

# Trustworthy programming for multiple instruction sets

Principal Investigator: Mike Gordon (Mike.Gordon@cl.cam.ac.uk)

Researcher Co-Investigator: Magnus Myreen (Magnus.Myreen@cl.cam.ac.uk)

## Part 1: Track Record

The proposed research builds upon previous work at Cambridge on the formal verification of ARM processor hardware and assembly level software. We will use formal models of the ARM4T instruction set architecture (ISA) resulting from the completed EPSRC project *Formal Specification and Verification of ARM-based Systems* (GR/T20106/01).

The aim of the project is to continue and broaden work started in Magnus Myreen's PhD and to deepen existing international collaborations, particularly with Professor Slind at the University of Utah. Slind's group already makes use of tools and techniques developed by Myreen, and we make use of their work.

### Mike Gordon (Principal Investigator)

Mike Gordon works on the application of theorem proving to formal verification of hardware and software. He leads the Hardware Verification Group (HVG) at the University of Cambridge Computer Laboratory. HVG has been in existence for over twenty years and members have worked on a wide variety of formal specification and verification problems including low-level hardware, processors, floating point algorithms, probabilistic algorithms and the analysis of mixed hardware software systems. The tool used for this work is the HOL system, which was initially developed at Cambridge. The current version is HOL4, which is a thriving open source project hosted by SourceForge with 25 registered developers.<sup>1</sup>

Ex members of HVG include past and present faculty at universities in the UK, Australia, Italy, North America and Hong Kong, researchers in government facilities in France, the USA and Australia and technical staff of several international companies.

HVG is part of the Automated Reasoning Group (ARG), which is jointly led by Professors Paulson and Gordon. This group is responsible for both the HOL and the Isabelle proof assistants as well as a number of other tools (e.g. Joe Hurd's Metis theorem prover which is both standalone and incorporated into HOL4 and Isabelle).

### Magnus Myreen (Researcher Co-Investigator)

Magnus Myreen is a third year PhD student supervised by Mike Gordon. He has published three papers on his work in refereed international conferences and coauthored a refereed conference paper. In his paper at TACAS 2007 [14], Myreen described how a Hoare logic can be defined so as to fit on top of a realistic model of a processor's ISA. His FSEN 2007 paper [13] presented how the Hoare logic has been instantiated to a detailed model of ARMv4, which has been certified against a hardware implementation of an ARM processor [8] as part of the completed EPSRC project GR/T20106/01. In his workshop paper at TPHOLs 2007 [12], Myreen outlines how the Hoare logic has been used in small to moderate sized manual proofs of ARM machine code. His forthcoming thesis, entitled *Formal verification of realistically modelled machine code*, will present his machine code Hoare logic, its instantiations (ARM and IA-32) and outline how it eases the proof effort of verification to a tractable setting without unsound assumptions or simplifications.

The project proposed here will require automation of parts of proofs involved in code verification. Myreen has prior experience in automation of verification proofs from two internships with professor Ralph-Johan Back at Åbo Akademi University in Finland. During these internships Myreen developed a verification condition generator (VCG) [1], which is now used in teaching formal methods at Åbo Akademi University.

---

<sup>1</sup><http://sourceforge.net/projects/hol/>

# PART 2: Proposed Research and its Context

## Overview and goals

The rapidly growing use of microprocessors in devices containing sensitive data (e.g. phones) and safety-critical systems (e.g. automobiles, avionics) is increasing the value of trustworthy software. Assembly code is particularly error-prone as it varies from processor to processor and even between different versions of an instruction set architecture (ISA) for the same processor family.<sup>2</sup> Some software must be implemented directly in assembler, such as run-time system components (e.g. storage management), performance-critical operations (e.g. arithmetic) and parts of operating systems (e.g. interrupt controllers). One cannot avoid having to create at least some code running on “bare metal”.

Our goal is to develop a new programming methodology for creating trustworthy assembly code software. The project has two parts:

- ▶ bottom-up creation of certified code components using proof-producing decompilation of assembly code into mathematical function definitions (**function extraction**);
- ▶ top-down compilation of certified implementations from mathematical function definitions by combining previously certified components (**code synthesis**).

The certification aspects are novel: they consist of automatically proving a new kind of processor-specific Hoare formulae semantically embedded in higher order logic.

Unlike other recent work on certified assembly code, we aim to go beyond establishing weak safety properties and instead handle functional correctness, termination and resource usage. We aim to generate deep proofs using very accurate ISA models. Our goals are complementary to the relatively shallow analyses based on the simplified semantics that underlie current industrial-scale bug-finding formal software verification tools.

