Chapter 1
Trustworthy Real-Time Systems

Stefan M. Petters, Kevin Elphinstone and Gernot Heiser

Abstract Embedded systems have experienced a dramatic increase in ubiquity and functionality. They have penetrated our life to a degree where we rely heavily on them and at the same time entrust them with vast amounts of personal information. The trust placed in them does not necessarily mean they are trustworthy. Within this chapter we summarise past research of the ERTOS group at NICTA in the area and provide the initial reasoning which motivated that research. Topics covered are a secure kernel design and the design and verification of this kernel design, as well as work on scheduling and WCET analysis.

1.1 Introduction

The market of embedded processors far surpasses the market of personal computers and servers. While being more prolific than their desktop counterparts, the progress in semiconductor technology has also brought unprecedented computing power to embedded systems. On the back of these opportunities the complexity of embedded applications is rising dramatically. Two typical examples are today's smartphones or cars. The amount of software contained in these devices is impressive, as for example 100 million lines of code (LOC) in a modern high end car [7] in 2009, while the Android operating system without applications weighs in at around 12 million LOC in 2010.

With the rising complexity there is also increased scope for problems leading to system failure. While it is clear that a degree of functional correctness is required,

Stefan M. Petters
CISTER/ISEP, Polytechnic Institute of Porto, Portugal e-mail: smp@isep.ipp.pt

Kevin Elphinstone
NICTA and UNSW, Sydney, Australia e-mail: kevine@cse.unsw.edu.au

Gernot Heiser
NICTA, UNSW and Open Kernel Labs, Sydney, Australia e-mail: gernot@nicta.com.au

temporal behaviour must also be considered even in the absence of safety-critical temporal requirements, as the utility of most systems is subject to degradation in the presence of temporal failures. In this paper we consider a *bug* to be any kind of design or implementation error. Aside from bugs, system failures may also be triggered by deliberate attacks or transient errors. Examples of the latter are electromagnetic interference or single-event upsets induced by ionised particles in space.

Another trend in embedded systems is towards mutable or open systems [14]. This ranges from upgrade facilities available in factory machinery, to tunable cars, to downloadable extensions and games on personal digital assistants or mobile phones. This post-deployment installation of applications not only increases the attack surface for a device, it also implies that the set of active tasks cannot be determined a-priori, which requires a dynamic approach to managing processor time. Furthermore, not the entire task set of a system will have hard real-time character, but some will be of best-effort or soft–real-time character. Such systems are usually referred to as mixed-criticality systems and require that the system components having real-time requirements be protected from those which have a best-effort character.

In summary we perceive the following requirements for modern embedded systems. Reliability and security are the central pillars of the increasing number of complex embedded systems we surround ourselves with. This covers not only the functional aspects, but also the temporal aspects of these systems. While not all requirement lapses will lead to catastrophic consequences, issues like confidentiality of personal and commercially sensitive data, as well as simple usability aspects, motivate the construction of trustworthy systems beyond the traditional safety domain. The lowest software layer or core needs to provide the basic mechanisms for allowing secure, reliable and feature rich embedded systems to be built. While the availability of such mechanisms alone is not sufficient to prevent unreliable systems to be developed, it is a precondition for reliable systems.

The following sections will provide an overview of past research at the ERTOS group at NICTA and the considerations which lead to this research agenda.

1.2 Comparison of Approaches

1.2.1 Real-Time Executives and Desktop Operating Systems

Currently two operating system (OS) paradigms dominate the embedded systems market: classical real-time executives and (more recently) stripped-down versions of desktop operating systems. The major advantage of real-time executives is their size. They fit on small devices and consume minimal resources. A real-time executive can usually be bypassed, as it is rarely used with memory protection enabled, even if that is available on the given platform. This leaves the executive unprotected, and a bug in an application may crash the entire system. Sometimes state is corrupted slowly, and the offending instructions can be very hard to track down. As a consequence,

real-time executives are mostly deployed on classical single-purpose devices of low to moderate complexity.

Stripped-down desktop operating systems have been introduced in embedded systems some time ago, but have become more mainstream to handle more complex embedded applications as well as harness the computational power available. This is driven by the realisation that embedded devices often take on tasks similar to a desktop computer. Examples of such stripped-down operating systems are embedded versions of Windows or Linux (such as Android). A major advantage of such operating systems is that – unlike real-time executives – the kernel is to some degree protected against misbehaving applications.

However, even a minimal configuration of Linux or Windows still features a large code base, hundreds of thousands of lines, and dubious real-time performance. While there are attempts to address real-time behaviour, such as the preempt-RT [30] effort in Linux, these are highly dependent on (for Linux kernel code) unusual coding standards, such as not disabling interrupts, which can be easily violated for example by legacy drivers.

1.2.2 Trusted vs. Trustworthy

While many embedded systems are trusted to some degree to perform according to their specification, it may be questioned whether these systems deserve the trust placed in them. Recalls of cars for *software updates* or required reboots of mobile phones indicate that there are indeed issues which need to be addressed.

