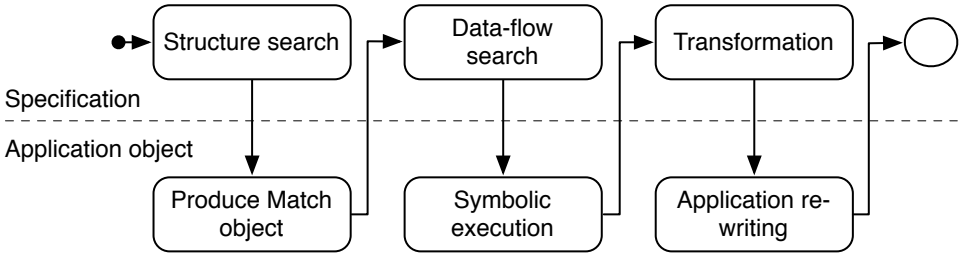

Figure 6.1. Currawong workflow (optimisation specification perspective)

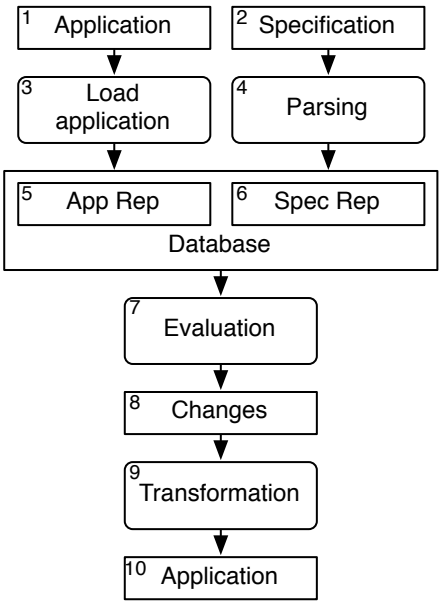

Figure 6.2. Currawong implementation (system agnostic version)

Figure 6.2 shows Currawong's architecture. Currawong is mostly written in the Python [Python Software Foundation 2010] programming language, but it makes use of several existing tools as part of its operation. Its design is based around that of a traditional logic-based application: the central data structure is a database of CSL clauses. Currawong execution proceeds as follows: firstly, the application (1 in the diagram) and specification (2) are both loaded (3 and 4). Currawong stores an abstracted form of the application, or app rep (5), as well as the parsed version of the optimisation specification, or spec rep (6), into a database. Here "database" is used in the Prolog sense, i.e., a searchable list of facts (statements about the program or specification) and rules (procedures to follow in order to deduce facts). Currawong then *evaluates* the spec rep (6). The result of the evaluation (7), if it is successful, is a set of changes to be applied to the application (8). These changes are applied (9), and the resulting new application (10) is written to

CSL type	Internal representation
atom	('atom', string value)
variable	('variable', string name)
structure	('structure', string name, terms...)

Table 6.1. Example type mappings, CSL to Python

disk.

Step 7, Evaluation, is where Currawong performs the optimisation work. Since Currawong specifcation language is declarative, "executing" the optimisation specification rule is equivalent to evaluating it (an example of a rule, `optimise/2`, was shown in Figure 5.13). In other words, the optimisation specification is presented as a search problem with zero or more solutions, where each solution represents a complete evaluation of the optimisation specifiation.

6.2. Implementing Currawong specification language

Currawong's Currawong specification language (CSL) support is built around a custom Prolog interpreter to which CSL-specific features were added. This method of implementation provided flexibility when designing CSL, because it is easy to extend the language—either by adding built-in functions, or by extending the grammar. For example, an early design experiment, later rejected, was to add support for constructing linear temporal logic terms as an integral part of the grammar. This approach provides flexibility at the expense of overall execution time—a suitable compromise for Currawong, but a non-prototype architecture optimiser would probably make use of an existing Prolog implementation for speed reasons.

Currawong uses a simple mapping between Python types and Prolog / CSL types. These mappings are used by both the interpreter and the parser. Each CSL type instance is represented internally by a Python tuple, where the first element is a string containing the type's name, and subsequent elements are the instance's value. Figure 6.1 shows a subset of the type mappings to demonstrate the idea.

6.2.1. Parser

CSL consists of three parsers: a parser for the core language, a parser for C templates, and a parser for Java templates. Each parser is generated by a backtracking LL(k) parser generator written by the author. Because both the generator and the parser itself are written in Python, the generator is well-suited to rapid prototyping: there is no compilation step. Changes to the grammar are automatically integrated into the grammar without a separate parser-building step, and adding code to execute when certain grammar constructs are encountered is seamless.

The core language parser produces, as output, a list of structures of the type shown in Table 6.1. Both the other parsers produce Python objects representing the parsed syn-

tax tree. The sytnax trees are deliberately kept opaque to the optimisation specification, which is required to use rules (i.e., call functions) on the Application object to investigate specific properties of the parsed source, rather than examining it directly.

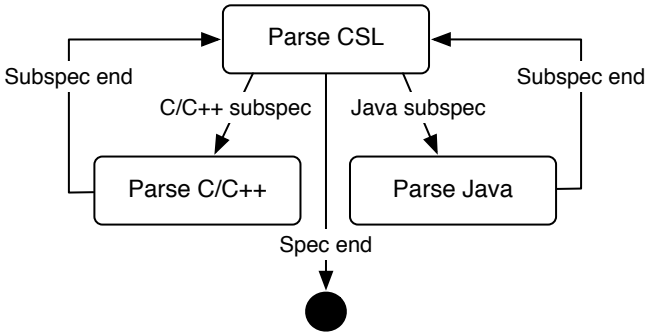

Figure 6.3. Control flow between CSL parsers

Figure 6.3 shows the interaction between the parsers. CSL requires that templated sections begin and end with braces ({ and }). This allows the CSL parser to treat templated sections within the specification as unparsed data. When such a template is encountered, CSL reads the braced portion, then calls relevant template-specific parser to produce a parsed representation. The object returned by the template parser is stored within the parsed representation of the CSL as a custom datatype. Note that this differs from the templating design outlined in Chapter 5: parsing of the template is performed when the specification as a whole is parsed, rather than at run-time.

6.2.2. Interpreter

Currawong executes optimisation specifications using an interpreter. This implements standard Prolog unification without the occurs check. Unification in this version of Currawong is implemented using Python *generators*, which are essentially co-routines. The core interpreter is very small (approximately 900 LOC, including comments), which aided debugging.

Besides implementing support for objects (as described in Section 5.7.2), the interpreter is extensible: additional built-in functions may be written (in Python) and made available to the interpreter; the interpreter in turn makes these functions available to the optimisation specification as built-in rules.

6.3. Matching

Application representations are exposed to the specification through an API: no attempt is made to give the specification access to the internal representation of the parsed application. The advantage of this approach is that both the internal representation, and conse-

quently the static analysis methods which operate on that internal representation, can be modified without creating incompatibilities for existing optimisation specifications.

Matching is highly language-specific. Therefore Currawong includes two implementations of structure search, control-flow search, and data-flow search (the three matching types, as discussed in the previous chapter): one for Java, and one for C. The Android-specific portion verifies bytecode compiled from Java source, and the CAmkES-specific portion verifies code compiled from C source. The common features of these implementations are discussed below. The Java- and C-specific portions are discussed in Sections 6.5 (Android) and 6.6 (CAmkES), respectively.

6.3.1. Structure search

A structure search matches a *template* to a portion (or portions) of the application. Chapter 5 outlines this process in general terms.

In Currawong, the presence of a template in the optimisation specification causes Currawong to create a custom data-containing object representing that template as matched against portions of code. Creation of this object is triggered by evaluation of an `Application.match` rule as discussed in Section 5.8.1. One template can therefore result in the creation of multiple objects over the course of evaluation, if it matches multiple portions of code.

The `match` rule is syntactically translated to `application_do_match(Application, Templatename)` (The Application object does not make use of the macro feature). When this rule is itself evaluated, a Match object is prepared by the appropriate (Java or C) application representation and returned.

Section 5.8.1 outlines the types of things that a structure search should produce: structural information, the locations of function calls within any matched methods, and any variables included within the structure. Match objects therefore contain methods to access these data.

Referencing structure information and function calls

As described above, structural information provided by a Match object is made available to the specification language through the `Match.feature/1` rule (see Appendix B for a complete list of Match object rules). This rule accepts a miniature domain-specific language to reference parts of the Match object's structure.

The feature specification language allows one to identify a particular structural portion of code. Figure 6.4 shows some examples of the specification language in action. Here for `Match.feature` declarations are shown, each referring to the template `MatchMethods`, above them. Note that the numbered lines are explanatory and are not a part of the specification language.

The optimisation author uses the feature specification language to identify a feature by specifying one or more scope names (i.e. module, class, or function names, separated by a dot). The default scope is assumed to be the scope of the highest-level container within the match; in Figure 6.4 this is the class `$ClassName`.

```
   MatchMethods is Java {
     class $ClassName {
       void method()
       {
         dosomething();
       }
     }
   }
1  Match.feature("method")
2  Match.feature("")
3  Match.feature("$ClassName.method")
4  Match.feature("_.method")
5  Match.feature("method.dosomething")
```

Figure 6.4. Using the ".feature" rule to access a Java method

If the Match object cannot locate the specified feature, it tries again from the scope above the current default scope: i.e. if the default scope is currently a class, the search is retried with the default scope set to the scope that contains classes, i.e., in Java, the package in which the class is defined. Therefore examples 1 and 3 in the figure are equivalent—both match the method named `method`, but the latter example does it after first attempting, and failing, to locate the contained scope within the class.

Two additional features are provided: templated variables may be used as part of the specification, as `$ClassName` is, above; and a "scope name" of a single underscore character (`_`) may be specified as a wildcard to indicate that any name is acceptable.

This mechanism is extended to describe the location of function calls within a match: the name of the function call may be specified as an additional scoped name after a function scope is matched. Support for referencing additional calls to the same function after the first one is currently not implemented; a simple solution is to extend the templating syntax to allow the template writer to uniquely name each call, so that the correct one can be specified unambiguously within the feature specification language.

The result of evaluation of a `Match.feature` rule is another custom datatype: a *match reference*. Match references are made available to the optimisation specification as opaque types, so that they may be passed as input parameters to transformation rules. The details of the reference are, however, hidden from the specification language. A match reference must provide sufficient information to the rest of the system so that it can be used as part of a transformation. There are four types of match reference required for structure search:

1. *scope* match references identify a particular scope, that is, a module, class, function, or similar;

2. *invocation* match references identify a particular function invocation.

3. *data* match references identify a particular variable within a function.

4. *variable* match references identify variables matched in the template (that is, names which were prefixed with a dollar sign, such as `$ClassName`).

These reference types are object-language-dependent, so their implementation is discussed below. In general terms, scope-related match references refer to a specific class in the hierarchical application representation built after scanning an application, whereas invocation references include a function-level scope match reference as well as some way of indicating the particular invocation within that reference (in both cases, an offset from the beginning of the function to the relevant instruction is used).

Variable match references are a special case. Variable match references are made available to the optimisation specification using the mechanism described in Section 5.7.2—i.e., evaluation of `Match.VariableName` results in immediate evaluation of `Match_match(Object)`, where `Object` is the custom object representing the actual variable reference. Because, however, variables can be specified at various different locations within the template, one of two different types of variable match reference is returned, depending on the semantic role of the variable within the matched template:

1. If the variable refers to a *scope* (i.e. if the variable is a class name in a Java template) then the result of evaluation of the variable is a scope match reference.

2. If the variable refers to a function parameter then the result of evaluation of the variable is a data match reference.

6.3.2. Control-flow search

No additional support is required of Match objects in order to implement control-flow search. To support the API described in Section 5.8.2, match objects must reference the previous (in terms of control flow) match object. Actually determining the path from one Match object to the next is up to the language-specific application representation.

6.3.3. Data-flow search

Currawong supports data-flow search with two symbolic execution engines: one for Java, and one for C. As discussed briefly in Section 5.4.2, symbolic execution is a type of execution method in which all possible values for a given control-flow path are simultaneously considered [King 1976]. In order to achieve this, regular variables within a program are replaced by *symbolic variables*, which can represent a set of values. In the case of integers, a symbolic variable may represent a range of numbers. In the case of pointers, a symbolic variable may represent a range of locations. The set of all symbolic variables along the current execution path is known as the *path condition* or *path constraint*.

When a control-flow-modifying instruction is encountered, the path condition must be checked to determine in which direction execution should continue. Consider the code in Figure 6.5. When the `if` statement is reached in a normal execution, control flows in exactly one of two directions depending upon the value of `a`. However, under symbolic execution, there is an additional possibility. If the value of `a` is not known,

```
int func(int a) {
  if (a < 3) {
    return a;
  } else {
    return 0;
  }
}
```

Figure 6.5. Data-dependent control flow modification

then the symbolic execution engine does not know which path should be taken. The only safe possibility is to explore *both* paths. This is known as an *execution fork*: the symbolic execution engine first updates its path condition to include the assumption that `a < 3`. It then executes the *true* path of the if statement. However, the symbolic execution engine must also investigate the *false* path of the if statement: the path condition is in this case updated to include the assumption that `a >= 3`. Thus as control-flow statements are encountered the amount of knowledge about variables in the program increases.

