Issues in Analysing L4 for its WCET

Mohit Singal [†] Stefan M. Petters [‡◇]

[†] Computer Science and Engineering, IIT Guwahati Assam, India

[◇]National ICT Australia* Sydney, Australia

[‡] School of Computer Science and Engineering, UNSW Sydney, Australia

Abstract

Real-time analysis of a system requires knowledge of the worst-case execution time of all code in the system. This requirement covers not only application code, but also operating system and kernel code. In this paper we discuss the issues specific to kernel code and how we aim to address these in our work towards analysing the *L4* microkernel for the worst-case execution times of all system-call primitives. The main focus in that process is to maximise the degree of automation of the analysis, as the analysis needs to be repeated for any subsequent version of the kernel.

1 Introduction

Embedded real-time systems are becoming in general increasingly complex. While simple systems remain numerous, the number of more complex high end embedded systems is steadily growing and their complexity makes the use of a realtime operating system (RTOS) more or less mandatory. Even more, robustness requirements and the integration of functionality formerly implemented on a set of loosely coupled CPUs onto a single chip both require memory protection similar to a desktop or server system. The assumptions of the work in worst-case execution time (WCET) analysis have mostly been focused on the application domain. This involves, for example, the assumption of planar code or requirements of universal knowledge of memory accesses. The *Potoroo* project we have embarked in aims to analyse the NICTA developed version of L4 microkernel N1 embedded API. To avoid any confusion we will denote the L4 kerenel in NICTA N1 embedded API *L4 N1* throughout the paper.

*National ICT Australia 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.

In embedded systems, this kernel is mainly targeted for mission critical and consumer electronics systems. This paper discusses the issues, that need to be solved for this specific kernel. It also examines if and how these issues might be addressed by related work.

While Potoroo aims to be laregly target independent, the analysis is currently performed on an ARM [1] processor based platform and some of the issues reported are specific to the ARM architecture. The choice of ARM is mainly driven by the use of *L4 N1*. While ports to other architectures exist, the ARM port of this kernel is the most progressively developed. However, the processors implementing the ARM architecture do exhibit many of the problems experienced on other architectures and thus is a good reference point for this and future work.

To our knowledge Colin and Puaut [2] published the only other work addressing the WCET analysis of an operating system kernel, in their case the RTEMS kernel. However, RTEMS does not provide the memory protection offered by *L4 N1* and their work did not cover the entire kernel. Our work aims to cover all runtime relevant parts of the *L4 N1* kernel and aims to provide an environment which allows the WCET analysis for a given hardware platform outside the academic lab environment. Mehnert et al. [3] have looked into the cost of address spaces, but have not fundamentally addressed the issue of WCET analysis of the kernel. Commercially available kernels are sometimes delivered with representative sample execution times for some kernel primitives on a give platform. However, these execution times are always expressively exempted from being in any way guaranteed.

The next section will briefly introduce our approach to WCET analysis. Section 3 digs into a detailed analysis of the issues we have encountered during our effort to analyse *L4 N1*. Finally Sections 4 and 5 conclude the paper with an outlook into future work and a summary of the papers findings.

2 WCET Approach

Many issues will be explained in the context of our WCET approach. As such we consider it helpful to briefly introduce our approach and toolset prior to actually discussing the issues experienced during analysis so far.

Our approach is measurement based, but instead of using the end-to-end measurements and safety factors common in industry, we use measurements obtained on the basic-block level together with a tree and a timing schema to compute the WCET of the kernel primitives. Using the tree allows us to implicitly cover any possible path through a kernel primitive, thus ensuring that WCET is not underestimated based on the path executed. Opposed to end-to-end measurements basic blocks exhibit their respective WCET much more easily, as the numbers of different execution times is for a basic block are usually much smaller due to a limited number of variation causing input states and cache misses. However, in order to provide guarantees that the WCET has been observed on the basic-block level we are also working on a static analysis approach [4]. Within this work we are aiming to establish the number of cache misses which could and should be observed during the measurements performed and compare the results obtained by static analysis with the measurements. The base line is to avoid detailed hardware modelling, but rather to stick to first order effects like cache misses in order to keep the static analysis light weight.

