Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level

Gernot Heiser* Kevin Elphinstone Ihor Kuz Gerwin Klein Stefan M. Petters
NICTA† and University of New South Wales
Sydney, Australia
{firstname.lastname}@nicta.com.au

ABSTRACT

As computer systems become increasingly mission-critical, used in life-critical situations, and relied upon to protect intellectual property, operating-system reliability is becoming an ever growing concern. In the past, mission- and life-critical embedded systems consisted of simple microcontrollers running a small amount of software that could be validated using traditional and informal techniques. However, with the growth of software complexity, traditional techniques for ensuring software reliability have not been able to keep up, leading to an overall degradation of reliability. This paper argues that microkernels are the best approach for delivering truly trustworthy computer systems in the foreseeable future. It presents the NICTA operating-systems research vision, centred around the L4 microkernel and based on four core projects. The *seL4* project is designing an improved API for a secure microkernel, *L4.verified* will produce a full formal verification of the microkernel, *Potoroo* combines execution-time measurements with static analysis to determine the worst case execution profiles of the kernel, and *CAmkES* provides a component architecture for building systems that use the microkernel. Through close collaboration with *Open Kernel Labs* (a NICTA spinoff) the research output of these projects will make its way into products over the next few years.

1. INTRODUCTION

Operating-system reliability is a growing concern as computer systems are becoming increasingly mission-critical for many organisations. Furthermore, embedded computing systems are increasingly used to handle sensitive personal data. For example, in some countries people routinely perform financial transactions via their mobile phones, and an increasing number of countries use embedded computer systems to store and access sensitive medical information. Embedded systems are also increasingly being used in life-critical situations, such as aircraft, automobiles and medical devices. Finally, personal computers as well as mobile devices are increasingly used to access valuable intellectual property, such as artistic media content, which the owners authorise for use under very restricted conditions.

This raises concerns about the reliability and trustworthiness of such systems. In the past, mission- and life-critical embedded systems consisted mostly of simple microcontrollers running a very small amount of software, which could be validated with traditional (informal) means. The strong growth in functionality (and consequently complexity) of these devices makes such validation methods less and less satisfactory. In a nutshell, software complexity is increasing much faster than the power of techniques for ensuring software reliability, leading to an overall degradation of reliability.

This problem of software reliability must be addressed at all levels of the software stack, and with a combination of approaches. In particular, the operating system must be a core part of the approach. Any guarantee of application-software correctness is useless if the application runs on top of a faulty operating system, which can interfere in arbitrary ways with the operation of the application software.

We argue that microkernels form the basis of the only feasible approach with a chance of delivering truly trustworthy computer systems in the foreseeable future. This is the core of the operating-systems research agenda at NICTA, Australia's centre of excellence in ICT research. In this paper we present the NICTA OS research vision, an overview of the research agenda that aims to deliver on this vision, and discuss the progress to date and the steps taken to deploy the research outputs for real-world use.

The remainder of this paper is structured as follows. Section 2 presents microkernels, and specifically L4, as a basis for trustworthy embedded systems . This is followed by an overview of the NICTA OS research and commercialisation agenda in Section 3. In particular we focus on four core projects: *seL4*, *L4.verified*, *Potoroo*, and *CAmkES*. Section 4 compares this research agenda to current related work. We note that while there is activity elsewhere on various subsets of our agenda, none is as comprehensive and ambitious as our projects. Finally Section 5 closes with a summary and conclusions.

2. OPERATING-SYSTEM COMPLEXITY AND ROBUSTNESS

2.1 Main-stream operating systems

The *kernel* of an operating system is usually defined as the part of the software that executes in the processor's privileged mode, giving it unrestricted access to all hardware resources and functions. In customary "monolithic" system design, the kernel provides most of the core OS functionality, including interrupt handling, memory management, access control, device drivers, network stacks, file systems, etc.

Novel hardware, increased hardware variety (e.g., multithreading, multicores, bus-connected SMP, NUMA) and more varied usage contexts (ranging from embedded systems to supercomputers) have lead to a strong growth in the number and complexity of OS ser-

*Also with Open Kernel Labs

†NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

vices. This is reflected by an immense growth in the size of kernel code. The Linux kernel now comprises about 4.1 millions lines of code (MLOC), having grown by a factor of 33 within 13 years (see Figure 1). Windows Vista is said to have 20 MLOC of kernel code.

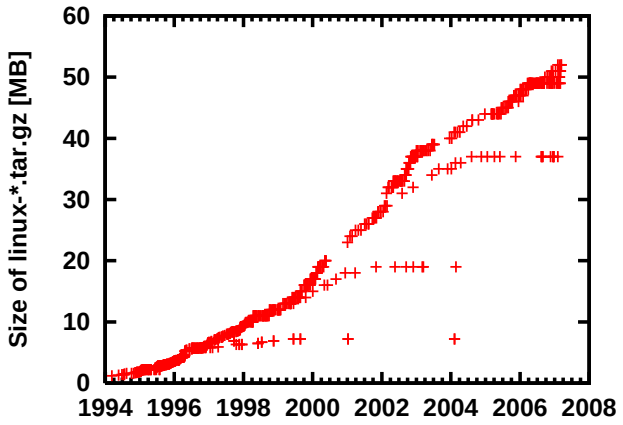

Figure 1: Size of the Linux kernel source

This growth is developing into a formidable challenge to system reliability. Well-engineered code can be expected to have of the order of 2 defects per kLOC [42], which puts the number of kernel bugs in modern operating systems literally into the tens of thousands. Since all this code executes in privileged mode, each bug has the potential to cause arbitrary damage. In fact, the majority of OS code is contained in drivers, which are on average of much lesser quality [6], meaning that the number of kernel bugs is probably up in the hundreds of thousands.