Implementing a complete symbolic execution engine, let alone two, is a large research project in itself. Currawong therefore includes proof-of-concept engines capable of performing the task described in Section 5.8.3—that is, determining whether parameters passed to one API function represent the same object as those passed to a previous function. The advantage of using symbolic execution even in this constrained context is that additional data-flow verification support can be added to Currawong without requiring modification of any optimisation specification.

Currawong's data-flow search support does not require any additions to Match objects. It makes use of data match references (supplied from a Match object resulting from a structure search).

6.4. Supporting code transformation

Code transformation involves both reference and application, as discussed in Sections 5.5 and 5.8.4.

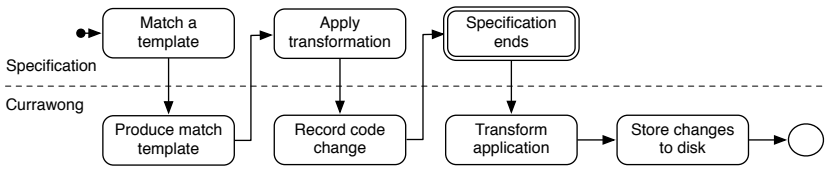

Figure 6.6. Code transformation process

Figure 6.6 shows the transformation process. The specfication begins by performing a template match using a `match` rule. This causes Currawong to produce a Match template

relating to the match. The specification then applies one or more transformations, which are recorded by Currawong. After applying transformations, the specification ends. At this point, Currawong transforms the application by modifying its code and meta-data.

Reference is achieved through Match objects, using the four match reference types described above. When transformation rules are evaluated, objects representing the transformation are created. These objects are created by side effect when the rule is evaluated. If the optimisation specification is evaluated successfully, all transformation rules are used to rewrite to the application in a language-specific way.

Objects which describe transformations contain the following information:

- A description of the type of transformation: for example, to rename a function;
- A reference to a match object to which the transformation should be applied;
- Any transformation-specific information: the new function name, in the case of a function-renaming transformation.

6.5. Android-specific portions

In this section, the name Currawong Java is used to refer to Currawong with Android and Java extensions.

Android applications are written using a standard Java development environment consisting of the standard Java compilation tools (such as `javac` [Oracle 2010]) and the Eclipse IDE [Eclipse Foundation 2010a]. Applications for Android are compiled to a set of `.class` files, as is standard practise. After compilation, however, each `.class` file is translated from Java bytecode to *Dalvik bytecode*, a custom bytecode supported only by the virtual machine supplied as a core portion of Android, *Dalvik* (Figure 3.2). This translation is apparently performed due to licensing issues, rather than for any technical reason. (Dalvik files are designed to be more efficient than `.class` files on mobile hardware, but this translation could instead be performed on the mobile device—indeed, Dalvik files still do undergo further optimisation after installation on a device.)

After translation, the bytecode files—retaining the Java convention of a single file per class—are combined into a single uncompressed archive, named classes.dex ("dex" is a shortened form of "Dalvik Executable"). This file is itself placed into an archive, along with any resources required by the application (such as sound files or images). This archive, called an Android Package (APK), is signed by the developer, and can be installed onto an Android device.

Figure 6.7 shows Currawong with extensions for Android. Currawong is applied to APK files. To apply an optimisation, Currawong first extracts the Dalvik bytecode from the APK. It then disassembles the bytecode and parses it to create an application representation. The optimisation specification is evaluated and, if it results in changes to the application, Currawong modifies the bytecode files directly, either by adding classes, or by modifying already-present classes. Finally, Currawong re-generates the APK. In summary, Currawong includes the following extensions to handle Android applications:

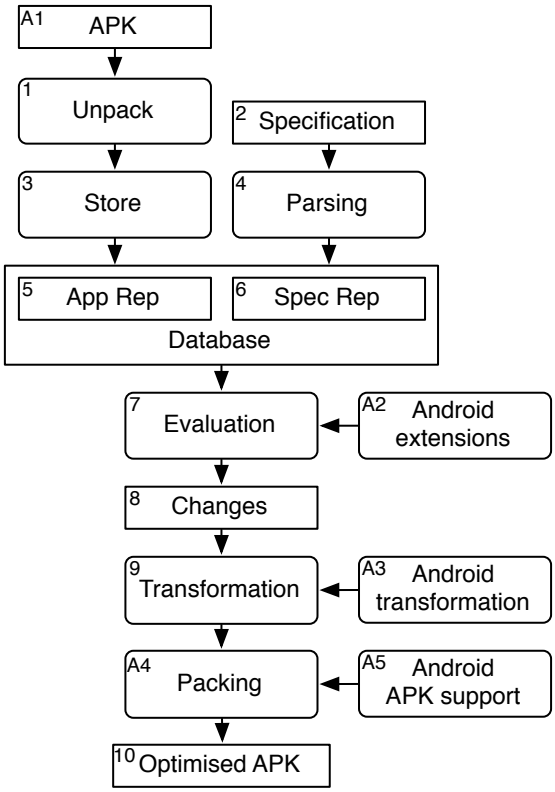

Figure 6.7. Currawong implementation (Android extensions)

1. Unpacking and disassembling, and subsequently assembling and re-packing (A1, A4 and A5 in the figure);

2. Creation of application representation, and support for static analysis of Android-specific byte codes (A2);

3. Support for transformation of Android applications (A3).

Figure 6.7 shows a high-level overview of this process. The Android-specific portions are discussed in detail below.

6.5.1. Unpacking, disassembling, and reassembling

Access to Dalvik bytecode is achieved through third-party utility programs. The APK file itself is a `zip` archive. Currawong uses the Dalvik bytecode disassember Baksmali, and its corresponding assembler, Smali, to decode and encode Dalvik VM bytecode [JesusFreke 2010]. Invocation of these utility programs comprises the bulk of the work performed by the "Unpack" stage in Figure 6.7.

```
.method public constructor <init>()V
  .registers 1
  .prologue
  .line 38
  invoke-direct {p0}, Landroid/app/Activity;-><init>()V
  return-void
.end method
```

Figure 6.8. Baksmali's disassembly of an automatically-generated constructor

Figure 6.8 shows an example of the code generated by Baksmali. This is an automatically-generated constructor method for an Android application. In Baksmali disassemblies, lines beginning with a dot control the assembler, or contain comments, so there are only two actual instructions in this snippet: `invoke-direct` and `return-void`. This constructor method simply calls the constructor of the method's class' parent, via `invoke-direct`, and then returns to the caller, via `return-void`. Notably, this disassembly retains a lot of information about the original code. Parameters and return values are enumerated and typed, and function invocations are specified by name.

In the "Store" stage, Baksmali's output is parsed by a custom parser, a process facilitated by the regular structure of the disassembly: each file produced by the disassembler represents a separate Java class. These are read from disk, and an in-memory hierarchy of objects, mirroring the namespace set out in the application, is created. The outermost level, Application, contains one or more Package objects, each of which contains Class objects. These in turn contain Fields (class-level global variables) and Methods. This hierarchy is illustrated by Figure 6.9. This figure shows Currawong's in-memory object hierarchy for a single class, containing one method and one field. The particular class shown is taken from the Lunar Lander example game provided with the Android software development kit.

6.5.2. Application representation

Figure 6.9 also shows the way the disassembled Java classes are represented internally. Each method contains within it both the original code and a *simulation*. The original method is used for structure and control-flow search. It is also used for rewriting purposes. The simulation is used for data-flow analysis (Section 6.5.3).

6.5.3. Matching

Currawong Java implements structure, control-flow, and data-flow search. Currawong Java produces Match objects that directly reference both the application representation and the template object.

Template objects are produced by the Java-specific parser. Internally they are a nested set of objects corresponding with the template in the optimisation specification. Objects used to represent a template include *scope objects*, which represent scopes such as mod-

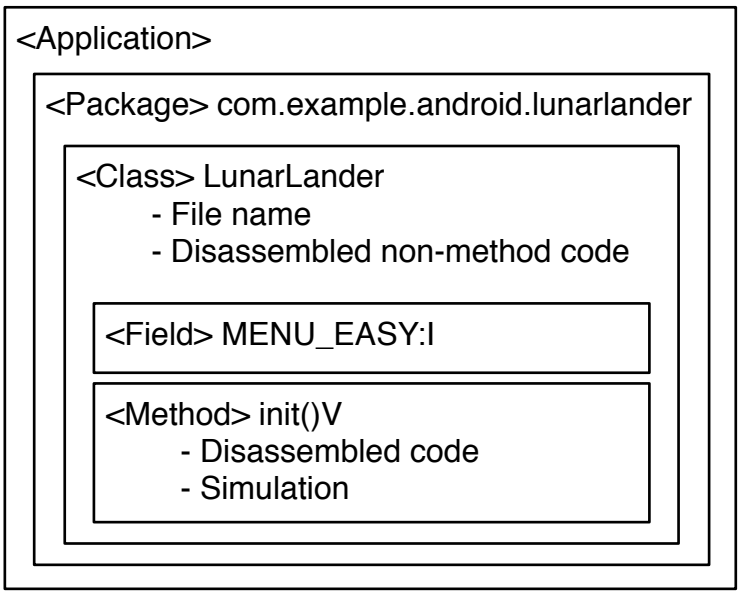

Figure 6.9. In-memory class hierarchy for a portion of the Android Lunar Lander game

ules, classes, and functions; *call objects*, which represent function calls; and *parameter objects*, which represent parameters to function calls. All template objects contain a scope object as the outermost object. For example, if the first level of the optimisation specification is a `class` declaration, the first scope object represents a templated class. Each object has domain-specific features:

- Class scope objects include references to their name and any classes they extend;
- Function scope objects include the function name, return type, and a list of parameters;
- Call objects include the called function's name, the return type, and a list of parameters;
- Parameter objects include the parameter's name and type.

An associative array mapping variable names to appropriate templated objects is constructed and associated with the template.

Structure search

The goal of structure search is to produce the four types of match reference required by the optimisation specification for verification and rewrite purposes. These reference types, described in Section 6.3.1, comprise references on scopes, invocations, data, and variables.

Currawong supports scope match references on class names and function names. To find matching scopes, Currawong recursively descends the scopes of the application representation (Figure 6.9) until it finds a scope which matches the outermost templated scope. At this point, Currawong produces a Match object. The match object is filled with information in the following way:

1. Each nested scope in the template is checked against the appropriate nested scope in the application. If a scope (i.e. a class or a function declaration) is specified in in the template but is not found in the application, the Match object is discarded and structure search at this location fails.

2. Whenever a child scope is encountered, its description is added to the Match object. Class scopes and method scopes are currently added.

3. Invocation and data match references are added to the appropriate scopes. To discover these references, the dissassembled method code is scanned, and method calls are detected.

4. When scope references, invocation references, and data match references are added, a check is made against the template to see if the relevant portion of the template makes uses of a variable match reference (that is, a name prefixed with a dollar sign). If a variable match reference is present, an entry is created inside the Match reference, referring to the appropriate scope, invocation, or data match reference.

Control-flow search

Control-flow search operates at the level of functions and is based on a control-flow object called the *reachable set*. To build the reachable set for a given control-flow search the following algorithm is used:

1. Start with an empty set named reachable-from, which lists scopes which can be reached from the calling scope through function calls; and an empty queue, the *work queue*.

2. Add the start object, which should be a function-level scope, to the work queue.

3. Take an item from the work queue. Add the item to the reachable-from set.

4. Get a list of outgoing function calls from the item.

5. For each outgoing function call, find the appropriate scope object in the application representation and add it to the work queue, as long as it is not already in reachable-from.

6. Continue until the work queue is empty.

7. The resulting set reachable-from consitutes the set of scopes accessible from the start object.

The reachable set is used in control-flow search. As described in Section 5.8.2, control-flow search requires a template object indicating the target of the match, as well as a "predecessor" match object indicating the object prior to this one.

To perform control-flow search, a reachable set is built where the start object corresponds with the function-level scope of the predecessor Match object. A structure search is then performed using the supplied Template. The result is accepted if and only if the resulting top-level scope is in the reachable set.

Data-flow search

To construct the simulation of a method, Currawong begins by examining each instruction in the disassembled code. Instructions are similar to assembly language: each instruction is composed of an *opcode* and zero or more arguments, where an *opcode* represents one of the instructions in the Dalvik virtual machine's instruction set. For each instruction in the dissasembly, Currawong creates a *simulation* of that instruction. An instruction simulation consists of a reference to a function which implements the behaviour of the instruction.