Our methods are not tied to a particular instruction set. Initially we will work with two instruction sets: ARM and a subset of IA-32, both of which are very widely used. We already have access to formal specifications of these.

We aim to conduct diverse and realistic case studies, including multi-word arithmetic as used in cryptography and storage allocation and management routines used for runtime support of compiled code. Towards the end of the project we hope to verify a complete interpreter for a simple language based on pure LISP supporting high precision arithmetic — a first step towards creating verified implementations of functional languages on bare metal.

A long-term application, probably beyond the scope of this project, is creating certified run-time code for real domain-specific functional languages. A motivating example is the Haskell-based Cryptol language [9], which is used for specifying cryptographic algorithms.

We plan to recruit a PhD student to explore the feasibility of creating verified operating systems components such as drivers, networking software, software-hardware interfaces, boot loaders and virtualisation support. This requires modelling parts of the hardware environment. Verifying a complete operating system is too much for a single PhD student, but we intend to collaborate with the University of Utah (see attached letter from Prof. Slind).

## Research programme and methodology

In recent years processor models, programming logics and theorem proving have advanced to a stage where one can potentially build small, but real, software applications and provide proof of correctness based on solid semantical foundations. We aim to explore this potential.

To verify primitive components, we plan to extract automatically a function describing the state changes computed by code segments (function extraction). This function is then used to create a correctness certification. Properties of the function can then be deduced using methods that do not need to deal with the full details of a particular ISA.

We also plan to synthesise code from higher-level functional specifications. The compiler doing the synthesis will be parametrised on the ISA and on the library of verified components.

---

<sup>2</sup>For example, ARMv4T has 16-bit instructions not found in earlier versions of the ARM ISA, such as ARMv4.

## Function extraction

The following example illustrates the idea of function extraction on ARM and IA-32 code that compute the length of a linked-list.

<i>arm_code</i>	<i>x86_code</i>	comments
	<code>xor eax, eax</code>	– set r0/eax to 0
<code>mov r0,#0</code>	<code>loop: test esi, esi</code>	– compare r1/esi with 0
<code>L: cmp r1,#0</code>	<code>jz exit</code>	– if r1/esi is not 0:
<code>ldrne r1,[r1]</code>	<code>mov esi, ds:[esi]</code>	– load r1/esi into r1/esi
<code>addne r0,r0,#1</code>	<code>inc eax</code>	– increment r0/eax
<code>bne L</code>	<code>jmp loop</code>	– jump to compare
	<code>exit:</code>	

The extracted functions,  $f$  and  $g$  below, show how the differences between ARM and IA-32 code largely disappear when abstracted to functions ( $\oplus$  is bitwise exclusive-or, so  $w \oplus w = 0$ ).

$f(r_0, r_1, m) = \text{let } r_0 = 0 \text{ in } f_2(r_0, r_1, m)$	$g(eax, esi, m) = \text{let } eax = eax \oplus eax \text{ in } g_2(eax, esi, m)$
$f_2(r_0, r_1, m) = \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else let } r_1 = m(r_1) \text{ in let } r_0 = r_0 + 1 \text{ in } f_2(r_0, r_1, m)$	$g_2(eax, esi, m) = \text{if } esi = 0 \text{ then } (eax, esi, m) \text{ else let } esi = m(es) \text{ in let } eax = eax + 1 \text{ in } g_2(eax, esi, m)$

We aim automatically to extract such mathematical functions from assembly code. If the code has loops, like the example above, then we will generate a recursive function. We plan to use the proof that the recursion terminates to suggest the induction needed to prove that the function correctly characterises the execution of the code on the processor.

A challenge of the proposed research is to scale our preliminary ideas on function extraction from trivial examples, like the code above, to significant real applications. We are optimistic that scaling will be feasible since the method is compositional, so can be applied to component code fragments and the resulting extracted functions combined.

We will certify that extracted functions accurately describe the code using processor-specific Hoare triples. For example, the fact that the code examples above execute the functions  $f$  and  $g$  would be certified by the following theorems, whose proofs we plan to fully automate:

$\{ R0 \ w_0 \ * \ R1 \ w_1 \ * \ MEM \ m \ * \ PC \ p \}$
<i>arm_code</i>
$\{ \exists w'_0 \ w'_1 \ m'. (w'_0, w'_1, m') = f(w_0, w_1, m) \ * \ R0 \ w'_0 \ * \ R1 \ w'_1 \ * \ MEM \ m' \ * \ PC \ (p+5) \}$
$\{ EAX \ w_0 \ * \ ESI \ w_1 \ * \ MEM \ m \ * \ PC \ p \}$
<i>x86_code</i>
$\{ \exists w_0 \ w_1 \ m. (w'_0, w'_1, m') = g(w_0, w_1, m) \ * \ EAX \ w'_0 \ * \ ESI \ w'_1 \ * \ MEM \ m' \ * \ PC \ (p+6) \}$

