Towards automatic performance optimisation of componentised systems

Nicholas FitzRoy-Dale
NICTA [*] and University of New South Wales
Sydney, Australia
nicholas.fitzroy-dale@nicta.com.au

Ihor Kuz
NICTA and University of New South Wales
Sydney, Australia
ihor.kuz@nicta.com.au

Abstract

Use of hardware-based memory protection to implement a componentised system is an effective way to enforce isolation between untrusted software components. Unfortunately this type of system design can lead to poor performance. Manual optimisation is error-prone and difficult. Instead, we describe a system to perform automatic optimisation of components, relying on three major functional units: a method to reconfigure the component system, simulations of each component in order to determine performance characteristics, and a system simulator that makes use of those characteristics to construct a ranking of optimisations. We start with a simple model and iteratively expand it until it is suitable for a wide variety of performance-measurement scenarios, and show that a small amount of information provided with each component allows for a wide variety of optimisation checks, such as scheduling, threading, and cache performance. We present our initial results with this system and discuss a number of interesting extensions.

1. Introduction

Using hardware-based memory protection is an effective way to enforce isolation between untrusted software components in computer systems when language techniques cannot be relied upon. Componentised software design (that is, designing software architectures as assemblies of components interacting through well-defined interfaces) provides a structured way of building such systems and both representing and enforcing the isolation at design and implementation time. Unfortunately past experience has shown that componentisation of system software (including modularisation and microkernel-based systems) often leads to poor performance of the resulting systems (Haeberlen et al. 2000). In particular the impact of component-to-component communication is a major cause of overall performance degradation.

This class of problems is caused by some system configuration or high-level design issue: interacting subsystems are not communicating efficiently enough; a priority or scheduler problem causes starvation; or the critical path simply involves too much code or data, reducing the efficiency of the CPU caches. Solving these sorts of problems manually is tedious and error-prone, particularly if the problem-solver is not familiar with all the code in the system. It is easy to reach an impasse in such a system in which the component implementer must know the characteristics of the complete system in order to produce efficient code, but the system assembler must know the details of each component's implementation in order to produce an efficient system.

We believe this problem is best solved by the component implementor providing additional information about each component in a way that can be used to *automatically* determine a components's performance in a given system configuration. There are three major advantages to automatic optimisation. Firstly, an automatic optimiser can check a large number of potential system configurations. This is important in whole-system optimisation, where the causes of performance problems are sometimes subtle and non-local. Secondly, if an automatic system is used, a lot of information can be encoded about the component, including special-case domain-specific optimisations that may otherwise be overlooked. Finally, the additional information required for automatic optimisation paves the way for automatic verification of the system configuration.

In Section 2 we describe the component system architecture and introduce an example component system. We then introduce the three major components of our optimisation tool in Section 3. In Section 4 we give some examples of the

[*] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

system, and present preliminary results. Finally, Section 5 discusses related work, and Section 6 concludes and outlines future research.

2. Componentised system architecture

When we refer to a *system* in this paper we mean an assembly of components interacting through well-defined interfaces. Our components are self-contained code objects containing at least one thread, and each component is separated from others in the system by hardware-mediated memory protection. The system layout is determined statically. This means that the set of components with which any given component may communicate (as well as any memory that the components will use for their communication) is determined when the system is compiled and linked. This combination of static configuration and hardware-mediated memory protection describes a mid-size embedded system, such as an advanced portable media player.

To illustrate some of the challenges involved, we introduce a running example: a networked video player. This is an embedded device which plays video received from a network interface on an LCD. Its architecture is shown informally in Figure 1. More formally, the system is described using an embedded component system with the following properties relevant to this paper:

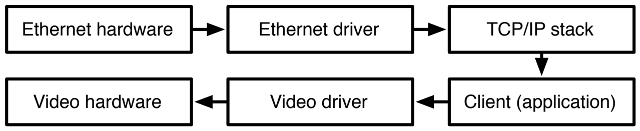

Figure 1. The networked video player

Components: A component is an independent piece of code which can only communicate with other components via well-defined *named interfaces*;

Connectors: Components interact with each other via *connectors*. A connector consists of two parts: the sender's portion, attached to the sending component; and the receiver's portion, attached to the receiving component. When communication occurs, the sender's portion must, at a minimum, marshal any data and notify the receiver's portion. The receiver's portion must (again at a minimum) unmarshal data and notify the receiving component. Connectors are extensible and may perform complex, multi-stage transactions. Connector portions communicate with their components through a function-call interface (they are essentially libraries);

Configuration: Components and connectors are *configured* with constants at compile-time. For example, a buffering connector may be configured with the size and maximum number of buffers it supports.