While we are not aware of formal studies of system reliability, anecdotal evidence certainly supports the view that it is deteriorating. Five years ago, the author's Linux desktop was essentially rebooted only after power outages or for hardware upgrades, the laptop maybe once a month. These days the laptop needs to be rebooted daily on heavy use, the desktop at least monthly.

While not all of this code is actually loaded in any particular system, the active code still accounts for in the order of a million LOC. Even systems stripped down to a bare minimum, such as embedded versions of Linux or Windows, will contain several 100 kLOC of kernel code.

Even these minimal systems are increasing in size and complexity. A recent study of the Linux kernel source [50] found that the sizes of Linux kernel modules have been growing linearly with the version number. More worrisome, the interdependencies of modules via global variables increased exponentially! This complexity increase can be expected to lead to a growth in number of bugs that is super-linear with code size.

2.2 Microkernels

Microkernels represent an alternative to monolithic systems as an OS design paradigm. While there are a range of views of what constitutes a microkernel, the most precise definition was given by Jochen Liedtke, who can be considered the father of modern microkernels:

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality. [40]

Note that this definition does not leave any space for "nanokernels", "picokernels", "femtokernels" and the like, kernels that claim to be even smaller than a microkernel (unless they form only a part of the kernel, meaning that there must be other code that also executes at the highest hardware privilege level).

A microkernel in this sense is a minimal substrate upon which the actual operating-system services are implemented at user level. Implicit in the definition is the notion that the microkernel itself provides no services, only mechanisms for implementing services. The microkernel also needs to be free of policies, as it would otherwise not be possible to implement arbitrary systems (and their policies) on top of it.

We know of no real kernel which is a microkernel in the strict sense of the above definition. In practice, compromises are made by including extra functionality in the kernel for performance reasons. In the following we will use the term *microkernel* in a slightly looser sense, applying it to systems that also contain a "small" amount of non-essential code (say, an order of magnitude less than the essential kernel code).

2.3 A brief history of microkernels

Like so many concepts in computer science, microkernels go back to at least the 1970's. The first known system that deserves to be called a microkernel is Brinch Hansen's *Nucleus* [4]. It clearly enunciated the basic ideas behind microkernels: the kernel supplies only primitives for process control and inter-process communication, the kernel has no built-in strategies, and all operating system policy is implemented as processes outside the kernel. Hydra [38], a combined hardware-software approach that explored alternative systems structures, took up Brinch Hansen's ideas and explicitly formulated the principle of separation of mechanism (in the kernel) and policy (in userland). The Mach system [47], a software-only approach based on the same principles, coined the term "microkernel". Ironically, Mach, which featured a code size of over 150 kLOC and over 200 system calls, fell far short of meeting the minimality requirement expressed in the above definition of a microkernel.

There were a number of contemporaries of Mach [27, 49], and a large number of followers — microkernels were very much *en vogue* in the late 80's. However, the enthusiasm gave way to disappointment in the early 90's, when it turned out that microkernel-based systems all exhibited poor performance. This lead to some spectacular failures, most notoriously IBM's $2G disaster with its Workplace OS [15]. Other commercial Mach projects, such as OSF/1 and NextStep (including its successor, Mac OS X), ended up "co-locating" OS services with the Mach kernel, giving up on any pretence of using a microkernel structure.

Inherent in microkernels is the need to invoke services in different address spaces, making the performance of kernel-provided inter-process communication (IPC) mechanisms critical. Kernels of the day exhibited IPC costs in the order of 100μs, almost irrespective of hardware [39].

An analysis of the performance of Unix on Mach, compared to na-

tive Unix, concluded that the performance problems were "inherent in the OS structure" [5], in other words, a result of the microkernel approach. However, Liedtke re-analysed the same data and showed that performance of the Mach-based system was limited by capacity cache misses in the Mach kernel [40] — Mach's footprint was too big.

Liedtke also showed that microkernel IPC can be made fast, by taking minimality seriously, and taking a very careful approach to design and implementation, with extensive micro-optimisation of algorithms and data structures [39]. While Mach and its contemporaries exhibited IPC costs that were 1–2 orders of magnitude above the architectural limit, his L4 kernel came within tens of percent of that limit, outperforming Mach by more than an order of magnitude.