The first Hoare triple says that if the register R0 holds  $w_0$ , register R1 holds  $w_1$  (where  $w_0$  and  $w_1$  are 32-bit words), the memory contents MEM is  $m$  (a partial function from 32-bit words to 32-bit words) and the program counter is  $p$ , then if *arm\_code* is executed on an ARM the new values of R0, R1 and MEM are given by the 3-tuple returned by function  $f$  and the program counter is incremented by 5. The second triple is similar, but its semantics is defined in terms of IA-32 execution.

The semantics of our Hoare triples is local: resources such as registers, flags and coprocessor state that are not explicitly mentioned in the precondition (the ‘footprint’) remain unchanged, so need not be mentioned. This solution to the frame problem, using the separating conjunction  $*$ -operator, originates from separation logic [15].

The definition of such a Hoare-triple has been developed by Myreen and is described in recent papers [14, 13, 12]. The proposed project will develop a methodology and tool support

for creating such Hoare-triple certificates automatically, rather than manually as they have been created so far.

Once functions describing code segments are extracted, the Hoare theorems can be used to establish correctness properties. For example, if  $list(a, m, xs)$  specifies that an abstract list  $xs$  is represented in memory  $m$  by a linked list at address  $a$ .<sup>3</sup>, then it follows that

$$list(j, m, xs) \Rightarrow (f(i, j, m) = (length(xs), 0, m))$$

which is easy to prove and does not depend on details of ARM. By Hoare logic it then follows:

$$\begin{aligned} & \{ R0\ w_0 * R1\ w_1 * MEM\ m * list(w_1, m, xs) * PC\ p \} \\ & \text{arm\_code} \\ & \{ \exists w'_0\ w'_1\ m'. (w'_0, w'_1, m') = f(w_0, w_1, m) * \\ & \quad R0\ w'_0 * R1\ w'_1 * MEM\ m' * (w'_0 = length(xs)) * PC\ (p+5) \} \end{aligned}$$

The mathematical functions  $f$  and  $g$  can trivially be proved equal (as  $w \oplus w = 0$ ), hence, for this example, we get for free that the IA-32 code also computes the length:

$$\begin{aligned} & \{ EAX\ w_0 * ESI\ w_1 * MEM\ m * list(w_1, m, xs) * PC\ p \} \\ & \text{x86\_code} \\ & \{ \exists w'_0\ w'_1\ m'. (w'_0, w'_1, m') = g(w_0, w_1, m) * \\ & \quad EAX\ w'_0 * ESI\ w'_1 * MEM\ m' * (w'_0 = length(xs)) * PC\ (p+6) \} \end{aligned}$$

This example illustrates the goal of hiding the ISA-specific details from the verifier.

Standard techniques for data refinement (coupling invariants) will be used to lift reasoning to more natural abstract data types such as integers, sets, finite lists and trees.

## Code synthesis

Verified components such as storage management routines (e.g. malloc) or cryptographic algorithms (e.g. Montgomery multiplication) need to be combined into software applications. Assembling software components raises trustworthiness challenges. We propose to use a formal synthesis method to compile specifications to machine-specific assembly code.

Our tools will be parametrised on a library of previously certified components (characterised by Hoare triples) as well as an ISA model. The tools will compile functional-program style specifications (expressed in higher order logic) into assembly code plus Hoare logic correctness certificates. Thus when compiling “let  $y = h(x)$  in  $\dots$ ”, if  $h$  is a function with an already certified implementation then a link to the code implementing  $h$  will be created and the proof certificate for  $h$  used to prove a certificate for the program being synthesised. When functions like  $h$  operate on more abstract data-types than words, the corresponding certifying theorems in the library will involve both operation refinement and data refinement.

The code synthesis work is planned to be a collaboration with Professor Slind's group at the University of Utah. They have developed a proof-producing compiler that converts recursive function definitions in higher order logic into equivalent functions that correspond to ARM code (variable names reflect names of ARM resources and each let-expression has a corresponding ARM instruction). We have already collaborated with Utah to implement a proof-of-concept back-end to their compiler that synthesises true ARM assembler from the output of the compiler, together with certifying Hoare triples. This is currently being extended to synthesise IA-32 code as well. The synthesis part of the work proposed here has emerged from this collaboration.