To perform data-flow analysis, Currawong uses symbolic execution. The implementation of symbolic execution in Currawong does not improve meaningfully on any existing symbolic execution tool—in fact, in many ways it is signficantly less well-developed. This is not a disadvantage: the proof-of-concept symbolic execution support implemented here is sufficient to demonstrate that symbolic execution is a reasonable option for data-flow analysis in Currawong, and is sufficient to verify optimisations in simple examples (see Chapter 7).

To perform symbolic execution, a *symbolic register set* is constructed of appropriate size for the matched function. Each opcode is then examined in the function simulation in sequence. The classes of supported opcodes, and their effect on registers, is described below.

Move instructions: copy registers. Registers are copied without regard to their contents.

Return instructions: terminate the execution.

Constant declarations: set the value of a given register to a concrete value.

Object creation functions: set the value of the given register to the unknown value (in symbolic execution terms, the "unknown value" indicates that a given variable could have any possible concrete value).

Comparisons: see below.

Array and object field access functions: set the value of the given register to the uknown value.

Invocations: set the value of each affected register to the unknown value (calls are not followed).

Other opcodes: are ignored.

Comparisons always involve two registers. When a comparison is encountered, the engine must decide which control flow path to take. The decision is made according to the following rules:

- If the comparison does not involve integers, examine both paths.
- If both operands are concrete, perform the comparison concretely and take the appropriate single path.
- If one or both operands are symbolic, attempt to decide which path should be taken, by first attempting to prove that the comparison is true, and then by attempting to prove that it is false. Currawong makes use of the `python-constraint` constraint solver in order to make this decision [Labix 2010]. If the truth or falsity of the comparison can be proved, that path is omitted from the execution.

6.5.4. Transformation and output

Transformation in Currawong Java consists of performing refactorings, and adding code.

Performing refactorings

Refactorings, such as method renaming, are performed on the disassembled application. Both method definitions and method invocations have distinctive, fully-typed signatures which can be recognised via purely-textual means. Modifications are made in-place to the in-memory representation of the code.

Adding code

Currawong adds code indirectly, by adding packages to the application, and directly, through the use of the `Application.Merge` rule. Indirect addition of code is a trivial process: no modification to the application is needed, because Java includes a mechanism for packages to self-initialise when they are loaded. Currawong therefore simply copies the named package to the application's APK file when re-building.

Currawong supports translation of a small number of Java instructions to Smali assembly language in order to implement direct addition of code to modules. Currently supported is the ability to add fields to classes, to implement class variables, the ability to add functions to classes, and the ability to call functions.

Generating output

Once the application has been rewritten in memory, Smali assembly language files for each class are written to disk. Currawong uses the Smali assembler to convert these files to a `classes.dex` file suitable for Dalvik.

The final task Currawong must perform is to sign the code. Android preserves application integrity through code signing. When a class has been modified, the application must be re-signed before it can be used. Currawong can apply a signature automatically—currently a special application developer signature is applied. This re-signing has negative practical implications. In particular, updates to the application cannot be automatically applied. A simple potential solution to this problem is to provide a way for the optimiser user to indicate to the update mechanism that the signature mis-match is due to application of an optimisation, and thus the application should still be considered for updates.

6.6. CAmkES-specific portions

In this section, the name Currawong C is used to refer to Currawong with CAmkES and C extensions. In CAmkES, a system is specified using a domain-specific Architecture Definition Language (ADL). ADL determines which components comprise a system, and how the components should communicate. Figure 6.10 shows an overview of Currawong C.

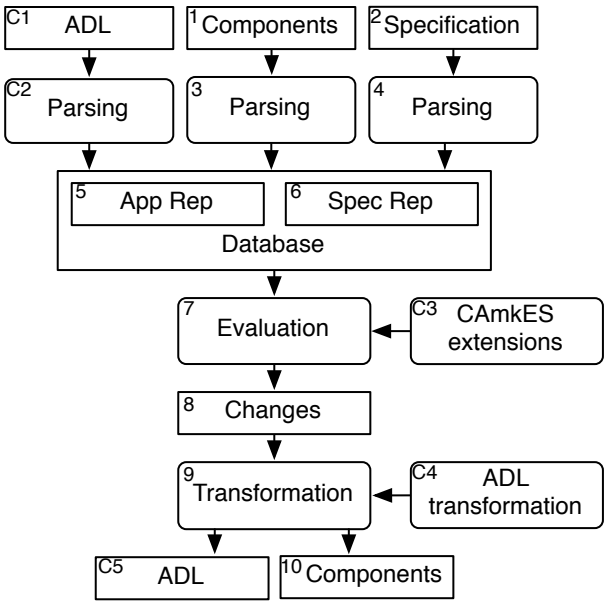

Figure 6.10. Currawong implementation (CAmkES extensions)

Unlike Android applications, CAmkES applications (more specifically, Binary CAmkES applications) are compiled directly to machine-specific code. This means that an architecture optimiser must be capable of reading the binary files which contain objects, performing verification of these objects, and changing their behaviour.

The additions to Currawong to support CAmkES and C are labelled with the letter “C” in the figure:

1. ADL (C1) is parsed (C2) in addition to the usual component loading;
2. Determining the suitability of an optimisation includes extensions for verification of CAmkES components (C3);
3. Transformation requires domain-specific knowledge of the ADL (C4 and C5).

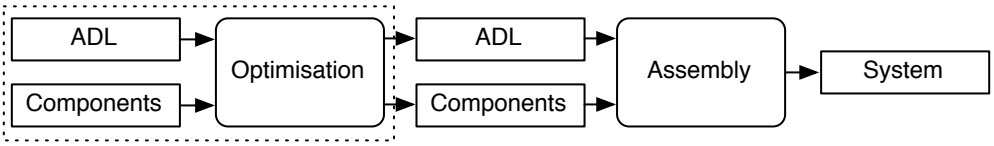

Figure 6.11. The CAmkES assembly process (Currawong is inside the dotted portion)

Figure 6.11 describes the role of Currawong with respect to the CAmkES system assembly process. CAmkES acts as a type of linker, combining component implementation files according to the description in the ADL. Currawong, indicated by the dotted section in the diagram, executes before the CAmkES assembly process occurs.

Currawong is currently limited to supporting the ARM processor, because this is the processor in the test environment used to evaluate Currawong (more information above the particular enviroment is given in Chapter 7).

6.6.1. Unpacking and application representation

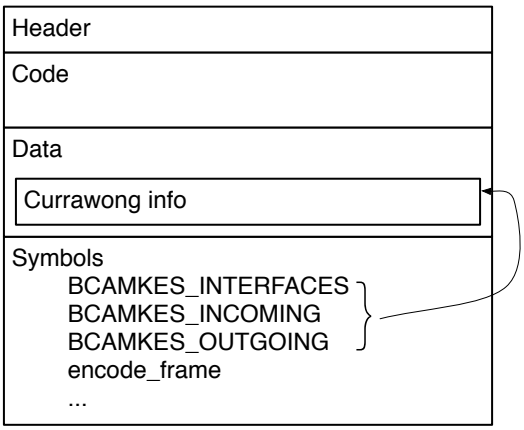

Figure 6.12. A Binary CAmkES component (ELF file)

Binary CAmkES components are stored as object (`.o`) files in ELF format [TIS Committee 1995]. Each component object has a fully-qualified name using a reverse-DNS scheme. That is, a component author assigns a name to her component by coming up with a short name, appending that to the name of domain name under her control, and reversing

the result. For example, the `codec` component produced by the owners of `example.com` would be named `com.example.codec.o`. This naming scheme provides a global guarantee of uniqueness.

CAmkES stores information relevant to the component system in the component files themselves. Figure 6.12 illustrates the additional information stored by CAmKES inside component files. Each component includes three extra symbols, BCAMKES_INTERFACES, BCAMKES_INCOMING, and BCAMKES_OUTGOING. These reference data also stored within the object file. These three symbols define the component's *interfaces*, as described in Section 3.1.

Currawong C builds an application representation by examining the object file's symbol table. The symbol table provides a list of functions defined by the component, as well as a list of functions the component expects to be able to call. ELF files can contain multiple symbol tables. Typically, components contain at least two: an information-rich symbol table, suitable for static linking and for debugging; and a minimal symbol table, suitable for dynamic linking. Currawong uses this latter table only, as the former one may not be present.

6.6.2. Matching

Currawong C, as with Currawong Java, supports structure search, control-flow search, and data-flow search.

Production of Match objects follows a process similar to that of Currawong Java, but the method through which information is gathered about a component differs. These differences vary depending on the type of verification being performed.

Structure search

Currawong C supports the four types of reference described in the system-agnostic portion of the design, in Section 6.3.1. However, in addition to examining individual components in order to determine structure, Currawong C also examines the ADL specification describing the component system's layout. As discussed in Chapter 3, ADL describes a component-based system in terms of components and the connections between them.

The following two scopes are supported:

- Component scope: This is a CAmkES scope. As discussed in Chapter 3, each component in binary CAmkES has a unique name. A simple syntactic transformation of the component's file name is used as the component's fully-qualified name.

- Function scope: This is a C scope. To find functions within a file, the component object's symbol table is scanned. This produces both a list of function names (for matching purposes) and a set of pointers to binary code (for symbolic execution).

Currawong produces invocation match references for C code by scanning the binary code discovered during scope search. Specifically, Currawong searches the code for instances of the `bl` and `blx` (branch and link) instructions, which are used to call functions.

Currawong only searches for branches to functions which are mentioned in the component's symbol table. The intent behind this approach is that functions referred to in the component's symbol table are those functions which are externally-visible or implemented in other modules—that is, functions which comprise the publically-accessible API of the component, or which comprise the API of another component upon which the present component relies.

Data match references are produced from the list of invocation match references as well as the list of function scopes.

Control-flow search

Unlike Currawong Java, which builds a separate reachable set for control-flow search, Currawong C uses its symbolic execution engine to perform control-flow search. This is because component binary files do not contain enough information to trivially build a reachable set. In practice, this is not an issue, because the only control-flow search that is required of binary components tends to be in the context of data-flow analysis.

Data-flow search

Like Currawong Java, Currawong C implements a proof-of-concept symbolic execution engine. The start and end points of execution are defined by invocation match references. Currawong implements a somewhat abstract machine model: registers are modelled, but memory accesses are not, with the result that loads from memory result in the associated register being assigned the unknown value. This approach works for small loops and for consecutive functions which are not separated by a large amount of code (and for which, therefore, data remain in registers).

As per the Java implementation, Currawong C models integers, which are normally the basis for loops. It can also determine whether a pointer returned from (or passed to) one function is the same as a pointer returned from (or passed to) another function. However, Currawong's implementation cannot track more complex properties, such as "the data in a specific region has not changed between two calls".

6.6.3. Transformation and output

Currawong C implements transformations in a similar way to Currawong Java, by storing the set of transformations and applying them when evaluation of the optimisation specification succeeds. However, Currawong C supports a different set of transformations to Currawong Java:

Renaming functions: Currawong C renames functions by modifying the symbol table of the component defining the function, i.e. without modifying code in any way.

Renaming function calls: A new symbol is added to the affected component's symbol table. This symbol is marked as not defined within the component object. Code referencing the old symbol (containing the original function name) is updated to reference the new symbol.

Interposing components: The new component is added to the ADL, and connections described by the ADL between the two original components are modified to include the new, interposed component.

After modification, Currawong writes new versions of the components and ADL to disk.

6.7. Example Java optimisation

This section describes a Java-based architecture optimisation. Consider a callback-based event-processing API for mouse movements. To use this API, applications provide a class that implements a special interface named `MouseEventHandler` that provides mouse movement events. When the API has finished processing the event, it calls the special method `next()`, which informs the event processor that the application is ready to receive more events. An example of code making use of this interface is shown in Figure 6.13.

```
public class Handler implements api.MouseEventHandler {
  public void mouseEvent(Context c, Event e) {
    ...
    c.next();
  }
}
```

Figure 6.13. Example application using the MouseEventHandler API

Suppose that a more efficient API was created, which would only be used by applications which always called `next` in the function that handled the mouse event. To use the new API, applications should call `efficient_next` rather than `next`.

```
MatchEvent is Java {
  class $ClassName implements api.MouseEventHandler {
    public void mouseEvent(Context $Context, Event $Event) {
      $Context.next();
    }
  }
}
optimise(mouseevents, App) is
  Match = App.match(MatchEvent),
  App.rename_call(Match.feature("$ClassName.mouseEvent"),
      "$Context.next", "$Context.efficient_next()").
```

Figure 6.14. Optimisation specification for the MouseEventHandler optimisation

Figure 6.14 shows the optimisation specification. The Match template closely resembles the code to be matched. The optimisation specification first performs the match, and

then renames the method call.

To perform this optimisation, the following steps take place:

1. Application representation: The application is loaded and an application representation is generated;
2. Match object generation: A Match object is generated corresponding to the MatchEvent template in Figure 6.14;
3. Method renaming: The method is recognised and renamed;
4. Application rewriting: the application is modified and written to disk.