Our toolset is depicted in Figure 1. In terms of building blocks it is very similar to the pWCET toolset used by Colin et al. [5, 6]. The major difference is a shift in associating the computational weight in terms of WCET onto the edges of the control flow graph (CFG) rather than the nodes used in the previous work. This increases accuracy of the analysis by separating effects caused for example by a branch prediction unit into separate entities. However, more important for our work are the changes in the subsequent tree representation and timing schema, which allow us to deal efficiently with non well-structured code.

The kernel binary image is the base for our analysis. By analysing primarily from the compiled and linked executable, we avoid second guessing the effects of the compiler. The generation of traces from the executable may either be intrusive via instrumentation code added to the executable or non intrusive, via hardware supported tracing mechanisms or cycle accurate simulations. Whenever available hardware supported tracing will be used, as it is non-intrusive and is not subject to the question of whether the simulator matches 100 % the hardware.

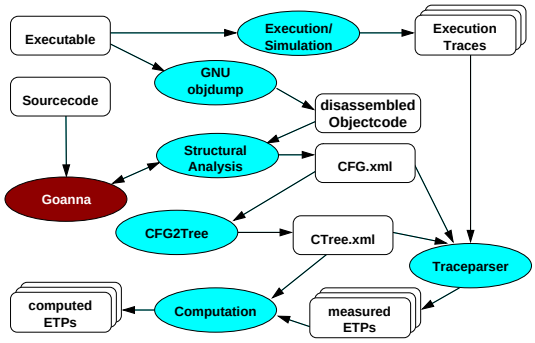

Figure 1: Toolchain Overview

The extraction of a control-flow graph (CFG) from the kernel binary image is split into three steps. In a first step `objdump` from GNU binutils is deployed to disassemble the code. This circumvents the problem of dealing with different binary formats. As such it minimises the hardware dependent part of the toolset. The second step of translating the code into a base CFG is left to a comparably simple program. Currently this program only deals with ARM code but can be ported to other CPUs with moderate effort. Besides analysing the output of `objdump` it also queries the *Goanna* [7] tool (which we have used as source code analyser/parser) to obtain additional information to fill in information missing in the object code analysis. The details of this interaction will be discussed in later sections of this paper. In a third and architecture independent step the CFG is augmented with various metadata which can be obtained with some effort from the base CFG. The metadata consists, for example, of loop-nesting levels, backward edges etc. This step has been separated from the second step to keep the architecture dependent part modular. In Figure 1, the second and third step are for simplicity of the representation joined into the process of *structural analysis*.

The CFG generated by the previous step is used to generate the tree representation with *CFG2Tree*. The parent nodes may either be of type sequence, alternative, or loop while the leaf nodes of the tree represent the transitions in the CFG. The leaf nodes may contain a call to another function. The traceparser uses the CFG, which actually describes already all the possible transitions which may occur, as well as the tree representation to convert the execution traces into measured ETPs. The traceparser does not only produce ETPs for all the leave nodes of the tree, but also of parent nodes. This is useful when trying to track down where and why overestimations are introduced in the later computation process. For this the computed and the measured ETPs

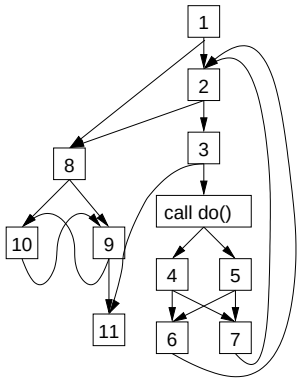

Figure 2: Sample Control-Flow Graph of Structures Found in the *L4 N1*Microkernel

can be directly compared.

Finally the computation stage takes the schema rules to produce ETPs for the parent nodes of the tree. Sequences form simple additions, alternatives use the $\max$ operator and loops use multiplication with the number of loop iteration and add the loop entry and loop exit.

3 WCET Analysis of an RTOS Kernel

In this section we discuss the main issues encountered when analysing *L4 N1*. However, similar problems can be expected when analysing any other kernel. The constructs and issues listed below are often used in operating system kernels or are caused by compiler optimisations. *L4 N1* source exhibits a reasonable number of these. As removal of these would heavily affect the performance of the kernel, we deem that we have to work around the problem, rather than avoiding it by imposing strict coding rules and switching compiler optimisations off. Figure 2 depicts a number of constructs in the CFG for illustration purposes. In reality the constructs are much larger and span up to several dozens of nodes in the control-flow graph.