Trustworthiness can be established in several ways, collectively referred to as *verification*. Firstly functional correctness can be assured by performing exhaustive testing. While this is possible for trivial examples, the approach fails for more demanding code. Systematic code inspection helps to increase the confidence that the code performs as expected, but as complexity increases, this also fails to provide real assurance, as it is in general not practical to consider all interactions between different parts of a complex system. The logical extension of systematic code inspection is employing *formal verification* methods. Formal verification provides a mathematical proof of correctness, but becomes infeasible beyond a few thousand lines of code.

The poor scalability of thorough verification approaches requires a system that is "small" (of limited conceptual complexity). A small system is not only easier to verify, it is less likely to be faulty (or vulnerable) in the first place. However, smallness is not necessarily required for the whole system, only the "critical" bits. This is encapsulated in the concept of the *trusted computing base* (TCB) [28]. The TCB is the collection of components and mechanisms in hardware, firmware, and software that are critical to enforce the security policy of an entire computer system. A bug in the TCB has the potential to undermine the security or safety of the whole system.

The TCB contains, among others, all software executing in the hardware's privileged mode, this is generally referred to as the *operating system kernel*. In the case of traditional microcontrollers that do not offer memory protection hardware, or when employing a real-time executive which does not utilise memory protection, *all software* is part of the TCB. In such a system of non-trivial size, security or safety become practically impossible to assure.

1.2.3 Minimising and Managing the TCB

To enable any meaningful verification, the size of the TCB needs to be minimised. This requires minimising the OS kernel. The most practical way to minimize kernel size is to restrict it to provide just basic mechanisms for securely managing the hardware, and move all actual system services (including the implementation of application-specific policies) into user-mode code. Such a minimised OS kernel is called a *microkernel* [20].

Some non-kernel components of such a microkernel-based system are still part of the TCB, despite executing without hardware privileges. However, they are now themselves subject to kernel-mediated memory protection and forced to interact via well-defined interfaces. This greatly aids their verification, as they can often verified independently of the rest of the system. The kernel itself can be made small enough that it is within reach of the most rigorous verification approaches. The main prerequisites for using this approach is that the hardware provides memory protection (in the form of a memory-management or memory-protection unit) and dual-mode execution (the distinction between privileged and unprivileged modes).

Figure 1.1 provides a comparison of the three different approaches to system structure. In systems built on top of traditional real-time executive, all code is part of the TCB (indicated by the grey background) and as such are safety/security-critical. A (stripped-down) desktop OS removes application code from the TCB, as the OS is protected by hardware mechanisms. However, the OS itself is still large (100,000s of lines of code) and infeasible to verify completely. In the microkernel-based system, the TCB consists of the (much smaller) kernel plus whatever essential services the critical system operations depend on, the total can be as small as 10,000 lines of code.

While a microkernel-based OS does not ensure that a system is well-designed, it provides the basic mechanisms for building robust systems, by encapsulating services into individual address spaces [15]. In general, the microkernel provides a good base for componentised systems, as envisaged by, for example, the Autosar consortium. Finally it simplifies reasoning about fault isolation properties of the system, and lends support for advanced features such as hot swapping and hot upgrades.

Device drivers are an interesting case in point. Traditionally they are in the kernel and therefore part of the TCB. In a microkernel-based system, they are user-mode components and might be removed from the TCB. There are two reasons why even

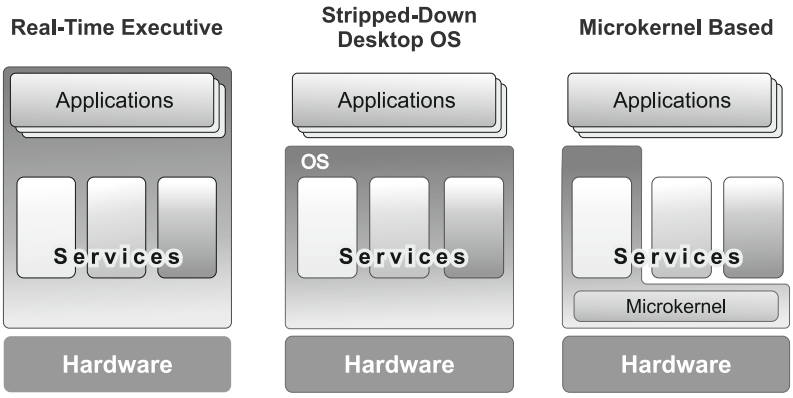

Fig. 1.1 Comparison of Approaches

user-mode drivers may be part of the TCB: Firstly, the correct operation of the system may depend on the correct operation of the device (e.g. a safety-critical actuator). Secondly, if the device is capable of direct memory access (DMA), it could overwrite arbitrary memory, including the microkernel. The latter problem can be avoided if the hardware platform features an *input/output memory management unit* (IOMMU), which makes DMA subject to memory protection just as any other memory access.