Typing: Connectors have a type. The type is meaningless to the component system itself, but serves to identify interchangeable connectors for optimisation purposes. For example, two connectors of type `stream` are functionally equivalent, but may have different performance characteristics.

Static layout: A system architecture consists of typed components joined by connectors. The layout is fixed at compile time.

As shown in Figure 1, this device contains a single *critical path* traversing all components — data moves from the network device driver to the LCD device driver via the TCP stack and the player application. This a data-intensive and performance-critical path: the right component must execute when data is available, and connectors between components must be fast enough that video is played smoothly. On battery-powered devices such as this one, improving performance beyond merely "usable" is beneficial as reducing overhead lowers power consumption and thus improves battery life. We will perform whole-system optimisation to improve the performance of this device.

3. System optimisation

The overall aim of the optimisation task is to take a system as input and produce a faster, yet otherwise functionally equivalent, system as output. In our system, the transformation occurs at the system architecture level: i.e. optimisation is achieved by changing the components and connectors in the system (by replacing them with other components and connectors), and not by modifying component and connector code itself (an orthogonal problem which we leave to the compiler).

We separate the problem into thee parts: the system-architecture transformation itself (the *optimisation functions*), simulation of individual components to determine their performance characteristics (*component simulations*), and the *system simulation* which uses optimisation-specific criteria to determine the performance of the transformed system relative to the original system (to determine whether the optimisation is an improvement). A high-level representation of this system is shown in Figure 2, and each part is discussed separately below.

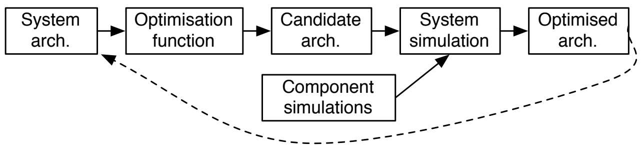

Figure 2. The optimisation architecture

3.1 Optimisation functions

Optimisations themselves are expressed by *optimisation functions*. These functions modify the system layout by

```
def stream_to_rbuf(node):
  if node.is_connector
      and node.ck_type == 'stream'
      and node.ck_name != 'RBufConnector'
      and node.client.ck_name == 'NIC':
    node.replace('RBufConnector')
```

Figure 3. The any-stream-to-RBuf optimisation function

adding, removing, or replacing components and connectors. Optimisation functions are written in Python (chosen because it is easy to learn and understand) and stored with the component or connector to which they apply. This allows for domain-specific optimisations, which only apply to a single component or connector.

Optimisation functions are supplied by the author of the component or connector and should represent the ways in which that component or connector can be used. This is one of the ways through which the component or connector author supplies additional information. Additionally, optimisation functions do not make any guarantee that the optimisation being implemented is beneficial. To determine whether a given optimisation is beneficial it is simply implemented on the system and its performance is simulated (see below). If there is a performance improvement, the optimisation is used on the real system.

The example optimisation function in Figure 3 illustrates the relevant properties. This particular function attempts to replace the outgoing connection from the network interface driver with a connector named `RBufConnector`, by checking that the supplied argument is of a valid type (it identifies itself as a "stream" connector, and it's connected to the network interface card component), and then by replacing it with a new connector. Optimisation functions are run once for every component and connector in the system — the current component or connector is passed as the function's only argument. The optimisation function references other components and connectors in the system through a relative-naming scheme: the `.client` and `.server` accessor functions for connectors, and through named interfaces for components. This style of relative referencing, borrowed from the Click router framework (Kohler et al. 2000), tends to make functions more readable. The function indicates success by modifying the system in some way. If the function is not successful (because some check failed), it should not make any modifications to the system state.