3.1 Non Well-Structured Code

In an RTOS kernel there is deliberate deviation from structured coding, in particulara with regards to the use of `goto` statements. The current implementation of *L4 N1* for ARM processors contains more than 20 `goto` statements. This number is not including non well-structured code written in assembly. Use of such coding technique to optimise the kernel leads to deviation from properly defined structures and thus, introduces more complexity to resolve. For example, consider the edge between node-4 and node-6 (edge `4-6`) in Figure 2. Three other transitions namely, `4-7, 5-6, 5-7` provide alternate paths through which control can flow. Such non planar code can be resolved by duplicating some of the nodes in the tree.

Syntax-tree based approaches, like the one used by Colin and Puaut [2] as well as the approach by Theiling et al. [8], which uses an integer linear programming approach, should technically be able to deal with such code. However, well-structured code is a typical restriction of many static WCET analysis approaches.

A similar problem exists with irreducible loops as formed by nodes `9` and `10` in Figure 2. Again this can be resolved by duplicating nodes. Currently our toolset has not yet implemented an algorithm to identify irreducible loops like the one presented by Sreedhar et al. [9] and as such this is manually resolved.

3.2 Multiple Loop Exits

The use of `break` or `return` statements in loops leads to multiple points of exit out of these loops. *L4 N1* for ARM currently contains 18 break statements in loops. `return` statements within loops are translated by the `gcc` ARM compiler into branches to the end of the function. This leads to code being *virtually* shared by the loop exit which results in the main function body bypassing the loop if the loop is part of a conditional. The transition `3-11` in Figure 2 demonstrates such a situation. It is an issue, since the loop exit notionally stretches to the nearest common node outside the loop, which in this case is the return node at the end of the function. Such code exists in various locations like, for example, the IPC slowpath implementation. Within our approach this is solved by duplicating the code shared between the loop exit and the main sequence bypassing the loop.

3.3 Inline Assembly

The introduction of inline assembly text into the source code in turn introduces difficulties when querying a source code analysis tool like *Goanna*. This typically occurs in sections of kernel which are expected to be executed several more times than others. Assembly text is inserted in 23 places in the current *L4 N1* implementation.

3.4 Assembly Files

Besides inline assembly, the kernel also has a considerable code written in assembly. This covers, in partic-

```
pistachio/kernel/include/arch/arm/ptab.h:51
f0008170: e3520007  cmp     r2, #7  ; 0x7
;; Switch statement indirect jump
f0008174: 979ff102  ldrls   pc, [pc, r2, lsl #2]
f0008178: ea0000c2  b       f0008488
;; Jump table begins after an instruction
f000817c: f0008470  andnv   r8, r0, r0, ror r4
f0008180: f000847c  andnv   r8, r0, ip, ror r4
f0008184: f0008488  andnv   r8, r0, r8, lsl #9
f0008188: f000847c  andnv   r8, r0, ip, ror r4
f000818c: f0008494  mulnv   r0, r4, r4
f0008190: f000847c  andnv   r8, r0, ip, ror r4
f0008194: f0008488  andnv   r8, r0, r8, lsl #9
f0008198: f000841c  andnv   r8, r0, ip, lsl r4
```

Figure 3: Switch statement from *L4 N1* kernel objdump

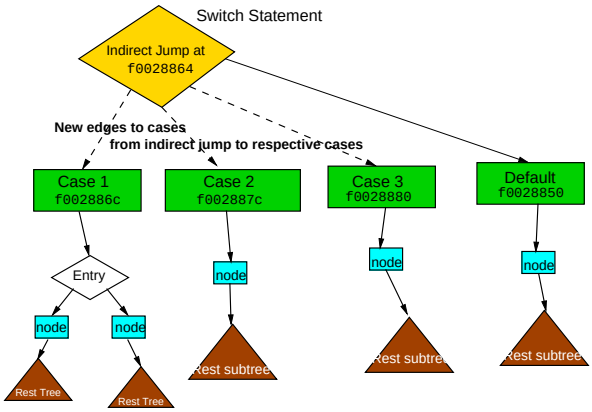

Figure 4: Corresponding graph after patching switch statement

ular, the trap code resolving interrupt handling and the performance critical IPC fastpath. Being highly optimised code, it adheres to little convention in terms of standard C or C++ compiled code. In particular it introduces irreducible loops such as the transitions `9-10` and `10-9` in Figure 2. As mentioned earlier such loops need duplication of nodes to be represented within the tree. Additionally, the detection of these loops and translation into tree is non-trivial. In particular, the irreducible loops within the kernel span more than 10 CFG nodes.