1.3 Design and Verification

When setting out to create a trustworthy operating system, a basic question is which design approach to use. Traditionally two different approaches, representative of different mindsets, have been pursued. In a bottom-up approach, depicted in Figure 1.2, expert programmers design and implement an operating system, which will later be subject to verification by formal-methods experts. Given the expensive nature of formal verification, it is performed as a last step after the system is mature and thoroughly tested to ensure acceptable performance. Since expert kernel pro-

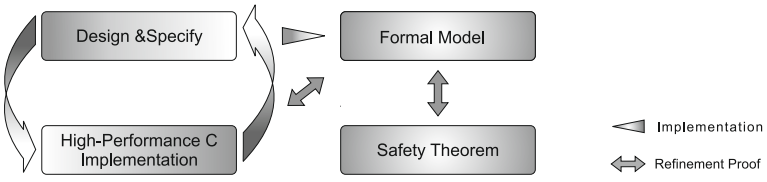

Fig. 1.2 Expert Programmer Design Flow

grammers are not normally formal-verification experts, this likely leads to a system which in the best case is not very amenable to proving the required properties of the system, and in the worst case does not even expose the properties stipulated in the initial requirements, or be practically intractable to formally reason about due to complexity.

An alternative approach usually favoured by formal methods experts is to start with a formal specification and high-level design to ensure the desired properties will be achieved [16]. The design is then implemented by kernel programmers, this approach is illustrated in Figure 1.3. While ensuring that the implementation does not impinge on the properties proved for the design is reasonably straight forward, the design decisions taken on such high abstraction levels likely result in an operating system which might be correct, but exhibits unacceptable performance, or is impractical for programmers build applications upon.

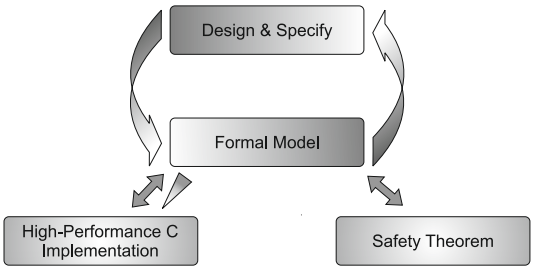

Fig. 1.3 Formal Methods Centered Design Flow

A third way, employed by NICTA, is to assemble a team of developers with both formal methods and programming experts willing to at least partially learn the repective expertise of their counterparts, and work together to achieve a provably trustworthy high-performance operating system. However, development of expertise alone is not enough, as one must reconcile the different development approaches of both groups. Additionally, the requirement of an iterative approach to the design still exists. This has two implications. On the one hand it requires a quick assessment of the performance impact of design decisions, on the other hand it requires a low barrier to formally reasoning about the design, in order to ensure optimisations aimed at improving performance still maintain the required formal properties.

The team at NICTA has approached this problem by choosing Haskell as the specification and design language [13]. Its functional style makes it amenable to automatic translation into the input of an interactive theorem prover. It also forms part of an executable specification which may be used to assess the performance impact of design decisions by requiring a concrete implementation of data structures and algorithms, and enables a much faster turn-around of the design of the API, compared to implementing any specification change in a low-level programming language like C.

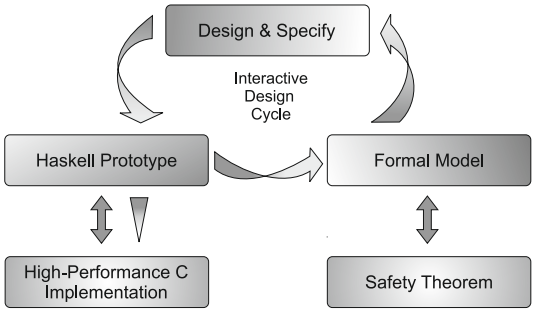

Fig. 1.4 Iterative Design Approach

With an executable specification, an operating system in-the-loop can be used to assess the utility, usability and general performance of API design decisions without resorting to the time-consuming implementation associated with bare hardware prototypes. Figure 1.5 illustrates the approach taken at NICTA. A set of test applications are be ported to the operating system. The applications themselves execute on a simulator, which is modified such that it only directly simulates user-mode (i.e. application) execution. Transitions to kernel mode (via system calls, interrupts, and other exceptions) result in transfer of control to the Haskell model, which replaces the kernel-mode portion of the simulator itself. The Haskell model manipulates the user-level state such that to applications it appears that an operating system running in kernel mode had serviced the exception.