3.2 Component simulations

Component simulations describe the behaviour of a component or connector by providing a simplified implementation of it. A simulation function reduces the component to a set of properties that are essential for measuring its performance. The simulation is beneficial (compared to simply executing the component's code) because it executes quickly. This is a highly desirable property because the system simulator (discussed below) may execute a component simulation many times while coming up with an appropriate optimisation. Unfortunately, the amount of information that must be provided about the behaviour of a component or connector in order to provide for conceivable performance evaluation is quite large. For example, for accurate cache simulation the component simulation must return accurate information about the amount of, and location of, code and data. This information is not always readily available, and, more importantly, is not always necessary, as Section 4 discusses. We propose a modular approach, through which the component simulation can supply detailed information if it is available; if the information is not available then some classes of performance evaluation will not be possible, but others may be.

```
class sim_rbuf(Connector):
  def sim_send(self, system, data):
    # Data arrived from client. Pass it on.
    system.ipc(4)
    system.call(self.server, "data_in", data)
```

Figure 4. A simple RBufs connector simulation

A component or connector's simulation is implemented as a Python class, and should be supplied by the component implementor. The class contains a Python function for each component function to be simulated (since components and connectors communicate with a standard C function interface, there must be a simulation for every function in every interface of the component). Each function simulation calls one or more *system functions*, which represents component activity relevant to performance analysis. Figure 4 shows a simple example: a data-passing component dealing with data which is already shared in memory simply passes it from the client to the server. The component simulation is data-centric: primarily concerned with the movement of data across interfaces. Data movement is tracked using an abstraction: data are represented by an object called a *named data object* consisting of a unique identifier and a length. Memory sharing is performed by allowing two components simulations to both reference the same named data object. Objects can also be copied, representing real data copying. We go to this amount of trouble so we can accurately measure the cost of data copying in the system.

3.3 System simulation

The final of the three optimisation tasks is system simulation. The system simulation takes a given system architecture and simulates it using the component simulations in order to produce an ordering of optimisations. For example, for simulating raw throughput, a system simulation may be supplied with a starting component and an ending component. The system simulator calls the appropriate function in the simulation of the starting component, and accumulates performance data until the appropriate interface of the supplied end component simulation is called. A more sophisticated system simulation may take cache usage, concurrency, and scheduling into account. When an ordering is produced, the top-ranked optimised system becomes the new base system, and the process may be repeated.

The system simulation effectively determines the abstraction that component simulations must conform to. We must therefore anticipate the type of information we need from the component simulation. We are currently interested in modelling data flow (for throughput estimation), CPU usage (for load estimation), inter-component control flow (required in order to correctly simulate component interactions), and cache utilisation. The system simulator must therefore make it possible for component simulations to indicate data flow, CPU usage etc. to the system simulator, which component simulations do by calling appropriate functions on the system simulator.

3.4 Putting it together

Optimisation begins with the system designer running the system simulator. This simulator takes an architecture description as input. First, it generates a base-line performance value for the original system. This value is specific to the type of optimisation required: for example, for a throughput optimisation, it may represent the total number of CPU cycles required to send data from one component to another. It then generates a list of optimisations to perform. This set is the collection of all optimisations included with every component and connector included in the system architecture. The first function in the list is then run, generating a new system architecture. This system architecture is then simulated to generate a performance value for the new system. This process is then performed for each remaining optimisation. The system with the largest performance improvement is then selected, and the process is repeated until the system's performance cannot be improved further. This iterative process is required because some optimisatons may be mutually exclusive.

4. Example optimisations

To illustrate the potential of this system, we present some preliminary results measuring system throughput. These results refer to the networked video player described in Figure 1. This system uses components connected via connectors of type `stream`, which pass blocks of data. We investigated two styles of stream connectors: UNIX pipe style, in which data are copied from component to component by the kernel, and RBufs style (Leslie et al. 1996), in which data are passed via shared memory.

The optimisation function used is very similar to that shown in Figure 3, without the requirement that the source component be the NIC – i.e., all streams interfaces, initially modelled using UNIX pipes are converted to RBufs.

```
class sim_pipe(Connector):
  def sim_send(self, system, data):
    system.pipe(data)
    system.call(self.server, "data_in", new_data)
```

Figure 5. A UNIX pipes connector simulation

Name	Cycles
$\text{notify}_{\text{pipe}}$	2189
$\text{notify}_{\text{sem}}$	471
ipc_{1k}	3761
ctx	1596
$\text{ctx}_{\text{notify}}$	3320
mem_{1K}	420

Table 1. Microbenchmark results in CPU cycles – Linux 2.6, Core 2 Duo (single core), 2.2GHz.

The component simulation for RBufs is the same as that shown in Figure 4. The component simulation for UNIX pipes is shown in Figure 5.