3.5 Indexed Jumping

Indexed jumping occurs when there are multiple cases in a `switch` statement. The compiler creates a hash table of all the `case` addresses and makes an indirect jump to these while optimising the code. Control flows to the respective `case` depending upon the address stored in a register or by directly indexing the hash table. Ultimately, we need to obtain edges to all possible branch targets contained in the *jump table*

As a first approach, line references in source code seem to be an answer to the location (address) of each `case` body. But this is not true since an optimising compiler distorts the resulting object code (in many cases even merges several cases, extracts common statements, etc).

Knowing the typical anatomy of switch statements, we can reconstruct the possible control flow with moderate effort by parsing the jump table itself. Parsing a jump table involves the main issue of identifying it correctly. This task becomes more difficult when the table is embedded in the code segment by the compiler. We have observed that the jump table always lies one instruction after the `switch` statement indirect jump. This behavior being constant in all our cases, including the *L4 N1* kernel, we chose to use this as an identifying criteria. (see Figure 3, code modified for best view)

We can patch the switch statement by extracting the addresses of each case statement and creating an edge to it. This is illustrated in Figure 4 where we have drawn three more edges from switch indirect jump node to the respective `case`.

3.6 Returns

The kernel version investigated contained a large number of register indirect jumps. Technically closely related to the above, the problem is that there is no jump table which can easily identified. In most cases these are compiler generated return statements. In ARM the register `lr` contains the return address for a function call. For recursive functions `lr` is pushed onto the stack which can be easily identified. However, in some cases the function is so small that it can make use another local register which is used to store `lr` prior to a call to another function and later moved onto the `pc` to implement a return statement. This requires tracking of registers within the tool evaluating the output of `objdump` to distinguish return statements from genuine register indirect branches or function calls. For the time being it is not planned to implement full tracking of all register content, but rather tracking of where the return address of a function is stored. Alternatively we might deploy lookups in the source code using *Goanna* to solve this problem.

3.7 Function call targets

This issue refers to a set of locations created by the programmer himself, which are not easily retrievable from

```
for (int i = 7; i < IRQS; ++i)  /* 0..6 are reserved */
{
 if (status & (1ul << i)) {
  void (*irq_handler)(int, arm_irq_context_t *) =
   (void (*)(int, arm_irq_context_t *))interrupt_handlers[i];
  irq_handler(i, context);
  return;
 }
}
```

Figure 5: *L4 N1* kernel interrupt vector array indexed in a loop

the object code. A good example of this type of coding would be an *interrupt vector table*, which distributes incoming hardware interrupts to registered handlers. In the *L4 N1* kernel code, a jump to these routines is made through an array of function pointers. Since the code is accessing a global array, ascertaining where this array is initialised is quite tedious in the sense that it may be initialised anywhere in the source code spanning considerable number of files. In addition, the initialisation may be obscure or actually happening dynamically at runtime. Since the source code analysis by *Goanna* has so far been unable to identify the content of global variables, manual analysis is the only way to describe these constucts. However, there are only few locations in the kernel which make use of this kind of function calling and thus the required manual intervention is limited.

Besides knowing the targets of called functions, there is also the issue of encoding it appropriately in the CFG and tree. Since our toolset allows only one `call` per CFG node, we need to circumvent the problem. We have done that by allowing stand alone function call nodes, which have no measured execution time themselves. This is useful for encoding alternative functions to be called and is used, for example, in the *L4 N1* kernel debugger where depending upon an enviroment variable either one or the other function is called. Although the kernel debugger is irrelevant for our analysis, there are some constructs in this part of the code which are interesting for analysis. Furthermore, constructs similar to these may be included later during development of kernel. A special case is where the function array is indexed by a loop control variable or any descendant of that as has been used in the *L4 N1* interrupt vector table (see Figure 5, extract taken from `irq.cc`). In this case only one of the target functions is executed for an interrupt. However, the latency is different since the code checks each bit of the interrupt mask with each loop iteration until it hits the correct one and calls the handler function.