Unfortunately, the Mach experience had already given microkernels a very bad image, and Liedtke's results were largely ignored. The academic community by and large lost all interest in microkernels. Instead, virtual machines [20] are experiencing a renaissance [1], driven by a need to provide stronger resource isolation between subsystems than what is offered by mainstream operating systems, and the desire to run multiple operating systems on the same hardware platform. Virtual machines have a lot in common with microkernel, in the sense that both are (potentially) small kernels with very limited functionality. In fact, microkernels have long been used as virtual-machine monitors [21, 24]. Interestingly, most hypervisors seem much larger (in terms of code size) than L4.

2.4 Microkernels in the embedded-systems domain

Despite their general abandonment, microkernels did survive in niche areas, especially in the embedded-systems industry. QNX [27], while originally not performing much better than Mach, is in widespread use as a real-time OS, and Integrity OS from Green Hills Software is well established in the military and aerospace domain. Integrity has limited functionality (less than L4 for example), which makes it unsuitable for systems that require efficient resource sharing between protection domains, but this makes it easier to validate the kernel for correctness and timeliness — it is presently undergoing Common Criteria EAL6 certification.

More recently, virtualisation has become an important technology in parts of the embedded-systems industry. Unlike the server domain, where virtualisation is used to improve utilisation and performance separation (QoS), in the embedded world this trend is driven by the desire to use different OS environments concurrently and cooperatively. The industry is going through a transition from closed systems, well served by simple real-time OSes (RTOSes) without memory protection, to open systems comparable in power and functionality to personal computers, requiring modern OS technology and standard APIs. Virtualisation supports the co-existence of a high-level OS, such as embedded versions of Linux or Windows, with an RTOS that supports (legacy) real-time software. Unlike the server domain, the emphasis is less on separation, as the nature of an embedded system requires a strong cooperation of all components, including high-bandwidth, low-latency communication between subsystems (virtual machines).

L4 has repeatedly demonstrated its suitability as a high-performance virtual-machine monitor for server and desktop systems [24, 36, 37]. Its small size makes it a suitable candidate for embedded systems as well, and its emphasis on high communication performance makes it superior to plain hypervisors. It is therefore not surprising that L4 is increasingly used in embedded systems.

2.5 Towards a trustworthy TCB

However, using a microkernel as simply a hypervisor not only underutilises the kernel, it also does little to address the problem of system robustness and trustworthiness [29]. A hypervisor is designed to run a complete operating system in each virtual machine, and in general does not provide support for running applications natively. In contrast, a microkernel is designed as a platform on which minimal services can be provided. As such, a microkernel is an excellent approach to providing a minimal trusted computing base (TCB).

Making the TCB small has obvious advantages: Since software is buggy, minimising the amount of software has the benefit of minimising security-critical bugs. The question arises whether this can be taken to the next step: Can we completely eliminate bugs from the TCB, thereby making it not only *trusted*, but actually *trustworthy*?

We believe that this question can be answered in the affirmative — under the right conditions. We believe that if the TCB is small enough, it can be made correct in a strict sense, and *shown to be correct*.

Size is the key. In the end, "shown to be correct" implies a form of proof. This can take the form of exhaustive testing, or formal mathematical proof. The former is infeasible for anything larger than a few dozen, or at best a few hundred, LOC. The latter scales somewhat better, but is still only feasible for of the order of ten thousand lines of code. This limits the size of the kernel, since for the purpose of verification it cannot be subdivided. Hence, keeping the kernel small, at around 10 kLOC, is the key to verification. Microkernels are the only class of kernels which fit this restriction while at the same time providing mechanisms of sufficient generality to support the construction of arbitrary systems. Virtual machines monitors, which can be made similarly small or even smaller, lack this generality. Specifically, the L4 microkernel is roughly 10 kLOC, and therefore a prime candidate for a trustworthy TCB [57].

3. THE NICTA AGENDA: TRUSTWORTHY EMBEDDED SOFTWARE

In this section we describe our research program, which aims to develop a demonstrably-correct foundation for trustworthy systems. The program primarily targets embedded systems because we see the greatest need, as well as the greatest opportunities in this field.

The need has been discussed above; the opportunities arise due to significant changes that are happening in the embedded-systems industry. There is a growing realisation in the industry that the widely-used unprotected real-time operating systems (RTOSes) are reaching their use-by date, and companies are looking for better solutions. This development, which is happening in different verticals at different times (the mobile-phone handset industry is currently in the middle of it) is creating a unique opportunity: a once-in-a-generation chance to change operating-system technology. If the right solution is presented, it has a real chance of being adopted.

The program has four main research components: kernel API,

kernel verification, temporal analysis, and component technology. These will be discussed below, followed by an accompanying fifth component: commercialisation.

3.1 Kernel API: seL4

The seL4 (secure embedded L4) project aims to evolve L4 into a platform for constructing secure embedded systems. The project's close relationship with the L4.verified project (Section 3.2) also creates interesting issues beyond those immediately driven by the desire for security. Some of the issues the project is tackling specifically are kernel physical memory management, the management of authority to invoke kernel services, the provision of a development environment amenable to both kernel prototyping and formal verification of the end result, and development of kernel mechanisms that are highly flexible, as kernel changes trigger potentially expensive re-verification.