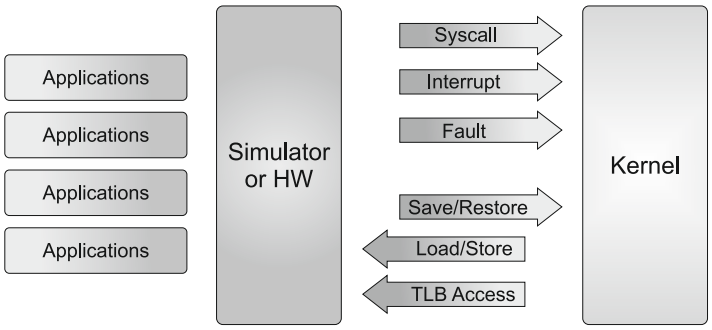

Fig. 1.5 Kernel In the Loop [11]

The approach of running the specification also avoids ambiguities of an plain English specification. Haskell is a functional programming language with well-defined semantics, which enables automatic translation into an input usable by theorem provers. This then provides the starting point for the verification effort. This is eased by the fact that the theorem prover Isabelle/HOL [24] is based on lambda calculus, which also forms the core of Haskell.

Some parts of the kernel, such as the exact scheduling algorithm, are left underspecified in the NICTA work, to allow a later exploration of design choices. Difficulties in the verification process can be addressed, if needed, by changes to the kernel design or API. This flexibility proved to be of fundamental importance in reducing the time to proceed through design iterations.

Despite the common roots of Haskell and Isabelle/HOL, some restrictions were required to facilitate the automatic translation. The executable model avoided the use of lazy evaluation, unlimited recursion, and other problematic Haskell features. Additionally, to support the later reimplementation of the production design in C, the model also avoided use of garbage collection beyond automatic stack variables. The interested reader is referred to relevant publications for further details [17, 18].

1.4 Security and Safety

Security comprises the aspects of confidentiality, integrity and availability; in a particular application scenario some of these tend to be more important than others. Safety is similar, except that confidentiality is not relevant but the (data) integrity aspect is extended to functional correctness.

In many cases of embedded systems, security and safety requirements can be fulfilled using isolation of components. Isolation can be used to prevent a misbehaving application from bringing down another application, or prevent a malicious application from intruding on private data or denying critical actions.

With respect to security and safety requirements, isolation relates to the spatial as well as temporal domains. In the spatial domain it leads to address-space isolation as the basic mechanism. Temporal isolation as it relates to scheduling is discussed in Section 1.5. In this section examine temporal isolation in the context of avoidance of denial-of-service attacks through carefully crafted system calls to, for example, cause a very long running kernel operation and thus deny the other applications of the service of the operating system.

Memory management needs to be carefully considered to avoid starvation of this resource at runtime and consequent unbounded or long running system calls. One challenge is that resources such as pages and threads not only require the memory that is part of the user-visible resource, but additional memory for the in-kernel metadata required for management of the resource. Implicit allocation of this memory creates a potential exploit, by forcing the kernel to allocate large amounts of memory for the metadata, which leads to the starvation of memory needed for other tasks. One way to address the starvation issue is to avoid dynamic memory allocation in kernel. In order to achieve this in the seL4 microkernel [12], seL4's design team promoted all metadata to explicitly allocated first-class objects. The metadata is included in the object allocation process and is part of the memory for the object, thus enabling direct control of memory consumption.

A further building block in providing spatial isolation is managing the authority of object creation and authority transfer between applications. The seL4 kernel is

implemented as a capability-based system. Capabilities [10] are a tamper-proof representation of the authority to access and/or modify certain data and invoke kernel services. A capability, and thus the corresponding authority, can be passed from application to application allowing concepts like shared memory or task creation to be implemented.

During bootstrapping, all free memory not used by the kernel is handed to the first task as *untyped memory* (UM). Such UM may be subdivided, passed to another application or cast into some other type (e.g. a thread-control block) via a *retype* invocation. The capability, and thus authority to perform these operations, comes with the UM. After invoking the retype method and creating an object of different type, the invokee receives full authority over the object. When it passes the capability to another task it may do so with reduced authority. The key observations to note are that all kernel data structures are objects of a specific type; all typed objects are allocated (via retype) using authority to UM, and UM is a finite resource. Hence applications cannot exceed the memory footprint beyond the authority they possess.

A typical system architecture based on the seL4 microkernel is depicted in Figure 1.6. The system is comprised of a sensitive application which needs to be protected from unknown or untrusted convenience and legacy functions also running on the same hardware. This might, for example, separate the security-critical crypto-functionality and processing of unencrypted data from wireless networking functionality, or the life-supporting functions in a medical device from the GUI stack. Spatial isolation is assured between all parts of the system. However, trusted services and drivers can be accessed from the untrusted subsystem through well-defined and enforced interfaces. A potential use of a Linux server allows legacy or comfort