These steps are discussed individually.

6.7.1. Application representation

The application is supplied as an Android `.apk` file. This file is unpacked and disassembled. Figure 6.15 shows the resulting scope object.

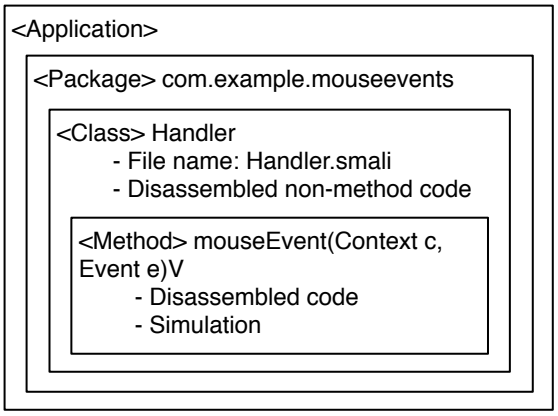

Figure 6.15. The MouseEventHandler example application, internal representation

The disassembled code is stored within the scope object. Figure 6.16 shows the disassembly for the function shown in Figure 6.13.

6.7.2. Match object generation

Currawong generates a Match object according to the procedure described in Section 6.3. The resulting Match object for this application is shown in Figure 6.17. In this figure, numbers within angle brackets represent distinct objects. The first portion of the Match object resembles the scope hierarchy of the actual Application, and is used to provide support for the `Match.feature/1` rule. The second portion of the Match object contains all the variables supplied in the template, as well as the corresponding classes to which they relate.

```
.method public mouseEvent(Lcom/example/Context;Lcom/example/Event;)V
  .registers 3
  .parameter "c"
  .parameter "e"
  .prologue
  .line 6
  invoke-virtual {p1, p2}, Lcom/example/Context;->next(Lcom/example/Event;)V
  .line 8
  return-void
.end method
```

Figure 6.16. Disassembled code for the MouseEventHandler example application

6.7.3. Method renaming

Once the Match object has been generated, method renaming can take place. Evaluation of the rule results in an object being added to the logic database which contains data similar to that shown in Figure 6.18. Note that the numbers in angle brackets here refer to object references contained within the Match object of Figure 6.17.

6.7.4. Application rewriting

When the optimisation specification has been completely evaluated, the application is rewritten. In this case, the rename-call Modification object is examined. To rename the call, the dissasembled method is modified in-place. Currawong knows which line of the disassembly to modify, because it is contained within the Call object, which itself part of the Match object, which is referenced from the Modification object.

The rewritten assembly language code is written back to disk, the application is assembled using the Smali assembler, the resulting file is added to an Android application package (`.apk`), and the new package is signed.

6.8. Discussion

This implementation of Currawong satisfies the requirements of an architecture optimiser as described in Chapter 5: it supports multiple languages, operates on binary files, and is capable of applying architecture-level optimisations after applying a number of techniques to ensure that the optimisation will behave as expected. Although it is a prototype implementation, a number of powerful techniques are supported.

Currawong uses symbolic execution to implement data-flow verification. The advantage is that the engine can be continuously extended and improved without requiring modifications to the optimisation specification. However it is by no means the only choice. One interesting alternative is to support data-flow verification at run-time: data-flow conditions would be compiled to the target language (Dalvik or binary code) and inserted as verification conditions before the transformed code. The advantage of this approach is

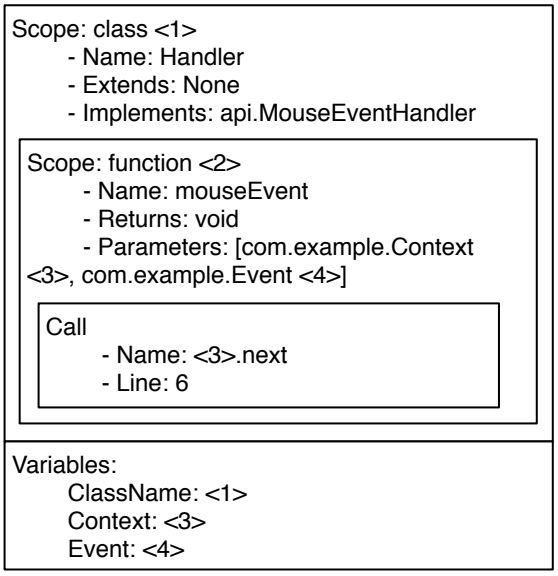

Figure 6.17. The MouseEventHandler match object

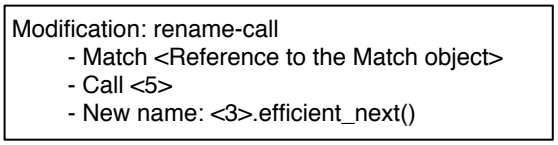

Figure 6.18. The method rename description

that data-flow verification that would be difficult or impossible to perform statically could be possible if performed dynamically. One can imagine an architcture optimiser making use of a hybrid scheme: static analysis where possible, falling back to dynamic analysis.

Even if Currawong continues to use symbolic execution, a strong case can be made that Currawong should use existing symbolic execution engines, such as Java PathFinder [Havelund and Pressburger 2000] or EXE [Cadar et al. 2008]. This suggestion is difficult to argue with: using an existing execution engine would give Currawong better data-flow search capabilities. Unfortunately, integrating two existing symbolic execution engines was ultimately rejected due to a lack of readily-integrable code. Although many engines exist for Java code, none exists for Dalvik code. Converting from Dalvik to Java, while certainly achievable, is non-trivial (Dalvik is register-based while Java is stack-based, for example). Similarly, several symbolic execution engines exist for C source code, but the author found no publically-available symbolic execution engines for ARM binaries while implementing Currawong. Recent (2009) work by Chipounov et al. indicate promising work in the area of symbolic execution on the QEmu system simulator [Chipounov et al. 2009]. Their work is currently limited to the x86 architecture, but

could be extended to other architectures with relative ease.

Despite the potential for improvements to Currawong's data-flow analysis, the current system as is a successful capable of an architecture optimiser, performing a wide variety of architecture optimisations in two languages and across two entirely different systems. The next chapter demonstrates Currawong's capabilities as applied to a variety of CAmkES and Android-based applications.

7. Evaluation

> In considering any new subject, there is frequently a tendency, first, to overrate what we find to be already interesting or remarkable; and, secondly, by a sort of natural reaction, to undervalue the true state of the case.
>
> – Ada Augusta [Menabrea and Augusta 1842]

Can Currawong be used to optimise real systems? In this chapter, Currawong is applied to applications running on two very different systems: a strongly component-oriented research system and a real-world system. The results show that Currawong is capable of high-performance architecture optimisation of both systems.

7.1. Introduction

An architecture optimiser must meet the design guidelines outlined in Section 5.1: that is, it should recognise anti-patterns, apply remedies, work without source code, and support multiple languages. Chapter 6 described Currawong, which meets these criteria. Meeting design goals is not the same as being effective, however. This chapter demonstrates Currawong's effectiveness at performing real architecture optimisations.

This chapter has three goals. The first goal is to demonstrate that Currawong can perform a variety of optimisations across multiple systems. To demonstrate this, Currawong is applied to two systems. The first system implements the architecture described in Section 3.3.1, the componentised video player, on the CAmkES component-based system. The second system is the Android mobile operating system: optimisations are are applied to a variety of Android applications. This division demonstrates Currawong's performance both in an ideal componentised system (CAmkES) and on publically-available, commercial-quality code (Android).

The second goal of this chapter is to demonstrate that the type of optimisations that Currawong can perform are worthwhile. Unfortunately there is no simple answer to the question "what degree of performance improvement is worthwhile?" The answer depends on the application and on the platform. For example, a 5% performance improvement to the main loop of a fast-paced action game may be worthwhile simply in terms of increased run-time on a battery-operated device; but a 5% performance improvement to the configuration panel of an application is probably not worthwhile.

Nonetheless, I set some intuitive lower bounds for whether or not a performance improvement was worthwhile for these tests. All performance improvements measured here

are active for the majority of the application's runtime—componentised video player performance improvements are active whilst playing video; Android performance improvements are active for the entire duration of the application. Furthermore, most of the performance improvements benchmarked here apply to applications which display video, and result in reducing the amount of data, specifically image data, that must be processed (either by copying, or by transformation). Thus most of the performance improvements here would improve the frame-rate of the application, if it were CPU bound; or reduce the application's CPU usage, if it were not. A 10% performance improvement for a CPU-bound application running at 30 frames per second could increase the frame rate by three frames per second. Intuitively, this seems to represent the lower bounds of "worthwhile".

The third and final goal of the chapter is to demonstrate the limitations of Currawong—or, at least, to place it properly in context. Some of the demonstrations below fail the significance test, but would probably pass it if placed in a different context.

7.1.1. Test hardware

Experiments were run on an HTC Dream smartphone, also known as the ADP1 or G1 [HTC Corporation 2010b]. This phone includes two processor cores: an ARM 9, and an ARM 11. The ARM 9 implements what is known as "baseband" functionality: low-level interaction with the various mobile radios on the phone in order to provide basic phone service. The ARM 11 core implements all other functionality: this is the core on which the Android operating system, and all applications, run. Both cores run at a maximum speed of 528 MHz. The phone incorporates an LCD running at a resolution of 320 by 480 pixels with 16-bit colour.

For the CAmkES tests, the phone ran on the OKL4 microkernel, version 3.0 [OK Labs 2010]. This version of the microkernel was ported to the HTC Dream smartphone by the author [NICTA 2010]. For the Android tests, the phone ran a standard Android 1.6 system image as supplied by the manufacturer [HTC Corporation 2010a]. The Linux kernel was modified to provide convenient access to the phone's cycle counter, but was otherwise not changed.

7.1.2. Methodology

To perform the tests, each application was started from a freshly-booted phone. Each experiment ran for several seconds, and experiments were repeated to ensure that the result was consistent. Results are reported in terms of millions of CPU cycles taken. This number was obtained from the CCNT register of the ARM processor's performance monitoring unit.

An interesting quirk of this particular phone is that the ARM9 and ARM11 cores share access to the memory bus, with the result that code executing on the ARM9 can impact the performance of the ARM11. This problem was mitigated by not running tests during the first twenty seconds after boot (because testing showed that this was when the ARM 9 was most active) and by verifying that the results of multiple runs of the same test showed little variance.

Anti-pattern	Example	Remedy
Context switching	Same-domain decoder	Merge protection domains
Copying	Eliminate RGB conversion	Replace component
Overly-generic API	New file system API	Interpose component
Unsuitable data	New file system API	Interpose component
Reprocessing	Eliminate RGB conversion	Replace component

Table 7.1. Summary of CAmkES-specific examples

7.2. CAmkES

The three optimisations implemented for CAmKES were designed to replicate potential real-world optimisation. However, the system on which they were implemented is a prototype. The absolute numbers presented here are, therefore, less important than the capability being demonstrated.

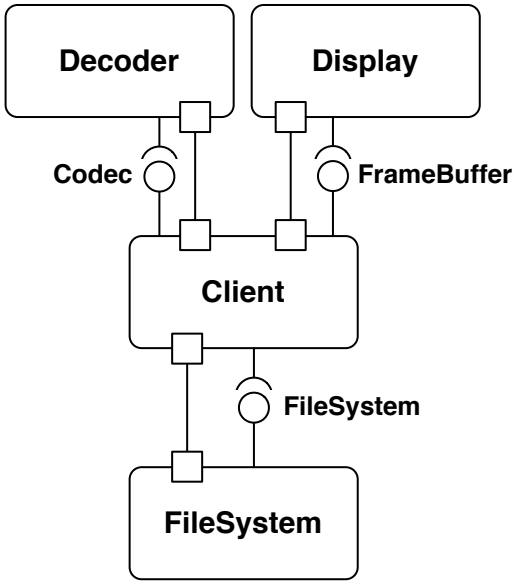

Figure 7.1. Componentised video player

The CAmkES examples are based around the componentised video player, introduced in Chapter 3. The video player's architecture is reproduced in Figure 7.1.

Currawong's support for architectural optimisation of component systems is demonstrated through three examples, each of which starts with the basic componentised video player (Figure 7.1) and transforms it by merging protection domains, replacing components, or interposing components. Table 7.1 summarises the examples and their purpose. The first column of this figure lists anti-patterns; the second column ("Example") names the example below which addresses the anti-pattern; and the final column ("Remedy")

describes the remedy employed by the example.

The first example, *same-domain decoder*, addresses a possible context switching anti-pattern by combining the protection domains of two components. The second example, *eliminate RGB conversion*, deals with unnecessary data reprocessing and copying by replacing the Decoder component. The final example, *new file system API*, addresses issues with an overly-generic API and unsuitable data structures by interposing a component between the Client and the FileSystem components.