The system simulation is a straightforward throughput estimation taking costs of IPC and data copying into account. It contains a rudimentary network device simulator capable of generating packets for the NIC component simulation. The scheduler switches components only when they call other components — i.e. scheduling is co-operative. This is not unrealistic as it approximates the common case where only one component is runnable. The system simulation is parameterised by microbenchmarked results of performance of various tasks under Linux. Results of these microbenchmarks (in CPU cycles) are summarised in Table 1.

Under Linux, context-switch time is highly dependent on the system call performing the context switch. As a result, explicit context switches implemented using Linux's high-performance *futex* mechanism ($\text{notify}_{\text{sem}}$ in the figure) are significantly faster than implicit context switches resulting from sending data into a pipe ($\text{notify}_{\text{pipe}}$ in the figure), but in the former case data must be copied separately. Component simulation calls result in the system simulator applying microbenchmark figures in order to rank the optimised system: for example, `system.ipc()` in the RBufs simulation results in an additional $\text{notify}_{\text{sem}}$ in the system simulation for that path.

To evaluate the model, we implemented two componentised systems: an RBufs-style system with memory sharing, and a UNIX pipes implementation. Both implemented systems consisted of three classes of component: a packet source, representing the Ethernet hardware; four packet transports, representing the Ethernet driver, stack, client, and video driver; and a packet sink, representing the video hardware. For each system, we apply the system simulation described above and compare the results with the actual per-

System	Model	Benchmark	Ratio
RBufs shared	4818	6119	0.79:1
UNIX pipes	18805	18678	1.01:1

Table 2. Performance model vs measured results: cost in CPU cycles to send a single packet through the system.

formance numbers obtained by running the code. The RBuf connector improves performance through queueing. In this case the queue length for the RBufs connector was set to 10 packets. The results of this evaluation are shown in Table 2. These results show that the cost model is capable of producing a correct ordering of systems in terms of overall latency. The accuracy of the model can be seen as the ratio between the predicted performance value and the benchmarked value, shown in the 'ratio' column.

The RBufs prediction is less accurate than the UNIX pipes prediction. This relects the fact that the UNIX pipes measurement takes into account all kernel code involved in implementing the pipe, including cache effects, whereas in the RBufs example more non-kernel code is executed as the buffer manipulation is performed in user-space. The discrepancy points to the usefuless of a more detailed component simulation, but it nonetheless seems to indicate that such a detailed simulation is not necessary for at least some component optimisation tasks, i.e., those focusing on throughput performance in network-limited systems. In fact, pipelines involving network data can contribute significantly to system-wide CPU and memory bandwidth utilisation (Druschel et al. 1993).

However, most optimisation tasks are not this simple. In the following sections we show how the basic model can be extended to support cache simulation and scheduling.

4.1 Measuring cache effects

Measuring cache effects requires extensive pre-analysis of the component or connector. Two classes of system functions are introduced: CPU usage functions, and data-manipulation functions.

CPU usage functions : The component simulation supplies CPU usage in the form of an instruction range (as a range of virtual addresses) and the probability of its execution. This is implemented as `system.cpu(start, end, prob = p)`, where *start* and *end* correspond with the beginning and end of the virtual address range. This may also be used to predict system CPU usage.

Data manipulation functions : The use of named data objects (see subsection 3.2) allows for accurate cache simulation of data usage in most cases. If a large amount of component-private state is manipulated, the component simulation class can use the same data-manipulation functions but retain the data within the class.

4.2 Scheduling

The scheduling simulation may be performed entirely by the system simulator, given enough auxilliary information from the components. A simplistic scheduler, such as the one used to measure throughput in the results above, can simply pass control to an IPCed component, resulting in what is effectively co-operative scheduling. However, pre-emptive scheduling can also be simulated by treating the component simulations as continuations which may be paused and resumed at any time. For example, a component may indicate CPU usage with a `system.cpu()` call as described above, which may trigger the system simulator to simulate a context switch. The simulator then determines the amount of code that would have been executed before the switch, selects a new component, and continues the simulation. When it returns to the original component, the simulator re-starts it from where it left off. Importantly, cache simulation remains valid through this process.

4.3 Managing non-determinism

Up until now we have assumed that component simulations always do the same thing given the same input data. This is not always true, so some way of expressing non-determinism to the system simulator may improve simulation accuracy. However, having component simulations literally be non-deterministic would result in a significantly more complicated system simulator. We compromise by defining the system simulation function `system.maybe(p, func)`, where `func` is a function representing a particular simulated component execution and `p` is the probability that this execution occurs. The sum of all probabilities should be 1. System simulators can use this information to investigate a performance landscape, testing different optimisations to determine the effect on individual components and the whole system. Such a system simulator may make use of simulated annealing or similar to reach an acceptable performance level.