We can apply this strategy in conjunction with unrolling of loops to solve the problem of multiple targets where an interrupt vector array has been used in

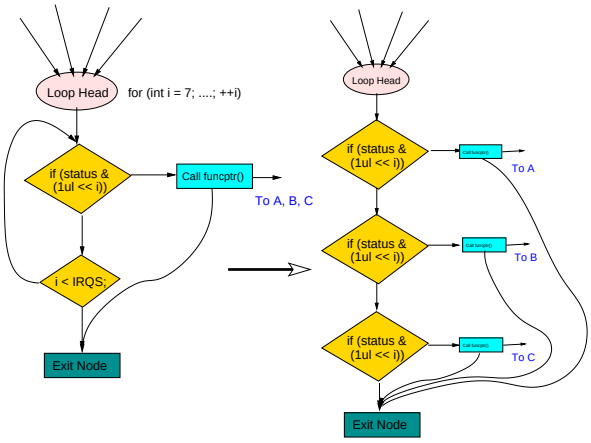

Figure 6: Stand alone nodes with loop unrolling technique for interrupt vector array

source code to address different interrupt handlers (see Figure 6).

3.8 Context Switch Jumps

Context switches are subject to three problems.

1. Context switches are nontrivial to positively identify without creating false positives. Thus it is considered inevitable to manually identify these. However, again there are only very few places in the kernel that actually perform a context switch, keeping the intervention at this stage very small. A result of the context switch is that the execution may transfer from any of the context switch nodes to any other node containing a context switch, requiring our tool to provide appropriate transitions in the automata performing the trace parsing.

2. After a context switch, an *asynchronous notification* may be delivered. This happens because a notification is issued when the sending thread is running and the receiving thread is not (this assumes a single threaded, single processor CPU). The current implementation of an *asynchronous notification* pushes a notification stack frame onto the stack of the receiving thread, thus executing the receive function prior to resuming the execution when the context switch passes control to the receiving thread. This is resolved by notionally adding a context switch to the start and end of the notification routine, allowing the *Traceparser* (which is part of the toolset) to switch to the notification and back from it. The time of the asyn-

chronous notification needs to be dealt with, depending on what the result of the analysis is to be used for. In the case of latency analysis, the time for the *asynchronous notification* needs to be added to the called function while for schedulability analysis, this needs to be considered separately as part of the communication cost.

3. Finally from a trace parsing point of view, performing a context switch means a `return` after a context switch no longer corresponds to the `call` performed before doing the context switch. As such again, the *Traceparser* needs to allow for `return` statements to connect to any possible location than only from where the returning function was called. Unfortunately this takes away some of the sanity checking available when doing the analysis such as checking that a `return` is returning to the place the function was called from. However, the likelyhood of a trace which is corrupted in such a particular way is very low.

3.9 Portability

As opposed to application code, which is built on top of hardware abstractions (standard libraries, and kernel), the kernel is supposed to be deployed on a variety of differebt systems. Having multiple target architectures supported by the kernel requires that the WCET estimation technique be portable, since it needs to be available for all target architectures. This is a major challenge as the work over the years has shown that this is not a trivial task. The support for multiple architectures is often reflected by many `#defines`. The `#define` is efficient and easy to use, but makes the code harder to read. Since manual intervention in the analysis is almost inevitable, the analysis of a kernel requires detailed knowledge of the kernel as well as fundamental understanding of WCET analysis.

3.10 Memory Management

A problem reported by Colin and Puaut [2] for the RTEMS kernel is code in the memory allocation can produce extremely long execution times that exceed the average execution time by a large margin. *L4 N1* suffers in the same manner during the unmapping of memory regions. However, this is only of theoretical relevance, as we would expect real-time applications to only make use of this system call when shutting down or doing exception handling. Obviously this still leaves the issue of non real-time applications effectively blocking the kernel while performing such a call as they shut down. It is expected that the issue will be addressed on the kernel side by changing the way memory management is handeled.