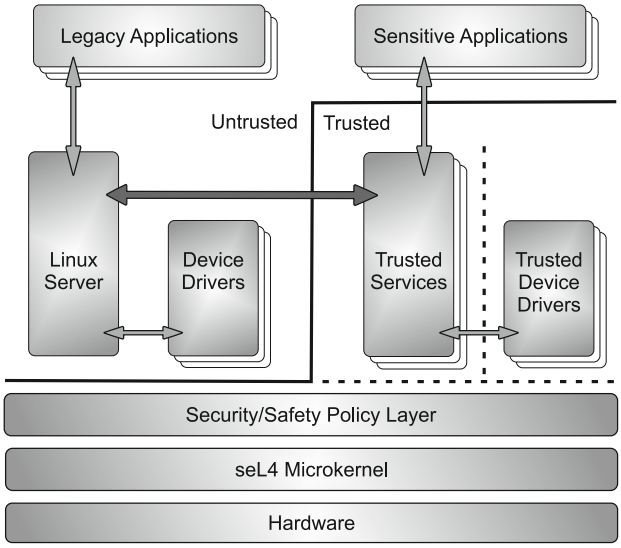

Fig. 1.6 seL4-based system architecture

Linux applications to be deployed alongside critical applications without porting to a new environment. Further information on seL4's security mechanisms can be found in the relevant publications [12, 17].

1.5 Temporal Isolation

The functional isolation and its verification discussed so far is instrumental to building trustworthy embedded systems. This needs to be augmented by temporal isolation, which we now look at in detail in the context of resource scheduling.

Most embedded operating systems in industrial use are either time-driven or fixed-priority–driven. The time-driven approach provides isolation of different applications at the cost of responsiveness. For example, OSEKTime [27] implements a table-driven scheduling approach that allows interrupt delivery only in one slot of the schedule, and otherwise requires interrupts to be disabled. Ultimately this leads to a potential latency of interrupt delivery of an entire scheduling frame. In contrast, fixed-priority schedulers offer the flexibility of fast response time for high-priority applications. However, fixed priority scheduling suffers from poor temporal isolation, as there is no inherent contract about computation time available to an application. Furthermore, relative urgency as well as importance of the application have to be integrated into a single priority value.

For traditional tightly-integrated systems with a known task set, these limitations were acceptable. However, the increasing integration of applications of different and potentially unknown vendors, as well as legacy subsystems, into one device highlights the issue of lack of temporal isolation.

Dependent on the circumstances temporal fault isolation may have different goals. Obviously it should allow critical components to continue to operate correctly in the presence of misbehaving component. It may also ease fault location by avoiding knock-on effects which hide the source of a problem.

Another aspect of temporal isolation is that subsystem may have different scheduling requirements. Best-effort type applications are traditionally scheduled using a fairness-based scheduling scheme, while real-time systems require strict guarantees and the implementation of which are often in juxtaposition to fairness. However, modern embedded systems have often both types of scheduling needs, leading to mixed-criticality systems with conflicting requirements [22].

A number of scheduling approaches have been proposed to address these issues. Examples include sporadic servers [31], deferrable servers [32], or constant-bandwidth servers [1]. In our discussion we will focus on the constant-bandwidth server. Rather than assigning tasks priorities or specific time slots, tasks are assigned a budget of execution time, which will be replenished periodically. This effectively assigns tasks a certain share of the execution time. While the approach guarantees the execution time budget to be available before the next replenishment, it does not stipulate in which order unused budgets may be consumed, allowing a secondary scheduling algorithm to be applied.

Depending on the criticality and the worst-case execution time (WCET) of the task, different sizes of the budget can be chosen. For hard real-time applications, budgets will essentially be dictated by the WCET of a task. For soft real-time applications, budgets might be chosen to be less than the WCET but higher than the average execution time. In this case, a slack-management approach, as described below, can use leftover budget to fulfill the needs of a task in most cases where the execution time exceeds the assigned budget. Finally, best-effort tasks with a long execution time and aperiodic inter-arrival may be assigned budgets and periods which are a fraction of the needs of an individual job of a task. Such time slicing in combination with slack management ensure fairness and progress among the multiple best-effort tasks.

Research on scheduling performed at NICTA is based on earlier work by Brandt et al. [6, 21]. A fundamental observation underlying their work is that neither a fairness-based approach retrofitted with some real-time mechanisms, nor a real-time scheduling approach with an emulated fairness layer will provide acceptable service for both classes of application needs. Instead an integrated solution offering native fairness and real-time guarantees is required.

In principle, Brandt's proposal has many similarities with other approaches based on bandwidth servers [2]. The distinguishing factors of this work are the detailed consideration of temporal reconfiguration constraints, integration of an albeit simple real-time analysis, and finally, effective dynamic slack management. Brandt's work uses *earliest deadline first (EDF)* [23] as the secondary scheduling algorithm. EDF has been proven to be optimal for a set of independent non-blocking tasks. Besides assuming independent deadlines of all tasks, deadlines were assumed to be equal to the release periods of tasks, which was also used as replenishment period.