7.2.1. Same-domain decoder

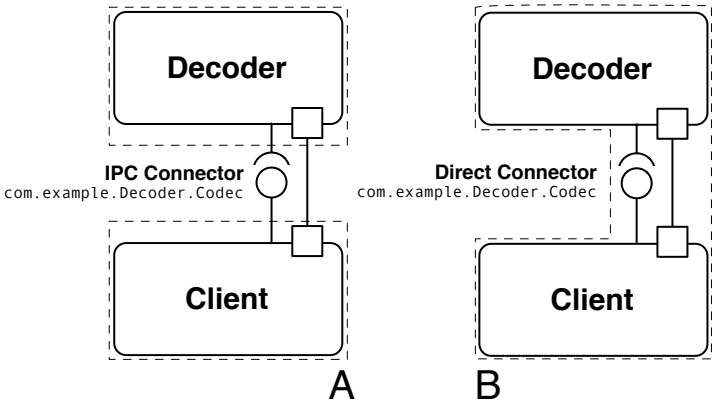

Figure 7.2. Protection domain merging in the componentised video player

This optimisation, introduced in Section 5.2.1, merges the protection domains of two components.

The CSL used to perform the optimisation is shown in Figure 7.3. The specification first searches for a component that matches the template shown in `MatchDecoder` (lines 1 to 12 in the figure). This specification makes use of CAmKES-specific extensions to the templating language to indicate architectural features of the system. The template matches a component which connects to a specific Decoder component, specified using a fully-qualified name; and which makes use of the `decode` function supplied by the functional connector between the two components. After this match is found, Currawong verifies that the two components are in separate protection domains (line 16). If they are, the specification instructs Currawong to combine the protection domains (line 17).

The system-level communication interface used for the functional connectors (that is, whether to use IPC or direct function calls) is not specified: selection of an appropriate method is left to Currawong. If the components share a protection domain, direct function calls are used. If they do not, IPC is used.

This example demonstrates the benefits of a declarative approach to specification. This specification relies purely on structure search, and in fact a large amount of searching is performed: producing a template requires a search across components in Currawong's

```
 1 MatchDecoder is C {
 2     component com.example.Decoder $Decoder;
 3
 4     component $Client {
 5         connector com.example.Decoder.Codec $Codec;
 6         connect $Codec to $Decoder;
 7
 8         anonymous $Func {
 9             $Codec.decode(_);
10         }
11     }
12 }
13
14 optimise(mergedecoder, App) is
15     Match = App.match(MatchDecoder),
16     App.DisjointComponentPDs(Match.Client, Match.Decoder),
17     App.AddToPD(Match.Decoder, Match.Client).
```

Figure 7.3. The "Merge protection domains" optimisation specification

Buffer size	**Switches**	**Separate-PD/stddev**	**Same-PD/stddev**	**Speedup %**
1 frame	320	228.34 / 0.03	226.32 / 0.02	100.89
2 frames	160	227.08 / 0.01	226.13 / 0.02	100.42
4 frames	80	225.49 / 0.02	225.02 / 0.01	100.21
8 frames	40	223.00 / 0.01	222.72 / 0.02	100.13

Table 7.2. The "Merge protection domains" optimisation, results

ADL representation, and ensuring that PDs are disjoint is a unification. However, this searching is not made explicit in the specification, for an overall improvement in specification readability.

The optimisation was tested by passing 160 frames of data from the Client to the Decoder for decoding. The Decoder simply copied the data to an output buffer, which was passed back to the client via shared memory. Thus the only operation being tested was the cost of memory copying and IPC overhead. This procedure was repeated thirty times.

The results are shown in Figure 7.2. For each test, the value presented is the average of thirty runs. This value represents millions of CPU cycles counted. A chi-square test for uniform distribution "Switches" records the total number of context switches performed in the case where Client and Decoder components reside in separate protection domains. The "Separate-PD" column represents the original case, where Client and Decoder components are separated. The "Same-PD" column represents the optimised case, in which Client and Decoder reside in the same protection domain. "Speedup" records the percentage difference between the separate-protection-domain and same-protection domain tests, where 100.00 is exactly the same speed. Each test was run multiple times with different buffer sizes—large buffers result in less communication between Client and

Decoder. These results show that the benefit due to optimisation in this case is negligible.

Discussion

Performing this particular optimisation on this particular system is not worthwhile. However, this "null result" is still interesting, as far as architecture optimisation is concerned. Combining protection domains was identified in Chapter 4 as a method that many domain-specific optimisation techniques use to improve performance: why is it so uninspiring here?

The problem lies in separating an optimisation remedy from its environment. The OKL4 microkernel platform on which CAmkES is based is very efficient at switching between protection domains. Other systems may have different priorities. Android, for example, is almost two orders of magnitude slower than OKL4 at performing IPC. A *ping-pong* test, in which a message is sent from one component to another, and then a reply is sent back to the original component, takes 1 592 CPU cycles on OKL4, but 95 053 CPU cycles on Android [Hills 2009]. Eliminating context switches due to IPC on Android is therefore a worthwhile goal. This possibility is explored further in Section 7.3.1.

Overall, the message from this null result is that context is important: optimisations that may be worthwhile for one system are ineffective on another.

7.2.2. Eliminate RGB conversion

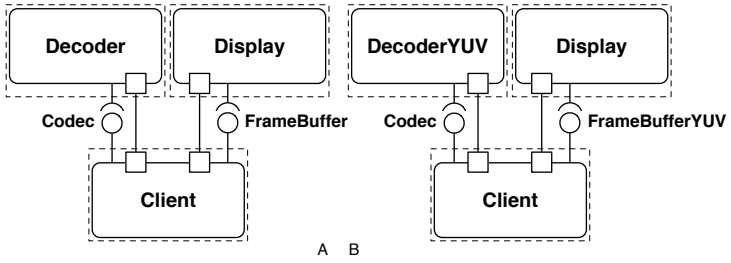

Figure 7.4. Component replacement in the componentised video player

The RGB conversion elimination optimisation was introduced in Section 5.2.2. This optimisation cannot rely entirely on structure search. If the Client component makes use of the data supplied by the Decoder, then the optimisation cannot be performed—the Client would not understand the new data format. Therefore, data-flow search is used to ensure that the Client simply passes data from Decoder to Display, without storing it or accessing it.

Figure 7.5 shows the complete optimisation specification. This specification is rather long and warrants some explanation:

1. The match specification (lines 1 to 16 in the figure) references three components. Decoder (line 2) and Display (line 3) are specified by fully-qualified name, ensuring

```
 1 MatchDecoderDisplay is C {
 2     component com.example.Decoder $Decoder;
 3     component com.example.Display $Display;
 4
 5     component $Client {
 6         connector com.example.Decoder.Codec $Codec;
 7         connect $Codec to $Decoder;
 8         connector com.example.Display.FrameBuffer $FrameBuffer;
 9         connect $FrameBuffer to $Display;
10
11         anonymous $Func {
12             $Codec.decode($Frame1);
13             $FrameBuffer.update($Frame2);
14        }
15     }
16 }
17
18 optimise(norgb, App) is
19     Match = App.match(MatchDecoderDisplay),
20     Match.Frame1 = Match.Frame2,
21     RestrictAccess = [Match.feature("$Func.$Codec.decode"),
22             Match.feature("$Func.$Display.update")],
23     Match.access(Match.feature("$Func"),
24             Match.Frame1, RestrictAccess),
25
26     App.replace_component(Match.Decoder,
27             "com.example.DecoderYUV"),
28     App.replace_connector(Match.FrameBuffer,
29             "com.example.Display.FrameBufferYUV").
```

Figure 7.5. The "Eliminate RGB conversion" optimisation specification

that the optimisation is only applied to systems using these unique components. No other match information is required of these components—this follows as a general consequence of the principle that unique matching reduces specification requirements (Section 5.4.2).

2. The third component reference (lines 5 to 16) does not specify a unique reference. The implication is that this optimisation specification applies to *any* component which makes use of the named Decoder and Display components. This match specifies a number of additional structure matches for this component. The client must have connectors of a named, unique type to both the Display and Decoder components (lines 6 to 9). The client must also define a function which calls specific functions on both connectors (lines 11 to 14).

3. The `optimise` method first performs a structure search (line 19).

4. Following the structure search, two data-flow searches are performed. The first

Before optimisation/stdev	After optimisation/stdev	Speedup %
263.12 / 2.69	114.65 / 1.84	229.50

Table 7.3. The "Eliminate RGB conversion" optimisation, results

verifies that the object passed to `decode()` is the same object that is passed to `update()` (line 20)—in other words, that decoded data is being passed to the frame buffer.

5. The second structure search verifies that the data being passed from one component to another is only used within the two functions named. A list is built containing the functions which are allowed to access the data (lines 21 and 22). This list is then passed to the `access` rule described in Section 5.8.3, and Appendix B. This rule verifies that only the named functions access the Frame1 object.

6. If both data-flow searches succeed, the system is modified: the Decoder component is replaced with a DecoderYUV component (lines 26 and 27), and the connection between Client and Display is similarly replaced (lines 28 and 29).

To evaluate the optimisation specification, the Decoder component was modified to perform YUV-to-RGB conversion.

The results are shown in Figure 7.3. As before, numbers shown are in millions of CPU cycles. "Speedup" shows the performance improvement of the no-conversion case relative to the with-conversion case.

Discussion

This optimisation specification eliminates a data-reprocessing step and also a data copy. These are both bus-intensive operations, so eliminating them is particularly useful on handheld devices—smartphones in general, and the G1 phone used in this test in particular—have slow buses. The final optimised application is faster than the original example, because the YUV encoding uses less space (one byte per pixel instead of two).

The current generation of smartphone displays do not have native YUV LCDs. However, it is common for the native format of a smartphone's included camera to be some variant of YUV, and a small variant of this optimisation which preserves the native data format if at all possible is quite plausible for modern devices.

This example demonstrates several design aspects of Currawong specification language that work to reduce the verbosity of the optimisation specification. For example, the structure search described in step 3 of the above description is quite complex: the structure search must first find a particular architectural layout (three components, two connectors), then it must examine one component for a particular sequence of function calls to other components. This work is handled by the implementation. CSL's Prolog heritage means that the specification is ultimately handled internally as a series of nested searches. Hiding the complexity of the nested searches means that Currawong does the right thing even in strange contingencies: for example, a component could satisfy all the criteria except

for the final data-flow search, and Currawong would ensure that it did not result in a false match.

7.2.3. Protocol translation

The final component-based remedy, *component interposition*, implements a memory-sharing optimisation in the file system component of the video player. It was introduced in Section 5.2.3

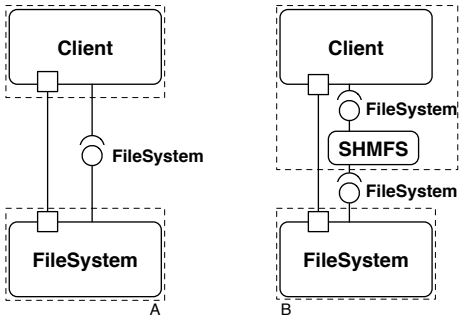

Figure 7.6. Component interposition in the componentised video player

The optimisation relies on domain-specific knowledge combined with static analysis. The memory allocator in this system is known to supply *page-aligned* memory regions for allocations equal to, or greater than, the page size. If the optimisation specification can verify that the allocated memory region used by the Client component for its data has a size which is a multiple of the page size, it can therefore assume that the data region is also page-aligned.

The optimisation specification is shown in Figure 7.7. Here the template specification (Lines 1 to 16 in the figure) identifies two components: the FileSystem and Client components. Once again, one component is not identified by fully-qualified name, but the other one is: this pattern allows optimisations to be "for" a particular component, while still capable of being applied to any client of that component. The specification identifies two functions within the component: memory allocation (line 9) and invocation of the `read` operation on the FileSystem component (line 13).

The optimisation specification first performs a structure search (line 19). It then attempts to verify that the size of the memory allocation is a multiple of the page size.If this succeeds, it then verifies that the pointer returned by `malloc` is the one that is passed in to `read` (line 20).

Once the match is complete, the actual transformation is simple, as it makes use of the built-in `interpose` rule to add a component.

The results of this optimisation are shown in Figure 7.4. Here a single 3-megabyte file was transferred from FileSystem to Client. Two buffer sizes were evaluated. As before, results are presented in millions of CPU cycles. Sizes are in bytes, and "Speedup" is the performance of the optimised case relative to the unoptimised case.

```
 1 MatchFS is C {
 2     component com.example.FileSystem $FSComponent;
 3
 4     component $Client {
 5         connector com.example.FileSystem.FileSystem $Files;
 6         connect $Files to $FSComponent;
 7
 8         anonymous $Init {
 9             $AllocBuffer = malloc($Size);
10         }
11
12         anonymous $Control {
13             $FSComponent.read(_, $ReadBuffer, _);
14         }
15     }
16 }
17
18 optimise(fsshm, App) is
19     Match = App.match(MatchFS),
20     is_pagesize(Match.Size),
21     Match.AllocBuffer = Match.ReadBuffer,
22
23     App.interpose(Match.Files, "com.example.SHMFS").
24
25 is_pagesize(Size) is
26     mod(Size, 4096, 0).
```

Figure 7.7. The "Protocol translation" optimisation specification

Discussion

As expected, halving the number of copies involved in this process doubles the throughput in the 64-kilobyte buffer case. Interestingly, however, the performance is not as dramatic in the 4-kilobyte buffer case. This is because data copied into the shared memory region are overwritten before the information in that region is written back to main memory. Thus the example applied to smaller packet sizes mostly remains within the CPU cache.