Physical memory is a limited resource that must be managed carefully, especially inside the kernel. Excessive consumption of kernel memory via application use of kernel services can result in denial of service to other applications due to memory exhaustion. In the embedded domain, the common approaches of quotas, or treating kernel physical memory as a cache of data stored elsewhere are less appropriate. It is difficult to avoid underutilisation with quotas, and caching kernel data does not provide the temporal guarantees needed for real-time systems. Our approach involves avoiding implicit memory allocation in the kernel entirely. Instead it provides a model to applications that enables them to explicitly allocate kernel data structures, by giving them authority to a subset of physical memory. The model has the benefit of providing a direct and simple relationship between possession of authority and kernel memory consumption, giving higher-level systems the ability to confine kernel memory to particular applications, or even implement traditional quotas and caching. Further details can be found in [10].

The seL4 kernel extends the original L4 kernel model with capabilities — all kernel services are accessed by invoking capabilities, with no default inherent authority conferred to processes. Capabilities, when confined to subsystems by construction, can achieve spatial partitioning of subsystems to provide strong fault isolation guarantees. Given the explicit memory management model, these isolation guarantees also extend to kernel physical memory consumed when providing services to applications.

The seL4 prototype uses a novel development approach for an operating system kernel [11]. Even small operating systems are complex enough to require prototyping to validate ideas. This is typically done by either exploring implementation details with an eye for efficiency, or providing a concrete implementation to gain experience in high-level system construction. Prototyping in a low-level language like C exposes the designer to time consuming debugging, combined with complex low-level hardware details. It is also difficult to extract a formal model of the kernel from a low-level C implementation. To address these issues, the seL4 prototype has been written in the functional programming language Haskell. The Haskell implementation evolved over time from a simple model to a complete kernel implementation, encompassing the management of hardware artifacts such as page tables. It exposed many implementation issues during the design of the kernel, while at the same time providing a precise basis for automatic extraction of a formal model of the implementation for use in a theorem-proving environment.

The prototype kernel model is detailed enough to process a stream of events resembling the exceptions a real kernel would process to manage applications on real hardware. We have coupled the kernel model with the user-level execution component of the QEMU machine simulator. Exceptions, which would normally result in the transfer to privileged execution, instead are used to drive the kernel model, which induces changes to user-level applications to mimic the behaviour of a real kernel running on the simulated machine. We have used this environment to construct high-level system software and applications on the new kernel, prior to a real bare-metal implementation existing. We also have a bare-metal implementation in C underway, based on the mature Haskell prototype. A high-performance C implementation, suitable for formal verification, is one of the core deliverables of the project.

We are now in the position where the Haskell prototype forms a platform for feedback and change based on issues discovered in implementing the model itself, issues discovered in its formalisation, issues uncovered by its use by high-level system software, and finally issues uncovered by re-implementation in C. We have found the functional prototype a productive *lingua franca* for the kernel designers, the formal methods practitioners, and the application developers. Further details of our approach can be found in [9].

Finally, we recognise that success in building a formally verified implementation of a secure kernel will be short-lived if the kernel is insufficiently general to support a broad class of systems. Kernel changes to support specific applications invalidate implementation proofs which can be time-consuming to re-prove. This adds stronger than usual motivation for the separation of policy and mechanism [38].

Our hope is that, with the benefit of 13 years of experience with microkernel API design, we will be able to produce something that approaches the stability of a hardware interface — a "kernel machine" that provides sufficient mechanisms to provide the isolation guarantees we seek, whilst also offering a very general machine for higher-level system construction. We will view ourselves successful if the attitude towards seL4 becomes similar to that towards hardware, where the natural tendency is to always work with it (or to some extent, around it), with changes to the underlying platform being an infrequent occurrence of last resort.

As the project currently stands, we have a mature implementation of seL4 in Haskell, together with infrastructure to run it in conjunction with various user-level simulators to support native development on the new API. We also have a pen-and-paper proof of the isolation ability of the new API. Additionally, we have an immature C version of the kernel, and are continuing to build infrastructure for performance analysis of the C implementation (using both simulated and real hardware environments).

3.2 Kernel verification: L4.verified

The critical component of the whole program is the formal correctness proof of the kernel. The L4.verified project aims to show, in the theorem prover Isabelle/HOL [45], that the high-performance C implementation produced by the seL4 project conforms to its abstract specification. The project uses three levels of specification: an abstract description of correct kernel behaviour in Isabelle/HOL, an executable specification that coincides with the Haskell prototype developed in the seL4 project, and the C implementation itself that is parsed directly into the theorem prover. We then prove that each layer is a formal refinement of the layer above. This implies

that all behaviours of the C code are subsumed in the abstract specification and that all Hoare-Logic properties and system invariants of the abstract specification are true of the C implementation.

Because full verification is commonly thought to be infeasible, the highest software certification levels (e.g., Common Criteria EAL7) currently require machine checked formal proof only for the high level design, which in some cases might be even more abstract than our top-level specification. Any correspondence between implementation and design is left to semi-formal arguments and code inspection. The L4.verified project goes beyond these requirements in providing exact, machine checked correspondence between formal model and actual code for a general purpose OS kernel.