In the work of Brandt et al., the traditional scheduling approach is split into a fundamental choice of how much bandwidth is allocated, and a dispatcher to decide when. The former step is taken by the resource allocator, which is active in any mode change of the system to recalculate available shares. The latter is performed by a more traditional scheduler. However, the scheduler enforces the budget allocation and manages unused budget, as well as overruns. Unused budget or slack can be collected and passed to tasks in need of extra budget.

Different aspects of this slack management are discussed by Lin and Brandt [21]. Slack can be passed around in the system under the presumption that slack is scheduled with either the deadline of the task it was associated with, or a more relaxed deadline. This property is required to ensure that the assumptions made in the resource allocator regarding budget use are maintained. A very fundamental corollary is the observation that a task exhausting its budget can be assigned the budget of the next release, if the deadline of the tasks is adjusted such that it reflects the deadline relative to this future release.

The core elements of this approach are depicted in Figure 1.7. Application arrival or departure in the system triggers a reconfiguration of the budget allocation. During this stage, a real-time analysis is performed, and, after successful completion, the task is provided with a budget and period and is added to the set of schedulable tasks. The EDF scheduler assigns budgets and thus associates applications with enforced

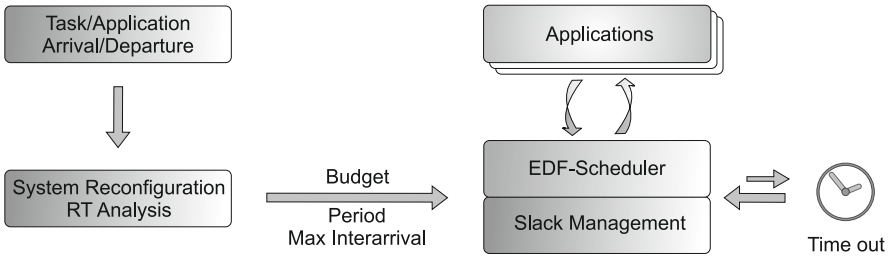

Fig. 1.7 Separation of Budget Allocation and Dispatch

timeouts. A slack manager integrated with the scheduler performs the collection and assignment of slack.

We have extended the approach of the resource allocator to perform a demand-bound–based schedulability test [3]. Fundamentally, these tests compute the worst-case request for computation in a given interval. If the worst-case demand does not exceed what can be provided in the same interval, the schedulability test is successful. An expressive event model, which describes the worst-case number of releases of a task – including release jitter and bursty releases – underpins this analysis.

Besides adding the capability to analyse bursty tasks, the resource allocator may also allow for deadlines which are different from the minimum inter-arrival times. This is relevant for rare, but highly important tasks, such as tasks responding to exceptional external input. We have also investigated this approach for critical sections, such as interrupt-service routines and critical sections in user-mode drivers [26]. Further work exploits the explicit knowledge of slack to facilitate practical dynamic voltage and frequency scaling in a real-time environment [19].

1.6 Worst-Case Execution-Time Analysis

The knowledge of the WCET is a fundamental building block for reliable real-time systems. The resource allocator requires such values to perform an overall schedulability analysis. The analysis of the WCET needs to be performed on all parts of the system involved in the delivery of a timely service for soft and hard real-time systems. This obviously covers the real-time applications, but also the kernel and services involved in the process.

The fundamental steps of WCET analysis are depicted in Figure 1.8. The code for which the analysis is to be performed is firstly analysed for its control-flow structure and operations performed along the nodes of that control-flow graph. In a second step, the constraint and flow analysis aims at restricting the set of possible flows through the program to those which are feasible. This is, for example, necessary to provide bounds on the number of iterations of a loop or to identify nodes in the control-flow graph which may be structurally executable, but are proven to be mutually exclusive due to data dependencies.

In a third step, the execution time of individual nodes in the control-flow graph are established. This may take into account restrictions developed in the flow analysis, as for example, the identified mutual exclusivity of nodes may imply that certain hardware states are not possible when one of the nodes is entered, which may avoid otherwise possible execution paths. Finally, the low-level timing results are combined using the constraints provided by the flow analysis, e.g. to bound the number of iterations for a given loop.

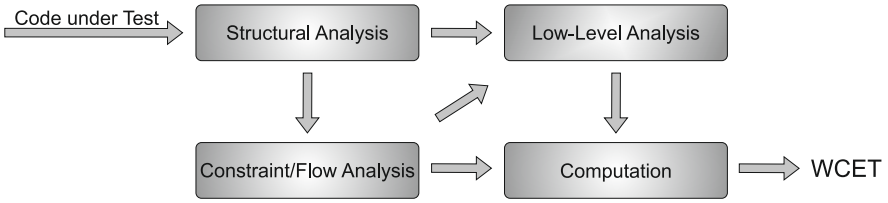

Fig. 1.8 Principal Steps of WCET analysis

Academic research on WCET is mostly based on static analysis, which lends itself well to the approach shown in Figure 1.8. In contrast, analysis of safety-critical systems in industry is largely performed using end-to-end measurements augmented with safety margins. The latter approach works well for simple architectures such as those employed by most 8- and 16-bit microcontrollers. It fails for more complex processors, such as the widely-deployed ARM architecture, which features instruction pipelining, branch prediction and multiple levels of cache.