It is quite conceivable that this component could be interposed even *without* the attendant data-flow analysis. In that case, the component could dynamically check whether the memory was page-aligned. The cost of performing this check will be small compared with the cost of reading data from the file system.

Buffer size	**Copying/stdev**	**Sharing/stdev**	**Speedup %**
4096	30.50 / 0.04	25.64 / 0.01	118.95
65536	43.23 / 0.02	22.59 / 0.01	191.37

Table 7.4. The "Protocol translation" optimisation, results

7.3. Android

To demonstrate performance on a commercial smartphone operating system, Currawong was applied to several Android applications. Architecture optimisations for Android require more code modification than those for CAmkES, because Android is not as strongly componentised as CAmkES.

Two architecture optimisations are shown. The first, "touch events", addresses a context switching anti-pattern by combining protection domains. The second, "redraw", addresses a copying anti-pattern by modifying the API.

7.3.1. Touch events

The touch events optimisation, described in Section 5.2.4, puts the application in control of reading and processing its own touch events. In the Android system, touch events received by the in-kernel touchscreen driver are delivered to the character device file `event0`. Applications with appropriate permission can therefore open this file and process touch events directly.

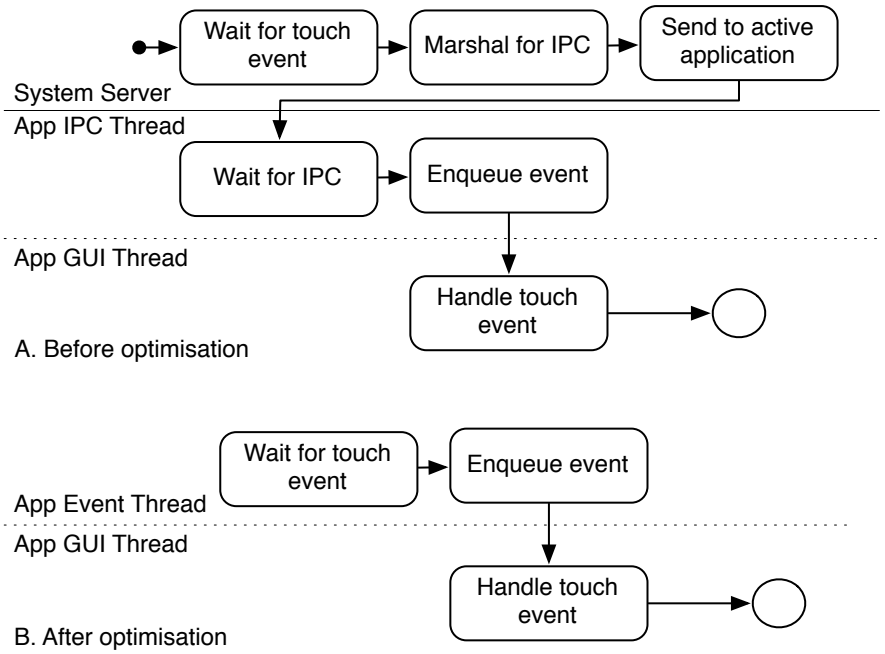

Figure 7.8. Touch events optimisation before (A) and after (B)

To implement the optimisation, custom code was written as an Android native code library, to be loaded into applications using the standard Java Native Interface method [Liang 1999]. This code creates a new application thread, which continuously reads touch events and delivers them to the application. Figure 7.8, section B, shows the process involved.

```
 1 MatchOnTouch is Java {
 2     class $ClassName implements android.view.View.OnTouchListener;
 3 }
 4
 5 MergeOnTouch is Java {
 6     class $ClassName {
 7         $ClassName
 8         {
 9             au.com.nicta.cw.OnTouch.register(this);
10         }
11     }
12 }
13
14 optimise(ontouch, App) is
15     Match = App.match(MatchOnTouch),
16     App.add_module('au.com.nicta.cw'),
17     App.merge_all(Match, MergeOnTouch).
```

Figure 7.9. The "Touch events" optimisation specification

Name	**Unopt / stdev**	**Opt / stdev**	**Speedup %**
Continuous Scrolling	49.23 / 10.17	43.19 / 5.34	113.98

Table 7.5. The "touch events" optimisation, results

The specification is shown in Figure 7.9. Finding an appropriate location to perform the optimisation is a simple structural match (lines 1 to 3, and line 15, in the figure). This match reflects the Android API—Java classes which are to receive touch events always implement the `OnTouchListener` interface.

The custom event-handling code takes care of reading events, encoding them into a format suitable for OnTouchListeners, and delivering them to the application. The only application requirement is that it inform the new event-delivery thread of the target for events, which it does by calling a function, register(), provided by the new code (line 9 in the figure).

Unlike the CAmkES optimisations, the Android specification adds a small amount of code using Currawong's code merging feature. When the merge is applied (in line 17), the merged object, MergeOnTouch, is applied to code corresponding to the portions of the Match object which match the structure of the merged object. In other words, the `register` call is added to the application wherever appropriate (but only in areas covered by the Match object).

To evaluate the optimisation, a sample application was written which displays an image which is larger than the physical display. Dragging a finger across the display scrolls the image. The test involved dragging a finger across the display for several seconds and measuring the number of CPU cycles consumed. To ensure consistency of input, events from the touchscreen were recorded and replayed to the application, so each test received

exactly the same number of inputs at exactly the same rate.

The results of the evaluation are shown in Table 7.5. Numbers are given in millions of CPU cycles. The first column shows the number of cycles required by the original application; the second column shows the number of cycles taken by the optimised version; and the final column, "Speedup" shows the percentage improvement of the optimised version.

Discussion

This type of architecture optimisation, if applied to other system devices, would result in a system resembling the Exokernel [Engler et al. 1995]: application-level, decentralised management of resources. As with Exokernel, the result is a reduction in CPU usage. The improvement can be viewed in two ways. In one sense, it is very significant: relocation of the processing code for a single device significantly lowered the CPU usage of the application. However, like all optimisations, this one should be considered in context. Of all the devices whose data are transferred via Binder, the touch device generates the largest amount of events per second, and continuous on-screen movement represents a worst case for this particular data path. The conclusion that should be drawn is that the optimiser user should consider the nature of the application to which an optimisation is to be applied.

The new code added by the touch events optimisation reads from the Android Linux device node `event0`. Normally, this is forbidden by the Android system. This evaluation did not include the permission management that would be necessary for a production version of the optimisation. This does not impact the result. Permission management would be simple to implement: the System Server would only need to appropriately manage the UNIX-level file permissions on the relevant device node. Even if this process was slow, however, it would not significantly reduce the benefit of the optimisation, because it would only be necessary to check permissions associated with the node when switching applications, i.e., off the critical path.

7.3.2. Redraw

The Android redraw optimisation replaces a slow but simple and easy-to-debug drawing interface with a faster but more complex interface. It was described in Section 5.2.5, and illustrated in Figure 7.10.

The optimisation specification is shown in Figure 7.11. Once again, the matching portion is a simple structure search (lines 1 to 6 in the figure). Most of the remainder of the specification adds code to the matched class. The first portion of merged code (lines 8 to 26) adds a number of functions to the class to partially implement the code necessary to support a SurfaceView. The second portion of merged code (lines 28 to 34) also supports the SurfaceView API. It is provided separately because it may be applied multiple times, one for each class constructor.

Results of the evaluation are shown in Figure 7.6. Three applications were tested. As before, numbers are shown in millions of CPU cycles. The "Unopt" column shows the

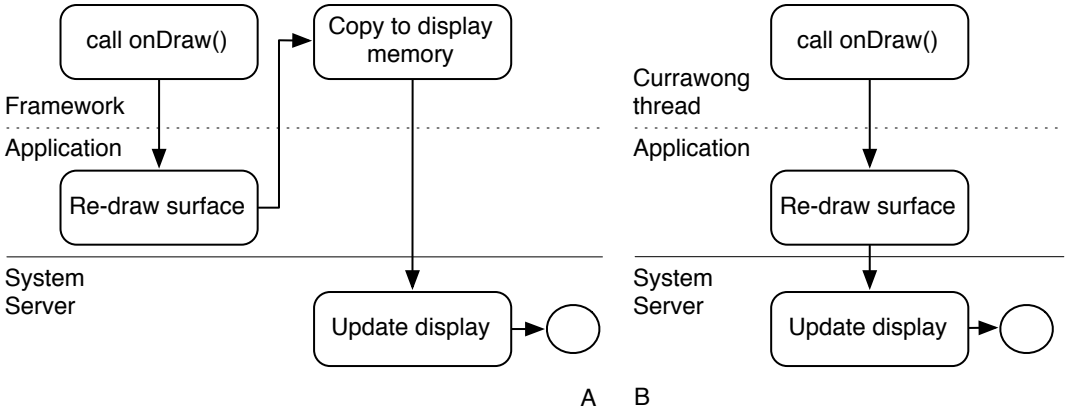

Figure 7.10. Redraw optimisation before (A) and after (B).

Name	**Unopt / stdev**	**Opt / stdev**	**Speedup %**
Redraw 60 FPS	78.12 / 3.51	37.67 / 0.34	207.38
Space War	724.93 / 3.37	413.13 / 10.94	175.47
Bonsai Blast	1384.33 / 32.95	1179.23 / 13.64	117.39

Table 7.6. The "Redraw" optimisation, results

performance of the application prior to optimisation; the "Opt" column shows the performance after optimisation; and the "Speedup" column shows the percentage improvement due to optimisation.

The first application, Redraw 60 FPS, is a test application custom-written for this optimisation. It simply draws to the display as quickly as possible. In Android, display updates are rate-limited to 60 frames per second, so this is the speed at which this application refreshes the display. Because it does very little other than re-draw the display, Redraw 60 FPS represents a best-case scenario for the redraw optimisation.

The other applications evaluated are both proprietary applications written by third parties and made available to all Android users via the Android Marketplace, a centralised application repository.

Space War is a top-down scrolling "shoot-em-up" game. The player controls a space ship which flies along a vertically-scrolling display. Other space ships appear from the top and sides of the display, and the player must shoot these other ships and avoid incoming laser fire to win the game.

Bonsai Blast is a puzzle game in which multi-coloured balls travel in single file along a pre-determined path, and the player must launch balls of the correct colour into the appropriate place in the line of balls in order to score points.

To evaluate each game, a custom program was written which counts the number of CPU cycles which elapse within a three-second window of game play. Care was taken to ensure that measurements between unoptimised and optimised runs measured the same part of

```
 1 MatchOnDraw is Java {
 2     class $ClassName extends android.view.View {
 3         protected void onDraw(Canvas _)
 4         { }
 5     }
 6 }
 7
 8 MergeOnDraw is Java {
 9     class $ClassName extends Android.view.SurfaceView
10             implements Android.view.SurfaceHolder.Callback {
11         private int _cw_tok;
12
13         public void surfaceDestroyed(SurfaceHolder s) { }
14
15         public void surfaceCreated(SurfaceHolder s) {}
16
17         public void surfaceChanged(SurfaceHolder s,
18             int fmt, int w, int h) {
19             _cw_tok = au.com.nicta.cw.Draw2D.init(this);
20         }
21
22         protected void invalidate() {
23             au.com.nicta.cw.Draw2D.invalidate(_cw_tok);
24         }
25     }
26 }
27
28 MergeOnDrawInit is Java {
29     class $ClassName {
30         $ClassName {
31             getHolder().addCallback(this);
32         }
33     }
34 }
35
36 optimise(ondraw, App) is
37     Match = App.match(MatchOnDraw),
38     App.add_package('au.com.nicta.cw'),
39     App.merge(Match, MergeOnDraw),
40     App.merge_all(Match, MergeOnDrawInit),
41     App.rename_method
42         (Match.ClassName, 'onDraw', '_cw_onDraw'),
43     App.merge(Match, MergeOnDrawInit).
```

Figure 7.11. The Android redraw optimisation

the game. Getting this right was a matter of trial and error. For example, the first level of Bonsai Blast gets harder (more balls appear on-screen) approximately fifteen seconds

into the start of the game. At this point, the game's CPU usage increases significantly. To ensure consistency, each run was measured multiple times, from the same location within the game.

Discussion

The redraw optimisation showed surprisingly promising performance improvements, even in applications which already make heavy use of the CPU, and it can be applied to commercial applications without changing their semantics.