The current state of the verification is that the seL4 kernel is precisely specified at both specification levels, with about 3.5 kLOC on the most abstract and 7 kLOC on the executable level. The C implementation is nearing completion and the proof that the executable specification formally refines the abstract kernel model is about 90% complete with 48 kLOC of Isabelle proof scripts. This includes a large number of system invariants that describe safe operation of the kernel and ensure consistency of internal data structures through all possible executions.

Additionally, the project has produced a number of proof infrastructure components, such as a refinement calculus for monadic functional programs, a Hoare-Logic for such programs with automated verification condition generator, a translator from Haskell into Isabelle/HOL [9, 12], an extensive proof library for low-level n-bit machine words, a formal model of the ARM architecture and instruction set, a detailed memory model for low-level pointer programs in C that is amenable to abstract separation logic reasoning, and a significant case study of this logic on the current L4 kernel memory allocator [58].

The refinement proof between abstract and executable specification has provided significant feedback into the design cycle, so far leading to 37 patches in the Haskell prototype and 109 changes to the abstract specification. Not all of the changes in either of the specifications were related to correctness issues or bugs. A large percentage were for proof convenience, making functions more general or more obviously correct (relying more on local properties instead of global system invariants), which was often easier to do on the abstract side.

Nevertheless, it is interesting to note that the Haskell prototype had a significantly smaller number of defects such as typos or missing definitions than would normally be expected for a new kernel. We believe that this is due to the fact that the Haskell prototype was repeatedly run against application code and extensively validated. The defects that were discovered in this relatively well-tested code were corner cases in the implementation, typos in rarely executed, but critical parts of the kernel, and design issues like unbounded execution time or unexpected behaviours. Unsurprisingly, these were implementation defects of the kind that may lurk for years in well-tested production code with the potential for system crashes or security exploits. It is precisely the absence of such defects that the formal verification guarantees.

The main remaining proof is the refinement step between the executable specification and the C implementation. The project is planned to finish in mid 2008.

3.3 Temporal analysis: Potoroo

For many embedded systems, functional correctness, as will be established by L4.verified, is not enough. Many embedded systems, particularly those where correctness is important, are real-time systems, in which case timeliness is as important as functional correctness. However, the temporal behaviour of the complete system can only be analysed if the temporal behaviour (or at least the temporal bounds) of the underlying kernel is known. The goal of the Potoroo project is, therefore, to develop a complete timing model of the kernel.

There are two basic approaches to this: static analysis and measurement. Static analysis is popular in academia, but generally shunned by industry. The reason is that it relies on accurate timing models of the underlying hardware, which are hard if not impossible to come by (often the manufacturers themselves may not know). Furthermore, the complexity of the timing behaviour of modern embedded processors, with multiple levels of caches, long pipelines, branch prediction units and other complications, make static analysis increasingly difficult. In general, static analysis is limited to simple or small pieces of code, simple processor models, and produces wildly-pessimistic results for all but the most trivial systems. Modern architecture features such as out-of-order execution or dynamic branch prediction do not lend themselves to static analysis due to their dynamic nature.

The method of choice in industry is measurement. However, there are no systematic measurement procedures that are guaranteed to produce reliable upper bounds on latencies of real-life code of any appreciable complexity. The standard approach in industry can be summarised as heavy benchmarking, and then applying a "safety factor". This obviously provides no guarantees. In the case of safety-critical systems manual analysis of the code is still practised, but obviously does not scale well.

The approach applied to the analysis of the kernel needs to be manageable from a complexity point of view, support easy migration between hardware architectures, and provide guarantees for the resulting worst-case execution times (WCET). In order to achieve this, we combine concepts from both static analysis as well as the dynamic, measurement-based analysis methods, similar to earlier work [2]. We measure the execution time at the basic block level. This is motivated by the fact that the best model of the system is the system itself. The measurements are used to create an execution time profile (ETP), which describes the temporal behaviour in greater detail than would a single number, as provided with a WCET. The measurements avoid the requirement of detailed hardware architecture modelling and are thus subject to neither an error prone exploration stage, nor the usual problems with portability.

We combine the measurements using a tree representation of the code ensuring that any possible combination of paths is accounted for and removing the largest contributor to the variability of execution times. The second largest source of execution-time variability is due to cache misses, and is eliminated in our approach by using a cache model derived from the hardware (e.g., [31]).

The resulting timing profile provides a highly accurate picture of the kernel's timing behaviour, but offers no guarantees as to the measurement coverage. Accordingly, we next address the problem of verifying that the supplied measurements cover all possible system behaviours with respect to execution time. We achieve this by considering a simpler problem of proving that all possible cache

miss counts have been observed during measurements. The cache miss counts are easily extracted from the timing profile, and are highly representative of programs behaviour; in other words, if all possible cache miss counts have been observed during measurement, then the measurement coverage is exhaustive with respect to execution time [51].

We solve the problem by using static analysis to extract from the kernel a function (program) that dynamically computes cache miss counts. This is achieved by first translating the kernel's binary image itself into a purely-functional program that readily lends itself to such symbolic manipulation. We then consider the problem of proving that the range of the computed cache miss function is no greater than that observed during measurement. Our algorithm proceeds by computing an inverse of the cache miss function (i.e., a function from cache miss counts to sets of machine states) and applying a novel form of range analysis to the inverted function in order to either prove totality of the supplied set of cache miss counts or disprove it by arriving at a counter-example. As a result, Potoroo provides a convenient framework for "debugging" measurement suites in order to achieve adequate coverage. The cache model also aims to tighten the resulting WCET and profiles during the combination stage by establishing dependencies between different units measured.