3.11 Parametric WCET

Run-time parameter dependent worst case execution times have also been experienced by Colin and Puaut [2]. This can either be caused by system parameters (e.g., the number of threads sending messages to a specific thread in the system) or caused by structural parameters (e.g., what kind of inter-process communication (IPC) is used for a specific system call in *L4 N1*). The IPC example is caused by the fact that all IPC functions in *L4 N1* use the same system call, but with different parameters.

As kernel code is quite complex to understand, the analysis of this code needs expertise in WCET analysis and OS construction. Otherwise the intrinsic interaction between different parts of the kernel may be misinterpreted or overlooked completely. In order to separate out different invocations of the same primitive (e.g., receive only IPC, send only IPC, IPC payload size, etc.) we currently need to manually remove irrelevant parts of the respective CFG. Future versions will use code annotations to identify different parts of the primitives.

3.12 Rapid Evolution

L4 N1 suffers from a very specific problem. The code is not fixed, but evolves rapidly over time, while at the same time the approach to analyse it is being developed and refined. Opposed to applications which are written and then deployed, different snapshots of the kernel will be deployed. Thus the analysis has to be performed repeatedly on slightly different versions of the kernel. This offers problems and opportunities. On one hand it requires the WCET approach used to require minimal user interaction, on the other hand it enables the use of annotations in the code.

Furthermore *L4 N1* uses memory protection and virtual addressing, which distinguishes it from most of the real-time operating systems around and is motivated by the fact that partitioning and fault isolation is a highly desirable feature in complex embedded systems. Static analysis requires the modelling of translation-lookaside buffers (TLB), which adds to the state space. For measurement-based approaches, this adds to the variability of the code, depending on the number of TLB misses. However, the kernel itself currently makes little use of the virtual memory, but applications analysed on top do suffer from this.

L4 N1 is coded in C++. While only a very limited subset of C++ is chosen, it nevertheless creates an additional engineering effort in the analysis approach. However, on the other hand, *L4 N1* as a microkernel is small compared to monolithic kernels which in turn makes the analysis much more tractable.

4 What's Next

Future work can be split into two categories. One which is related to development of the kernel itself and the other which looks at future tool enhancements.

So far the work has been carried out on a working snapshot of the kernel. Due to the experimental nature of the work it does not seem practical to track all changes to the kernel as they are made. The most substantial change of *L4 N1*in the last half year was the move to a single stack kernel. This implies the dissolution of the call/return relationship and subsequently a substantial change in the context switch modelling.

The next step is to look into a multi-processing version of the kernel. This includes looking at more fundamental issues of real-time in multi-processing environments and is in itself a large project.

On the tool-set and approach side of things, further automation and support for other architectures is on the agenda. This specifically covers the areas of

- register tracking, to automatically resolve more control flow instructions;
- irreducible loop identification and resolution;
- allowing for source code annotations to be taken into account.

The source code annotations are particularly relevant to provide separate WCETs for different but closely related kernel primitives. Besides these automation issues, we also want to continue working on the static analysis support for the approach [4].

5 Conclusion

In this paper we have listed a number of issues we have encountered in our effort to analyse the *L4 N1* microkernel for the WCET of all kernel primitives and how we resolved these. While we have mainly looked at *L4 N1* the insights should translate to a number of other kernels. The small footprint of *L4 N1* compared to monolithic kernels has certainly been helpful in keeping complexity of the analysis within managable levels. Besides the worst-case analysis, the approach can support kernel development in a number of ways. Hot-spot analysis can identify code portions which account for larger parts of the execution time both in terms of execution frequency as well as execution time, and thus help directing optimisation efforts. Furthermore our apporach can also be applied to detect dead code or code not covered in the regression tests.

References

[1] *ARM 7TDMI Data Sheet*, August 1995. ARM DDI 0029E.

[2] 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*, (Delft, Netherlands), pp. 191–198, June 13–15 2001.

[3] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, (Austin, TX, USA), 2002.

[4] 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.

[5] A. Colin and S. M. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, (Cancun, Mexico), Dec. 3–5 2003.

[6] 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*, (Austin, Texas, USA), pp. 279–288, Dec. 3–5 2002.

[7] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Goanna — A Static Model Checker," in *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems*, (Bonn, Germany), Aug. 2006.

[8] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by spearated cache and path analysis," *Journal of Real–Time Systems*, vol. 18, pp. 157–179, 2000.

[9] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying loops using dj graphs," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 6, pp. 649–658, 1996.