Static WCET analysis on its own has significant limitations too. For one, it is dependent on detailed hardware models, which are not available especially for the more complex architectures. The complexities of modern architectures also mean that the latency of many instructions is highly dependent on internal processor state, which is frequently infeasible to model completely, forcing static analysis to adopt a pessimistic view which leads to gross over-estimation of WCET, often by orders of magnitude.

Fortunately, debugging interfaces available for modern processors can support fine-grained execution tracing and latency measurement. This enables an approach that measures the execution time of small sections of code and combines the results for these to obtain an estimate of the WCET for the overall program using techniques borrowed from static analysis. This approach works under two assumptions: Firstly, the variation in execution time for a small section of code without branches (i.e. a basic block) is small and able to be determined by measurements, even if these are not exhaustive. Secondly, the dominant variation of execution times is caused by the variability of executed paths through the program, which is where static analysis helps [4, 8]. For high-end processors, cache analysis is required to augment the measurements at the basic-block level.

In order to further refine the WCET analysis, it was performed not only on the maximum execution time observed, but also on the distribution and thus providing

the frequency of observed execution times. Such execution-time profiles [25] can be interpreted as probability distributions and subsequently used in performing a probabilistic analysis [5].

Compared to application code, WCET analysis of an operating-system kernel presents extra challenges [9, 29], specifically in the structural and flow analysis depicted in Figure 1.8. Kernel code is highly (manually) optimised, and typically includes parts written in assembly language for efficiency. The combination of optimisation and assembly code means that the complexity of the structural analysis, which is one of the fundamental steps in WCET analysis, becomes extremely difficult to automate. Furthermore, OS code presents the challenge that a single point of entry – an exception-handler address – vectors to many different code paths, such as interrupt-service delivery, exception handling and a variety of system calls. This affects structural as well as flow analysis.

Dealing with the challenges especially of system-call vectors would require a substantial investment into the automatic translation of kernel code into a control-flow graph. However, such a tool would likely be even more complex than a complete hardware simulator, as it would need to consider a very large state space of possible input values (in architected registers). Such a complex tool would be almost impossible to get right, which undermines the point of determining WCET as part of the for safety analysis of a system.

The (for now) preferred alternative is to rely heavily on programmer annotations. While this scales poorly, it is acceptable in the case of a microkernel, owing to its small size, relatively slow evolution (especially in the case of a formally-verified kernel) and the fact that the structural and constraint analysis needs to be done only once. In the case of seL4, there is additional benefit from the many invariants which have been proved in the course of the formal verification. These can partially replace manual annotations, and the existing verification framework makes it relatively easy to add (and prove!) further invariants. In essence, the functional verification provides (very precise) knowledge about properties of the code way beyond what is normally available.

1.7 Conclusions

The rising complexity of embedded systems creates challenges in achieving trustworthiness in security- or safety-critical deployments. We have provided on overview of some of these challenges, and indicated ways of addressing them. Recent work at NICTA is a substantial step towards overcoming these challenges, with contributions in the areas of proofs of functional correctness of a microkernel, isolation in resource management and scheduling, and analysis of worst-case execution times.

While our aim has been to create a trustworthy foundation for embedded software systems in the form of a formally-verified microkernel, this is only the first step towards trustworthy systems. Given a precisely-specified, -behaved, and -analysable operating system, the challenge becomes how to lift the guarantees of the founda-

tion to the level of whole systems. This includes systems composed of software of varying degrees of trustworthiness that rely on the operating system to protect safety- or security-critical components from malfunctions introduced by other components, which may make up the vast majority of the overall code base. Our ongoing research aims at developing frameworks, methodologies, and tools to compose demonstrably trustworthy embedded systems from components of varying criticality and trustworthiness.

Acknowledgements 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.

This work was also supported by the Portuguese Fundação para a Ciência e a Tecnologia (CISTER Research Unit - FCT UI 608).