The results here show noticeably different speedup percentages across the three applications. There are two reasons. Firstly, the amount of additional processing performed by the application influences the proportional speed-up. Secondly, Bonsai Blast has a lower frame rate than Space War, and Space War has a lower frame rate than Redraw 60 FPS. A lower frame-rate means less optimisable work per second, which results in a reduced performance improvement.

Bugfix or optimisation?

It could be reasonably argued that an application's use of the Surface technique rather than the SurfaceView technique constitutes a bug, i.e., the application is simply using the wrong API. This is in contrast to an optimisation, in which the API is used correctly but inefficiently. This distinction is blurry, and goes to the heart of system software optimisation as a whole. In many cases, the SurfaceView case included, what constitutes a bugfix in one context is an optimisation in another context. In this specific example, applications written using the slower technique perform correctly and with acceptable performance. Furthermore, the application code is simpler and easier to maintain. In this context, using the slower technique is not a bug, but a design trade-off—presumably one of many made during development. Ultimately, the bugfix-or-optimisation question must always be framed in an appropriate context before we can determine whether Currawong is performing bugfixes, design-space exploration, or optimisation.

7.4. Costs of running Currawong

There are two types of execution cost that should be considered in relation to Currawong: the run-time cost imposed on the optimised application by Currawong, and the execution cost of running Currawong itself.

7.4.1. Application run-time cost

When considering run-time application execution-time costs, it is instructive to again divide the overall cost into that imposed by the optimisation system itself, and that which arises are a result of implementing an optimisation.

The smaller the run-time cost imposed by Currawong, the better: obviously, lower execution-time overheads results in better application performance, all else equal. There

is a subtler benefit also: the less run-time overhead Currawong imposes, the more predictable it becomes. This is an important benefit to optimisation writers: an optimisation that unpredictably introduces its own overheads makes writing optimisation specifications more difficult.

Consequently, Currawong does not impose any run-time overhead on optimised systems. This is achieved both through use of static (rather than dynamic) analysis for verification, and by direct re-writing of the application statically, rather than, for example, by rewriting the application at run-time; or by making use of tracing or debugging features of the operating system to interpose new code implementing a particular optimisation.

Currawong does not, however, make any guarantees about the performance of the modifications implemented by the optimisation specification. This is a design decision based on the assumption that the optimisation writer is a domain expert who has a good understanding of the performance characteristics of the API she is optimising.

7.4.2. Currawong execution cost

The other type of execution cost that should be considered is that taken by running Currawong. For this version of Currawong, that cost is negligible, even for optimisations that require significant analysis or which make heavy use of CSL's Prolog heritage to perform unification. Structure and data-flow search for each example here was very fast (less than one second). Total run-time was slightly longer, as it involved decompiling, parsing, and then reassembling the application. The reason for this is nothing to do with efficiency of implementation: Currawong simply does not perform a significant amount of static analysis. A future version of Currawong, with stronger support for data-flow search, would take longer.

Even if Currawong took a long time to execute, this would, in some sense, not matter: Currawong only needs to be run once, by the optimiser user; after that, the optimised application can be executed many times. Therefore the cost incurred by the optimiser user can be amortised across each run of the optimised application. Yang et al. make a similar point when discussing the static analysis tool to find bugs in kernel code [Yang et al. 2006]: even a high per-run cost is offset by the very large pay-off from a successful execution.

Yang's analysis in the above paper is a good data point illustrating the trade-off Currawong makes between depth of analysis and performance: Yang's symbolic evaluation engine takes approximately an hour to analyse functions with, in his words, "complex control flow but little symbolic looping".

7.5. Discussion

The aim of the chapter was to clearly demonstrate both that Currawong cannot improve a system that will not benefit from the optimisation being applied; and that, in systems which do benefit, significant performance gains can be made. The results presented in this chapter prove that both goals were achieved.

It is helpful to consider whether the optimisations performed by Currawong could be achieved as effectively using other means. Several alternate methods exist:

1. Source-code modification. The application authors could have performed any of these optimisations themselves while building the application. The optimisation could either have been performed manually, or by using the source-to-source optimisation techniques described in Chapter 2. In some cases, this approach is better than using Currawong, because the programmer has the potential to perform a wide range of algorithmic optimisations currently inaccessible to Currawong. For example, rather than attempting to eliminate RGB conversion, as described in Section 7.2.2, each component could be rewritten so that no component actually requires data in RGB format. However, a major motivating factor for Currawong, as described in Chapter 1, is the *absence* of source code, so that optimisations can be applied even if the application developer was unaware of them at the time of compilation. By definition, source-code-based techniques cannot be applied to optimisation problems for which the source code is not present, which is the same problem domain that Currawong addresses. Even if source code is available, binary modification confers a number of advantages related to target-platform adaptibility. For example, binary modifications can be performed when the exact hardware characteristics of the target device are known. This may not be the case for source-code-based or compile-time modifications. The additional information can be used to do hardware-dependent things, such as, for example, to adjust RAM usage to a level suitable for the device. The same advantage applies to adapting the component to software running on the platform. Keller and Hölzle demonstrate this type of adaptation in their Binary Component Adaptation system [Keller and Hölzle 1998].

2. Improved system libraries. The authors of system libraries which were the target of optimisations could improve them. This solution is similar to the previous one, in that it requires source code access, and this access may not be available. Even when they can be updated, modifying system libraries to include optimisations is a risky process: effectively *every* client of the APIs exposed by those libraries becomes optimised. This calls for more care than is required from Currawong, which is at least capable of performing static analysis to determine whether an application should be optimised. Nonetheless, careful modification of system libraries is an ideal solution, if it is possible: the converse of the "all applications become optimised" problem is that optimisation is applied to new applications without requiring an intermediate optimiser, such as Currawong.

3. Binary optimisation techniques. Systems such as Keller and Hölzle's , as discussed in Section 2.4, could be applied to these optimisations. Notably, Keller and Hölzle's system has a small run-time cost, which Currawong avoids; later systems, such as PROSE [Popovici et al. 2002], avoid this. Notably, these systems are presented as binary implementations of AOP, and are, unlike Currawong, limited to structural modifications.

The performance gains due to Currawong come without significant modification to the application, without requiring application source code, and without requiring the involvement of the original application authors. Applying the novel technique of architecture optimisation to complete systems can bring about significant performance improvement.

8. Conclusion

> I wish to God these calculations had been executed by steam!
>
> – Charles Babbage, possibly apocryphal

This dissertation described architecture optimisation, a novel high-level optimisation technique. Architecture optimisation is viable both on research systems, and on commercial operating systems on which significant thought has, presumably, already been given to performance tuning.

An architecture optimiser, Currawong, was implemented and tested. Currawong can optimise two completely different types of system, deal with multiple languages, and optimise without requiring source code, all without imposing a run-time cost. These characteristics distinguish it from all other optimisation techniques. Unlike traditional compiler optimisations, Currawong is extensible; unlike library optimisations, Currawong can optimise without source code, and across a larger domain; unlike refactorings, Currawong can modify the behaviour of code. Currawong delivered good performance results across a variety of optimisation categories and both systems.

This chapter summarises the main themes of the dissertation and discusses directions in which this type of work could be taken in the future.

8.1. Summary

Currawong is a tool to apply domain-specific, or *high-level*, optimisations to applications. It achieves its performance improvements by optimising at API boundaries. API boundaries are a good place to implement high-level optimisations, because an API call is a rich source of application-level information. For example, the fact that Java applications in the Android "Redraw" example implemented a specific method of a well-known interface meant that the optimiser could assume many things about the nature of the application: that it would be drawing to the screen, that it would be performing drawing calls on a particular object, that that object would be supplied in a callback fashion through a call to a known function within the application, and so on. Every piece of knowledge that can be assumed in this way is one less fact that must be verified through static analysis.

The introduction to Chapter 5 reproduces three criteria by Aho, Sethi, and Ullman, on the criteria for optimisations: they must preserve program meaning; they must, on average, speed up programs; and they must be worth the effort. It is interesting to note that Currawong does not, by itself, meet those criteria: Aho et al. would probably classify Currawong as an *optimisation framework*, or simply as a compiler. Only when combined with a well-written optimisation specification does Currawong become an optimiser.

This distinction may seem trivial, but it hides a design decision: Currawong relies heavily on the optimisation writer to produce correct optimisations. Currawong itself will obligingly apply "optimisations" that insert Trojan horses into code, slow it down instead of speed it up, or merely introduce subtle bugs. This is, of course, a deliberate design decision: the author of the API is best placed to write optimisations that take advantage of that API. Currawong helps the author by providing a static analysis toolkit. However, it remains to be seen whether API authors are up to the task of providing high-quality optimisations.

In many other ways, Currawong behaves like other optimisation systems. Currawong's matching emphasises conservatism: if a property cannot be verified, Currawong assumes the property did not verify. This conservatism property is of course an important part of ensuring that optimisations preserve program meaning. Optimisations can be additive: multiple optimisations can be applied to the same application resulting in performance increases in multiple areas.

8.1.1. Designing to be optimised

Architecture optimisation is applied to binary applications, by users (or system creators), after the code to be optimised has been written and packaged. One of the advantages of this approach is that the impact of bad design decisions can be ameliorated post hoc, by applying architecture optimisation. Realistically, however, this is only true to an extent.

Consider the protocol header optimisation described in Section 4.1.4. In this optimisation, the structure which defines a network packet was changed. The old structure required reallocation and copying whenever the packet size was increased, but the new structure deals with the problem more efficiently, by treating a packet as a linked list of regions. Assume that this optimisation is to be applied to two communicating components in a component system, one which consumes network buffers (such as a network card driver) and one which produces them (such as a TCP/IP stack)

A standard design for this interaction would be to share memory between the producer and consumer, and leave management of the format of the buffers to the components. However, if this design is chosen, creating a high-level optimisation becomes rather difficult. The optimiser would have to recognise that the output of standard C memory-management routines (such as `malloc` and `realloc`) were being applied to packet buffers, possibly involving the tracking of pointers to packet buffer objects throughout the entire application. If, instead, access to packet buffers was provided through a function-call API, applying an architecture optimisation is as simple as searching for uses of that API.

It seems that the best way to write optimisable APIs is to try to provide as high-level an interface as is possible: rather than expose the format of a data structure, provide an interface to the structure via a functional API; rather than allow applications to make arbitrary modifications, provide a set of single-purpose functions to do the job for them.

These criteria for API design are familiar: they were first proposed by Parnas in 1972 as part of a set of guidelines for writing code that is easy to maintain and change [Parnas 1972]. It is perhaps unsurprising that the old rules still apply in this new context.

8.2. Achievements

This dissertation describes an optimisation technique that is both novel and practical. Other high-level optimisation techniques, such as active libraries, have limited domain applicability by design. These techniques are designed to assist programmers. Currawong instead assists end users, the real beneficiaries of optimised applications, because it does not require programmer involvement when applying optimisations—which, in turn, means that applications which are no longer supported by their authors, or hardware on which the application author cannot test her application, can still benefit. Currawong combines the simplicity of refactoring with the static analysis of active libraries to support a wide range of optimisation types.

Currawong is a success with regards to performance. Currawong's excellent performance on Android is particularly notable because this large and commercially-supported system has already been through several performance-enhancing revisions.

8.3. Future work

Currawong is a successful demonstration of a novel technique, but it is by no means perfect.

Architecture optimisation's benefits are also, to some extent, its disadvantages. The requirement that Currawong work without source code limits both its ability to perform static analysis and its ability to make code modifications. This shortcoming is somewhat mitigated by the design principle that domain-specific knowledge reduces analysis requirements. The two effective Android optimisations presented in Chapter 7, for example, rely only on structure search, and it is a strength of Currawong that significant performance improvements can be achieved even without cutting-edge significant static analysis techniques. Nonetheless, stronger support for static analysis, probably in the form of a better symbolic execution engine, is an obvious target for future work.

One of architecture optimisation's key design goals was its support for multiple languages. This is an important goal, because most systems are written in multiple languages. Currawong met this goal through its support for both Java and C. However, it is worth revisiting the assumptions implicit in this design goal. One implicit assumption is that architecture optimisations are written in specific languages. For example, the CAmkES optimisations were both specified and written in C, and the Android optimisations were specified in Java, and written in a combination of Java and C. Requiring optimisations to be specified in a particular language has the advantage the specification can closely match the actual code and structure of the targeted application. However, none of the the optimisations evaluated in Chapter 7 is *necessarily* language-specific. It is reasonable to expect the RGB-to-YUV conversion problem, for example, to crop up in a Java program, or for the Android redraw optimisation to be applicable to C code. Perhaps a logical extension of support for multiple languages is support for general-purpose specification, supporting multiple languages. The specification would then become language-independent.

Currawong and its specification language, the author's Python-based parser, and the Bcamkes binary component system will all be released as open source software.