The first stage of the project, providing the measurement and analysis infrastructure, is close to completion. The second stage covering the static analysis of the cache behaviour of the kernel is scheduled for completion by mid 2008.

3.4 Component technology: CAmkES

While the above projects focus on the microkernel itself, providing complete embedded-systems solutions requires that we also focus on building the software systems that use the kernel. The goal is to build reliable and trustworthy systems, and to do so we must take full advantage of what the kernel has to offer. Doing this requires that the system be designed and built as a highly structured, componentised, software architecture, with individual system services contained in separate components. These components must have well defined interfaces, explicitly specifying any interaction points, and clearly detailing interaction between components. The aim of the CAmkES (component architectures for microkernel-based embedded systems) project is to provide an architecture for building such componentised microkernel-based systems.

With an appropriate granularity of components, a componentised design gives developers control over which functionality is included in the system, helping to keep the TCB small and manageable. Furthermore, by placing the components in separate protection domains, fault containment is provided, supported by the underlying hardware memory management unit and managed by the microkernel. Given proper runtime support, a structured design based on well-defined components also enables the system architecture to be changed at runtime, allowing the loading and unloading of components as necessary. This leads to possibilities such as hot-swapping of service implementations, dynamic update of OS and system services, and even detection and restarting of faulty services (including drivers).

Verification of a complete software system's correctness will also benefit from a well structured design. This takes advantage of the fact that components are isolated and interact using only the mechanisms provided by the microkernel. Since a verified kernel gives guarantees about interaction properties, we can reason about the system using models of the individual components and their explicit interaction patterns.

Such structuring does not always come for free, however, and there are tradeoffs involved in componentising a system. In particular, having many small components means that there will be more communication involved. While IPC in L4 has been made fast and efficient, too much communication will nevertheless have a performance impact. On the other hand, a design with many small components is good for safety, reliability and verification. Depending on system requirements, decisions must be made regarding these tradeoffs.

The CAmkES architecture [34] allows such componentised systems to be easily built. Besides providing for improved software engineering through component-based development (CBD) [56], the architecture is flexible and allows for the making of tradeoffs as discussed above. Furthermore, it has low overhead and allows software to take full advantage of the benefits that a finely-tuned high-performance microkernel provides. Finally being a framework for developing a wide range of embedded systems (including those with limited resources), it does not make a resulting system pay for features that it does not need or use.

Two properties of CAmkES are key to providing these features. The first is that connectors, like components, are first-class entities [54]. Rather than being implicit in the architecture, as they are in most component systems, connectors are explicitly defined and implemented by system developers. This means that new connectors may be added without modifying the core architecture. Thus, when designing a system, the developer specifies the components used, the connections between components, and the specific connectors used for these connections. In this way, the developer has full control over the mechanisms used for interaction between components, however, the concerns of component functionality and interaction remain cleanly separated. Not only does this provide flexibility of design, it also provides flexibility in making tradeoffs between performance and other desired properties.

For example, during the design and implementation of a network-access router based on CAmkES and L4 we took advantage of the flexibility of connectors to experiment with safety vs. performance tradeoffs. We implemented several connectors ranging from those that provided complete memory protection between components to no protection between components. These connectors were used to produce versions of the router that represented the different tradeoffs with regards to protection and performance. We also developed a version of the router that combined the different connectors internally. In this router, components of the network stack were connected with performance optimised connectors, while the stack itself was protected from application-level components using connectors that provided maximum protection.

The second key property is that the architecture is minimal and does not include superfluous features that negatively impact runtime resource consumption. While the core architecture only provides support for statically-defined and -deployed systems, it is designed to be extensible. Extra functionality is added in the form of extension components, which are similar to regular components. Extension components can be added and combined as required to achieve desired functionality. Currently we provide extensions that allow components and connectors to be dynamically created and

destroyed at runtime. Besides the benefit to modularity, this also benefits verification, since it can greatly reduce the number of system components that must be verified (i.e., only those that are actually used need to be verified).

The CAmkES project is nearing completion and currently provides the core architecture, dynamic extensions, and several demonstrator systems. A scheduled follow-on project will investigate advanced aspects of componentised (trustworthy) embedded systems including real-time and power management, correctness of components and their connections, and the application of further software engineering approaches, such as model driven development (MDD).

3.5 Commercialisation

In the last two years, the embedded-systems industry has started to show a serious interest in L4. For example, in November 2005 NICTA announced that wireless chipset maker Qualcomm was deploying L4 on their *Mobile Station Modem* chipsets, which are used in mobile phones and other wireless devices. The first mobile phones running on L4 reached end users in late 2006.

NICTA has a strong focus on practical research outcomes and commercialisation, so this interest was a welcome development. In order to maximise market opportunities, a company, called Open Kernel Labs, was spun out. The company is in a joint venture with NICTA, in order to ease the transfer of the technology resulting from the above research projects, and to ensure that the research outcomes are commercially relevant. Open Kernel Labs now develops, markets and supports its own brand of L4, called OKL4 (a derivative of L4Ka::Pistachio [35]).