References

1. Abeni, L., Buttazzo, G.: Integrating multimedia applications in hard real-time systems. In: Proceedings of the 19th IEEE Real-Time Systems Symposium, pp. 4–13. IEEE Computer Science Press, Madrid, Spain (1998)
2. Abeni, L., Lipari, G., Buttazzo, G.: Constant bandwidth vs. proportional share resource allocation. In: Proceedings of the 5th IEEE International Conference on Multimedia Computing and Systems, vol. 2, pp. 107–111. IEEE Computer Science Press, Florence, Italy (1999)
3. Albers, K., Slomka, F.: An event stream driven approximation for the analysis of real-time systems. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems. IEEE Computer Science Press, Catania, Italy (2004)
4. Bernat, G., Colin, A., Petters, S.M.: WCET analysis of probabilistic hard real–time systems. In: Proceedings of the 24th IEEE Real-Time Systems Symposium, pp. 279–288. Austin, Texas, USA (2002)
5. Bernat, G., Newby, M., Burns, A.: Probabilistic timing analysis: An approach using copulas. Journal of Embedded Computing **1**(2), 179–194 (2005)
6. Brandt, S.A., Banachowski, S., Lin, C., Bisson, T.: Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In: Proceedings of the 24th IEEE Real-Time Systems Symposium. Cancun, Mexico (2003)
7. Charette, R.N.: This car runs on code. IEEE Spectrum **46**(2) (2009). http://www.spectrum.ieee.org/feb09/7649
8. Colin, A., Petters, S.M.: Experimental evaluation of code properties for WCET analysis. In: Proceedings of the 24th IEEE International Real-Time Systems Symposium. Cancun, Mexico (2003)
9. Colin, A., Puaut, I.: Worst case execution time analysis of the RTEMS real-time operating system. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems, pp. 191–198. Delft, Netherlands (2001)
10. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Communications of the ACM **9**, 143–155 (1966)
11. Derrin, P., Elphinstone, K., Klein, G., Cock, D., Chakravarty, M.M.T.: Running the manual: An approach to high-assurance microkernel development. In: Proceedings of the ACM SIGPLAN Haskell Workshop. Portland, OR, USA (2006)
12. Elkaduwe, D., Derrin, P., Elphinstone, K.: Kernel design for isolation and assurance of physical memory. In: 1st Workshop on Isolation and Integration in Embedded Systems, pp. 35–40. ACM SIGOPS, Glasgow, UK (2008)

13. Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., Heiser, G.: Towards a practical, verified kernel. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems, pp. 117–122. San Diego, CA, USA (2007)
14. Heiser, G.: Hypervisors for consumer electronics. In: Proceedings of the 6th IEEE Consumer Communications and Networking Conference, pp. 1–5. Las Vegas, NV, USA (2009)
15. Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: MINIX 3: A highly reliable, self-repairing operating system. ACM Operating Systems Review **40**(3), 80–89 (2006)
16. Klein, G.: Operating system verification — an overview. Sādhanā **34**(1), 27–69 (2009)
17. Klein, G., Derrin, P., Elphinstone, K.: Experience report: seL4 — formally verifying a high-performance microkernel. In: Proceedings of the 14th International Conference on Functional Programming, pp. 91–96. ACM, Edinburgh, UK (2009)
18. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, pp. 207–220. ACM, Big Sky, MT, USA (2009)
19. Lawitzky, M.P., Snowdon, D.C., Petters, S.M.: Integrating real time and power management in a real system. In: Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications. Prague, Czech Republic (2008)
20. Liedtke, J.: On μ-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 237–250. Copper Mountain, CO, USA (1995)
21. Lin, C., Brandt, S.A.: Improving soft real-time performance through better slack management. In: Proceedings of the 26th IEEE Real-Time Systems Symposium. Miami, FL, USA (2005)
22. Lin, C., Kaldewey, T., Povzner, A., Brandt, S.A.: Diverse soft real-time processing in an integrated system. In: Proceedings of the 27th IEEE Real-Time Systems Symposium. IEEE Computer Science Press, Rio de Janeiro, Brazil (2006)
23. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM **20**, 46–61 (1973)
24. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer (2002)
25. Petters, S.M.: Execution-time profiles. Tech. rep., NICTA, Sydney, Australia (2007)
26. Petters, S.M., Lawitzky, M., Heffernan, R., Elphinstone, K.: Towards real multi-criticality scheduling. In: Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pp. 155–164. Beijing, China (2009)
27. Poledna, S., et al.: OSEKTime: a dependable real-time, fault-tolerant operating system and communication layer as an enabling technology for by-wire applications. In: SAE 2000 World Congress, pp. 51–70. Detroit, MI, USA (2000)
28. Rushby, J.: A trusted computing base for embedded systems. In: Proceedings of 7th DoD/NBS Computer Security Conference, pp. 294–311 (1984)
29. Singal, M., Petters, S.M.: Issues in analysing L4 for its WCET. In: Proceedings of the 1st International Workshop on Microkernels for Embedded Systems. NICTA, Sydney, Australia (2007)
30. Siro, A., Emde, C., Mc Guire, N.: Assessment of the realtime preemption patches (rt-preempt) and heir impact on the general purpose performance of the system. In: Proceedings of 9th Real-Time Linux Workshop. Linz, Austria (2007)
31. Stanovich, M., Baker, T.P., Wang, A.I., Harbour, M.G.: Diverse soft real-time processing in an integrated system. In: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE Computer Science Press, Stockholm, Sweden (2010)
32. Strosnider, J.K., Lehoczky, J.P., Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Transactions on Computers **44**(1), 179–194 (1995)