A. Glossary

Android A software stack for mobile devices that includes an operating system, middleware and key applications. More information is available at `http://developer.android.com/guide/basics/what-is-android.html`.

Binary CAmkES A dialect of *CAmkES* supporting a number of additional features, including support for composition of binary-only components.

Binder A remote procedure call mechanism for *Android*. Binder consists of a (relatively) small kernel implementation, a user-space library, and JNI hooks for use from Java.

CAmkES The Component Architecture for microkernel-based Embedded Systems is a component system which runs on various dialects of L4. More information is available from the CAmkES home page, at `http://www.ertos.nicta.com.au/software/camkes/`.

architecture A machine-readable high-level description of a componentised system. In CAmkES, this is the description of the componentised system in Architecture Definition Language.

component system A software development tool, and the run-time portion of that tool, which allows a system to be created as a set of one or more communicating components. Contrast with *componentised system*

componentised system See *system*

object language The programming language in which an application or component is written. Contrast with *specification language*.

platform Another name for *software stack*.

software stack A collection of interacting programs that, combined, provide a fully-functional operating environment for applications. For example, the Android software stack consists of the Linux kernel, a set of run-time libraries, a virtual machine, system components written in Java, and a number of user-accessible applications. A basic CAmkES software stack consists of an L4 microkernel and CAmkES run-time code.

specification language The language in which a specification is written. May be different to the *object language*.

system A binary implementation of an architecture (see *architecture*).

B. Summary: Currawong API

This chapter provides a brief overview of the code matching, checking, and modification rules provided by Currawong.

B.1. Application object

B.1.1. `match(+Name)`

Produce a Match object
Searches for the named template and returns a Match object representing the portion of the application corresponding to that template.
Input: `Name` is a reference to a previously-defined template.
Output: The return value is an opaque type representing a match object. This rule either fails, or succeeds exactly once.

B.1.2. `rename_call(+Scope, +Old, +New)`

Rename all references to a function call
Searches `Scope` and all child scopes for any references to function invocation for the function named `Old`, and replaces them with function invocations to the function named `New`.
Input: `Scope` is a reference to a scope from a Match object. `Old` and `New` are strings containing function names.
Output: None. The rule always succeeds.
`rename_method(+Scope, +Old, +New)`
Rename a method definition
Searches `Scope` and all child scopes for a declaration of the function `Old`, and renames it to `New`.
Input: `Scope` is a reference to a scope from a Match object. `Old` and `New` are strings containing function names.
Output: None. The rule always succeeds.

B.2. CAmkES-specific Application object rules

B.2.1. `access(+Scope, +Object, =Funcs)`

Checks object usage
Checks that `Object` is only used by `Funcs` within `Scope`.

Input: `Scope` is a reference to a matched scope (obtained via `Match.feature/1`). `Object` is a match variable representing data (i.e. a function parameter). `Funcs` is a list of functions.
Output: The list of functions which use `Object` is returned in `Funcs`. If `Funcs` is bound, and the list is the same, the rule succeeds. If `Funcs` is unbound, it is bound to the list of functions, and the rule succeeds.

B.2.2. `AddToPD(+Satellite, +Planet)`

Combines protection domains
`Satellite` is placed in `Planet`'s protection domain.
Input: `Satellite` and `Planet` are component references from a match specification.
Output: None: The rule always succeeds.

B.2.3. `DisjointComponentPDs(+PD1, +PD2)`

Succeeds if PD1 and PD2 have disjoint protection domains
This rule accepts two component references and succeeds if the components are in completely disjoint protection domains.
Input: `PD1` and `PD2` are component references from a match specification.
Output: None. The rule succeeds if `PD1` and `PD2` are in disjoint protection domains, and fails otherwise.

B.2.4. `replace_component(+Old, +New)`

Replace a component with another component
Replaces a component referenced by a match specification with a new component.
Input: `Old` is a reference to a component scope from a Match object. `New` is a string containing a fully-qualified component name.
Output: None. The rule always succeeds. In the current implementation, Currawong returns an error to the user and exits if the new component is not found.

B.2.5. `replace_connector(+Old, +New)`

Replace a connector with another connector
Replaces a connector referenced by a match specification with a new connector.
Input: `Old` is a reference to a connector scope from a Match object. `New` is a string containing a fully-qualified connector name.
Output: None. The rule always succeeds. In the current implementation, Currawong returns an error to the user and exits if the new component is not found.

B.2.6. `interpose(+Connector, +Component)`

Interpose a component along a connector

A new component of type `Component` is created. The named connector, `Connector`, is disconnected from its destination component and connected to the new component. A new connector of the same type as `Connector` is created and connected from the new component to the original connector's destination. The new component is placed in its own protection domain.
Input: `Connector` is a reference to a connector scope from a Match object. `Component` is a reference to a component scope from a Match object. These must refer to the same Match object, and `Connector` must be one of the connectors used by `Component`. **Output**: None. The rule always succeeds. Failure conditions are as per `replace_connector`.

B.3. Java-specific Application object rules

B.3.1. `add_module(+Name)`

Adds the named module to the application
The module identified by `Name` is physically copied to the application, so as to make it available to new code.
Input: `Name` is the fully-qualified name of the new module. **Output**: None. The rule always succeeds.

B.3.2. `merge(+Match, +Merge)`

Adds code to a match
The named template object's code and data is added to the portions of the application identified by the Match object according to the rules described in Section 5.8.4. Merging is performed at most once.
Input: `Match` is a match object. `Merge` is a template. **Output**: None. If the merge cannot complete (because the structural features identified in the template cannot be correlated with the structural features in the Match object), the rule fails. Otherwise, it succeeds.

B.3.3. `merge_all(+Match, +Merge)`

Adds code to a match, multiple times
This performs the same function as `merge/2`, but the merge is performed as many times as possible. This is used in the example described in Section 7.3.1 to add code to every class initialiser.

B.4. Match object

B.4.1. `feature(+Path)`

Access a structural feature of a Match object.
Match objects make the structure of the matched code portion available to the optimisation specification via the `feature` rule.

Input: `Path` is a string containing a path access specifier. The access specifier should indicate a scoped name corresponding to scopes in the generating template. Scopes may include templated variables, which are substituted. Scopes may also include an underscore (_) as a wildcard, in which case any scope may be substituted. Scopes are searched starting from contents of the outermost scope of the match; if this fails the scope above that scope is searched, and so on. If multiple matches are found, only the first one is returned.
Output: The return value is an opaque type representing the feature, to be passed to code-modifying rules.
Examples:
`Match.feature("method")`: Match the single scope named "method".
`Match.feature("")`: Match the outermost scope.
`Match.feature("$ClassName")`: Match the scope indicated by the $ClassName match variable.
`Match.feature("_.method")`: Match the scope "method", which is contained within a scope.

Bibliography

Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5), February 1993.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of ECOOP 2006*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer Berlin / Heidelberg, 2006.

John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2), June 2003.

Charles Babbage. *Passages from the Life of a Philosopher*. Rutgers University Press, 1864.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), December 1994.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5), May 2000.

Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, July 2001.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Operating Systems Review*, October 2006.

Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *IEEE International Conference on Software Reuse*, November 1994.

Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. *SIGPLAN Notices*, 44:114–126, January 2009.

John Bruno, José Brustoloni, Eran Gabber, Avi Silberschatz, and Christopher Small. Pebble: a component-based operating system for embedded applications. In *Proceedings of the USENIX Workshop on Embedded Systems*, March 1999.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), November 2008.

Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo, a dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, May 2009.

Hsiao-keng Jerry Chu. Zero-copy TCP in Solaris. In *USENIX '96: Proceedings of the 1996 USENIX Annual Technical Conference*, 1996.

David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 2006.

Willem de Bruijn and Herbert Bos. Beltway Buffers: Avoiding the OS Traffic Jam. In *Proceedings of the 27th Conference on Computer Communications (INFOCOM 2008)*, April 2008.

Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2005.

Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*. ACM, 2008.

Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *SIGOPS Operating Systems Review*, 27(5), 1993.

Adam Dunkels. *Design and Implementation of the lwIP TCP/IP Stack*. PhD thesis, Swedish Institute of Computer Science, 2001.

Eclipse Foundation. Eclipse.org home page, February 2010a. URL `http://eclipse.org/`.

Eclipse Foundation. Unleashing the power of refactoring, February 2010b. URL `http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html`.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS – Operating Systems Review*, 35(5), 2001.

Dawson R. Engler, Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, New York, NY, USA, 1995. ACM.

Nicholas FitzRoy-Dale and Ihor Kuz. Towards automatic performance optimisation of componentised systems. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. ACM, 2009.

Bryan Ford, Jay Lepreau, Stephen Clawson, Kevin Van Maren, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable Components for OS Implementation. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*. IEEE Computer Society, 1997.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional Computing Series, 1999.

L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3), 2001.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, 1995.

Kang Su Gatlin. Profile-guided optimization with Microsoft Visual C++, 2010. URL `http://msdn.microsoft.com/en-us/library/aa289170(VS.71).aspx`.

Google Inc. What Is Android?, February 2010. URL `http://developer.android.com/guide/basics/what-is-android.html`.

Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of the IEEE*, volume 93. IEEE, 2005.

Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *SIGOPS Operating Systems Review*, 41(4), 2007.

Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM, 2005.

Michael Hills. Native OKL4 Android stack. BE Thesis, NICTA, 2009.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

HTC Corporation. HTC Developer Center for the ADP1, 2010a. URL `http://developer.htc.com/adp.html`.

HTC Corporation. HTC Dream, 2010b. URL `http://www.htc.com/www/product/dream/specification.html`.

IBM. *The FORTRAN automatic coding system for the IBM 704 EDPM*. International Business Machines Corporation, October 1956.

International Standards Organisation. Information technology – Programming languages – Prolog – Part 1: General core. Technical Report 13211-1, ISO, 1995.

Daniel Isaaman, Jenny Tyler, and Martin Newton. *Computer Spacegames*. Usborn Publishing Ltd., 1982.

JesusFreke. Smali and Baksmali. `http://code.google.com/p/smali/`, 2010.

Stephen Kell. Configuration and adaptation of binary software components. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 2009.

Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1445, page 307, 1998.

Gregor Kiczales, John Lamping, Anurang Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ACM Computing Surveys*, 28, 1996.

Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

Stephen Cole Kleene. *Mathematical Logic*. Wiley, 1967.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), 2008.

Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1), 1988.

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5), 2007.

Labix. python-constraint, 2010. URL `http://labix.org/python-constraint`.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on code generation and optimization*. IEEE Computer Society, 2004.

Samuel J. Leffler and Marshall Kirk McKusick. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007.

Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201325772.

Jochen Liedtke. On Micro-Kernel Construction. In *Symposium on Operating Systems Principles*, 1995.

Luigi Federico Menabrea and Ada Augusta. Sketch of The Analytical Engine, invented by Charles Babbage. In *Bibliothèque Universelle de Genève*, number 42. Genève, October 1842.

David Miller. How SKBs work, 2010. URL `http://vger.kernel.org/~davem/skb.html`.

David Mosberger and Larry Peterson. Making paths explicit in the Scout operating system. In *OSDI '96: Proceedings of the USENIX 2nd Symposium on OS Design and Implementation*, volume 30. ACM Association for computing machinery, 1996.

MPlayer authors. MPlayer HQ, 2010. URL `http://www.mplayerhq.hu/`.

Erich Nahum, Tsipora Barzilai, and Dilip D. Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(1), 2002.

NICTA. OKL4 for the HTC Dream, 2010. URL `http://ertos.nicta.com.au/software/okl4htcdream/`.

Object Management Group. CORBA 3.0.3, Common Object Request Broker Architecture (Core Specification), 2004-03-01, 2004.

OK Labs. OKL4 Microkernel, 2010. URL `http://wiki.ok-labs.com/Microkernel`.

Oracle. Java home page, February 2010. URL `http://java.sun.com/`.

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computing Systems*, 18(1), 2000.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1972.

Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. ACM, 2002.

Calton Pu and Henry Massalin. The Synthesis Kernel. *Computing systems: the journal of the USENIX Association*, 1(1), 1988.

Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutler, and Koen De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth international symposium on signal processing and information technology*, 2005.

Python Software Foundation. Python Programming Language – Official Website, 2010. URL `http://python.org`.

Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for system software. In *Proceedings of the 4th ACM Symposium on Operating Systems Design and Implementation*, 2000.

Arch D. Robison. Impact of economics on compiler optimization. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. ACM, 2001.

Douglas R. Smith. Kids: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1990.

John A. Stankovic, Ruiquing Zhu, Ram Poornalingam, Chenyang Lu, Zhedong Yu, Marty Humphrey, and Brian Ellis. VEST: An aspect-based composition tool for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

Sander Tichelaar, Stephane Ducasse, Serge Demeyer, and Oscar Niestrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution*, 2000.

TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

Alan Turing. Can a machine think? In *The World of Mathematics*, volume 4. Simon & Schuster, 1956.

Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM Press, 1998.

Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12), 1994.

Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.