4. RELATED WORK

There are current research and commercial activities related to all of the projects discussed above. In this section we review the most relevant related projects.

The seL4 design is similar to early hardware-based capability systems such as CAP [43] in its approach to physical memory management, later software-based capability systems such a KeyKOS and EROS [23,53] in the management of authority, and virtual machine monitors such as Xen [1] (and to some extent L4), in the explicit management of address spaces and page tables. The Coyotos [52] and the Dresden L4.sec [32] projects have similar aims to seL4.

Early work on theorem-proving based OS verification includes PSOS [44] and UCLA Secure Unix [60]. A lack of mature mechanised theorem proving technology meant that while designs could be formalised, full implementation proofs were not achieved. Later, KIT [3], part of the CLI stack, described verification of process isolation properties down to object code level, but for an idealised kernel with far simpler and less general abstractions than modern microkernels.

The VeriSoft project [17] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS that is inspired by, but not very close to, L4. Performance and industry adoption are not goals of the project. The VFiasco project [30] aims to verify an existing kernel (L4/Fiasco) directly by developing a formal semantics for the subset of C++ used to build it. The added flexibility of our kernel design methodology and the much simpler semantics of C compared to C++ will produce a fully verified kernel earlier, cheaper and with more assurance.

The Coyotos team [52] take a different approach of defining a new low-level implementation language (BitC) with precise formal semantics, and hope to subsequently verify properties of the kernel that they are building. Although with less emphasis on high-level verification, the Singularity project also uses a type-safe imperative language (C#), but with additional compiler extensions to allow programmers and system architects to specify low-level checkable properties of the code, for example IPC contracts [13].

A significant number of groups are conducting research on WCET analysis, with several producing commercial tool sets. The leading static-analysis approach is that of Heckmann and Ferdinand [25], while the leading measurement-based approach is that of Bernat [2]. The latter is a precursor of the work of the Potoroo project. The combination of static analysis and measurements to provide guarantees on the computed results has been attempted by Yamamoto [63]. Their approach differs from ours in the way that the measurements and static analysis are used. While we aim to measure the behaviour for the worst case and confirm sufficient measurement coverage with static analysis, their approach is to measure a best case scenario in terms of cache contents and add a penalty for cache misses based on the static analysis. Our approach accounts for additional execution time which might be produced by the secondary effects of a cache miss, such as additional pipeline bubbles.

WCET analysis has been applied to operating-system code by Colin and Puaut [7], who used their static-analysis toolset on the RTEMS kernel. Compared to L4, RTEMS is a much simpler system, as it does not support memory protection and multiple privilege levels. Furthermore, the analysis was performed using a very simple architecture, lacking caches and branch prediction. The results contained overestimations of, on average, 80% of the measured WCET. Potoroo aims to support modern high-end embedded processors, such as those based on the ARM architecture, and to reduce pessimism.

Existing component architectures based on enterprise component technologies such as .Net [41], EJB [55], and CORBA Component Model (CCM) [46], fail to address the critical issues (including resource constraints, real-time performance, reliability, and energy use) of using components in embedded systems. Research and engineering efforts that focus on establishing component-based software engineering disciplines targeted specifically at embedded systems can be roughly divided into three categories.

The first category includes component architectures that target specific application domains such as field devices, consumer electronics, vehicular systems, etc. Examples of such architectures include PECOS [18], Koala [59] and Save [22]. While the models in this category are suited to embedded systems they are generally too restrictive, and do not provide the flexibility required to make them more broadly applicable in the embedded systems domain.

The second category encompasses component-based operating systems such as TinyOS [28], Pebble [16] and Think [14]. In these systems the component model is applied at the operating system level, and the details of the OS and the component model are intertwined. Other projects such as Knit [48], Click [33] and MMLITE [26] provide component architectures for building system level software. The main difference with the CAmkES approach is that our model

is designed so that it can optimally make use of the given kernel mechanisms, rather than provide mechanisms of its own or be used to build such mechanisms.

The third category consists of middleware-based component models tailored for embedded and real-time systems with a particular focus on non-functional attributes. Examples of these include CIAO [62], COMQUAD [19] and PECT [61]. Typically the overhead of the middleware support, even when tailored for embedded systems, is quite high. While originally closely tied to Microsoft platforms, the OpenCOM [8] project advocates an approach that more closely resembles ours in terms of broader applicability, as well as minimality and extensibility.

5. CONCLUSIONS

We have presented NICTA's research program for trustworthy embedded systems, based on exploiting the inherent strengths of microkernel technology, especially the small yet general-purpose L4 kernel. While there are groups working on projects similar to ours, to date we have not identified any other group with such a comprehensive and ambitious agenda, certainly not in such an advanced state.

The present set of research projects is to conclude within about a year, at which time the technology will be handed over to our spinoff Open Kernel Labs for commercialisation. Various results will make their way into products over the next two years. We believe that in 2009 it will be possible to deploy real embedded systems products that provably satisfy the highest possible levels of safety, security and trustworthiness.

6. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on OS Principles*, pages 164–177, Bolton Landing, NY, USA, Oct. 2003.

[2] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real–time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 3–5 2002.

[3] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[4] P. Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

[5] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, Dec. 1993.

[6] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on OS Principles*, pages 73–88, Lake Louise, Alta, Canada, Oct. 2001.

[7] A. Colin and I. Puaut. Worst case execution time analysis of the RTEMS real-time operating system. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, Netherlands, June 13–15 2001.

[8] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, Nov. 2004.

[9] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, Sept. 2006.

[10] D. Elkaduwe, P. Derrin, and K. Elphinstone. A memory allocation model for an embedded microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pages 28–34, Sydney, Australia, Jan. 2007. NICTA.

[11] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[12] K. Elphinstone, G. Klein, and R. Kolanski. Formalising a high-performance microkernel. In R. Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, Aug. 2006.

[13] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. of EuroSys2006*, April 2006.

[14] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.

[15] B. D. Fleisch, M. A. A. Co, and C. Tan. Workplace microkernel and OS: A case study. *Software: Practice and Experience*, 28:569–591, 1998.

[16] E. Gabber, C. Small, J. L. Bruno, J. C. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the USENIX Annual Technical Conference, General Track*, Monterey, CA, USA, June 1999.

[17] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, pages 1–16, Oxford, UK, 2005.

[18] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the PECOS approach. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*, Grenoble, France, Oct. 2002.

[19] S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, Lancaster, UK, Mar. 2004. ACM Press.

[20] R. P. Goldberg. Architecture of virtual machines. In *AFIPS*, pages 74–112, New York, June 1973.

[21] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the 1990 Summer USENIX Technical Conference*, June 1990.

[22] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. SaveCCM - a component model for safety-critical real-time systems. In *Proceedings of the 30th EUROMICRO Conference (EUROMICRO '04)*, Rennes, France, Sept. 2004.

[23] N. Hardy. KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, Oct. 1985.

[24] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, Oct. 1997.

[25] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of Design, Automation and Test in Europe, DATE 2005*, Mar. 2005.

[26] J. Helander and A. Forin. MMLite: A highly componentized system architecture. In *Proceedings of the 8th SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[27] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 113–126, Seattle, WA, USA, Apr. 1992.

[28] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, UK, Nov. 2000.

[29] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[30] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, UK, Oct. 2005.

[31] T. John and R. Baumgartl. Exact cache characterization by experimental parameter extraction. In *Proceedings of the 15th International Conference on Real-Time and Network Systems RTNS07*, pages 65–74, Nancy, France, Mar. 2007.

[32] B. Kauer and M. Völp. *L4.sec: Preliminary Microkernel Reference Manual*. Dresden University of Technology, http://os.inf.tu-dresden.de/L4/L4.Sec/, Oct. 2005. Last visited 2007-04-10.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[34] I. Kuz, Y. Liu, I. Gorton, and G. 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.

[35] L4Ka Team. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/.

[36] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520, National ICT Australia, Oct. 2005.

[37] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, USA, Dec. 2004.

[38] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in HYDRA. In *ACM Symposium on OS Principles*, pages 132–140, 1975.

[39] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–188, Asheville, NC, USA, Dec. 1993.

[40] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, Dec. 1995.

[41] J. Löwy. *Programming .NET Components*. O'Reilly & Associates, Inc, 2003.

[42] Y. K. Malaiya and J. Denton. Estimating defect density using test coverage. Technical Report 98-104, Colorado State University, 1998.

[43] R. Needham and R. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on OS Principles*, pages 1–10. ACM, Nov. 1977.

[44] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[45] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[46] Object Management Group. *CORBA Component Model*, 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-01.

[47] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[48] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[49] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[50] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. Maintainability of the Linux kernel. *IEE Proceedings: Software*, 149:18–23, 2002.

[51] S. Schaefer, B. Scholz, S. M. Petters, and G. Heiser. Static analysis support for measurement-based WCET analysis. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Work-in-Progress Session*, Sydney, Australia, Aug. 2006.

[52] J. Shapiro. Coyotos. www.coyotos.org, 2006.

[53] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on OS Principles*, pages 170–185, Charleston, SC, USA, Dec. 1999.

[54] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE '93: Selected papers from the Workshop on Studies of Software Design*, pages 17–32, London, UK, 1996. Springer-Verlag.

[55] Sun Microsystems. *Enterprise JavaBeans Specification Version 3.0*. http://java.sun.com/products/ejb/index.jsp.

[56] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.

[57] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[58] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, Jan. 2007.

[59] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, Mar. 2000.

[60] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[61] K. C. Wallnau. Volume III: A technology for predictable assembly from certifiable components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2003.

[62] N. Wang, D. C. Schmidt, M. Kircher, , and K. Parameswaran. Adaptive and reflective middleware for QoS-enabled CCM applications. *IEEE Distributed Systems Online*, 2(5), 2001.

[63] K. Yamamoto, Y. Ishikawa, and T. Matsui. Portable execution time analysis method. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing and Applications*, Sydney, Australia, Aug. 2006.