4.4 Potentially-unsafe optimisations

Many useful optimisations change the system configuration in unsafe ways. As a single example, consider a window server which multiplexes graphical information from many applications onto a single frame buffer. In such a system it is important that one application can't draw over another application's window. A simple way to implement this arrangement is to require all components to communicate with the window server via IPC, but this is rather slow as each drawing operation incurs a protection-domain-crossing performance hit. A faster option is to allow applications to draw directly to the shared frame buffer via a graphics library, but this requires that applications be trusted not to draw beyond their clipping region. Similar problems arise with other common optimisations, such as ILP in a network stack.

A somewhat practical solution is simply to require any optimisation that may compromise the system's safety assumptions to notify the system designer of that fact. This solution may be workable for simple cases, but is rather unsatisfying because it once again requires the system designer to know the behaviour of each component in detail. Automated component safety checking is one alternative: a symbolic execution engine for safety verification is future work.

5. Related work

We focus on performance prediction to determine the efficacy of optimisation, but performance prediction of componentised systems is not a new field. Yacoub proposes a technique for performance measurement that involves measurements taken around component-to-component connections ("glue code" in his terminology) with a method similar to ours (Yacoub 2002). However, the described method is purely analytical, with no predictive power. Chen et al. propose a model that takes a similar idea to ours (isolating the costs of the framework) by measuring the overheads of various parts of an Enterprise JavaBeans or CORBA framework (Chen et al. 2005). The result is a sophisticated cost model. Unlike our approach, however, this model is rather coarse-grained (measuring database transaction times, for example) and focuses on the interaction between the application and an end user, rather than component-to-component operations.

Becker et al. note that a successful performance evaluation technique should be accurate; adaptable; cost-effective; composable (able to work at different levels of abstraction), and scalable; should be *analyzable* (give some insight into the design flaw that is causing the performance bottleneck); and should be *universal* (easily-adaptable to different component technologies) (Becker et al. 2006). The technique we have described performs very well on some of these criteria due to its isolation of, and focus on, component connectors rather than components themselves. As a result, it is highly adaptable in the domain of interest (changing connectors and changing platforms), provides a high degree of analysability (since each connection is individually ranked), is universal (since the small set of component-technology-specific costs form the basis of the prediction), and cost-effective (since performance information need only be derived once for each re-usable connector).

6. Conclusion

In this paper we described an expandable system for performance estimation of componentised systems. We also presented some some initial results of the system applied to the task of throughput optimisation, which show promising correlation with measured performance numbers (ratio between 0.79:1 and 1.01:1). This work is nonetheless very much a work in progress. We are working on the following additional features, roughly in planned chronological order:

Cache and scheduler experiments: Support for cache and scheduler simulation is described above, but we are keen to back up the description with experiments and results as per the initial throughput examples.

Component modes: Typically components have modes in which behaviour is dramatically different: devices may be initialised or uninitialised; streaming connectors have a steady state and an overload state, etc. The system currently makes the assumption that there is only one interesting mode; incorporating more is future work.

Integration with existing component systems: The component system described in Section 2 is a minor extension of the CAmkES component architecture (Kuz et al. 2007). Integration with CAmkES will allow us to experiment on embedded systems based on the L4 microkernel.

Symbolic execution: We are interested in exploring the application of symbolic execution or other static analysis techniques to automatically generate the component model, taking the burden off the component designer.

References

S. Becker, L. Grunske, R. Mirandola, and S. Overhage. Performance prediction of component-based systems — a survey from an engineering perspective. In *Architecting Systems with Trustworthy Components*, volume 3938, pages 169–192. Springer-Verlag, 2006.

Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *J. Syst. Softw.*, 74 (1):35–43, 2005. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2003.05.005.

Peter Druschel, Mark B. Abbott, Michael A. Pagals, and Larry L. Peterson. Network subsystems design. *IEEE Network*, 7(4):8–17, 1993. URL `citeseer.ist.psu.edu/druschel93network.html`.

Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *WIESS'00: Proceedings of the 1st conference on Industrial Experiences with Systems Software*, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/354871.354874.

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007.

Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, 1996.

Sherif M. Yacoub. Performance analysis of component-based applications. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 299–315, London, UK, 2002. Springer-Verlag. ISBN 3-540-43985-4.