Starting from the output language of the Utah compiler, our plan is to develop richer machine-independent abstract assembly languages. For the cryptography arithmetic example we will develop a language with virtual registers that can hold multi-precision numbers. For the storage management example we plan to explore a language that has garbage collected virtual registers that can contain abstract values representing trees. We will synthesise real

<sup>3</sup>The shape predicate  $list$  depends on the structure of list cells: adjacent byte-addressed 32-bit words, with the first word being the next-cell pointer (or 0) and the second word the cell contents. The predicate  $list$  is thus defined by:  $list(a, m, []) = (a = 0)$  and  $list(a, m, (n, d) : l) = ((m(a) = n) \wedge (m(a+4) = d) \wedge list(n, m, l) \wedge (a \neq 0))$ .

ARM and IA-32 assembler from these languages. As we progress through case studies the abstract languages are expected to get richer. For example, only simple trees are needed for the initial storage management study, but we anticipate needing more complex trees (e.g. with various types of atoms including multi-precision numbers) for the pure LISP example.

We will collaborate with the Utah group so that our machine-independent languages mesh well with their compiler, e.g. so we can use it to generate input to our synthesis tools.

## Case Studies

Over the course of the project, we aim to conduct case studies which form a coherent 'stack' built on bare metal and that demonstrate significant applications. As case studies, we will initially create certified components for multi-word arithmetic and storage allocation and management. We will then use these components with the compiler to synthesise an interpreter for a language based on pure LISP supporting high precision arithmetic. This interpreter will then be used to build a simple engine for elliptic curve cryptography. All examples will be carried out on at least two architectures, ARM and IA-32.

## PhD Project: Verification of operating system components

Investigating how the above ideas can be applied in verification of operating systems components is an open-ended part of this project. It is to be done by a PhD student and is likely to be example driven. Initial examples we have in mind include an interrupt controller and thread scheduler. The interrupt controller was suggested by Professor John Regehr from the University of Utah. We would aim to look both at interrupts from peripherals and timer interrupts. The latter would lead into thread scheduling. An application of scheduling benefiting from certification would be the design and implementation of a very simple separation kernel to run multiple non-communicating threads. This work would use our Hoare logic tools to verify sub-routines for management of the data-structures used by the scheduler. The goal here is to investigate verification both that the threads cannot influence each other and also that each thread is safe and functionally correct in its environment.

## Research Schedule and Tasks

We will study and implement function extraction for ARM and IA-32 code and use this to create a library of certified core components for our case studies.

In collaboration with Professor Slind, we will then design and implement code synthesis tools for assembling certified components into applications.

The following research tasks are planned. Dependencies and scheduling are given in the separate one-page diagram.

- T1** Design and implement the creation of certified Hoare triples using function extraction.
- T2** Demonstrate function extraction on case studies:
  - 1. cryptography components, e.g. Montgomery multiplication;
  - 2. garbage collection, e.g. Cheney collector;
  - 3. functional language primitives: e.g. cons and tree-equality.
- T3** Develop Hoare triple producing code synthesis
- T4** Demonstrate code synthesis by case studies:
  - 1. a small engine for elliptic curve cryptography;
  - 2. implement a small functional language;
  - 3. add support for cryptographic primitives to functional language.
- T5** Investigate the verification of operating system components (PhD project).

Task **T2** and **T4** will measure how well the result of tasks **T1** and **T3**, respectively, apply to moderately sized applications.

Tasks **T1**, **T2**, **T3** and **T4** are to be done by the named researcher, Magnus Myreen. The PhD student will do **T5** and may collaborate on parts of tasks **T2** and **T4**.

## Background and Related work

Related work falls into two categories: reasoning directly at the assembly code level, and creating correct code via certifying compilation.

The classical work in the first category is Boyer and Yu's verification of M68020 assembly code with the Boyer-Moore theorem prover [2]. This approach was impressive for its time, but got bogged down in machine specific details: our approach aims to hide these details in the semantics of Hoare triples that are parametrised on an ISA model. Boyer and Yu also needed to explicitly specify changes to the entire machine state, including unchanging components. We plan to use local reasoning to simplify proofs and automatically manage unchanging state components (the frame problem) with a separating conjunction operator.

More recently Zhong Shao's group at Yale [7] use the Coq proof assistant to verify assembly code, including garbage collectors (GCs). We have verified some of Shao's examples, and found that our proofs are much shorter. For example, a verification of a mark-and-sweep GC took Shao's group 13000 lines of Coq, while we did the same example in less than 3000 lines of HOL. Another example GC took 7000 lines of Coq, but our verification was 1000 lines of HOL. Our proofs were manually guided. The automation we propose (extraction of computed functions) is likely to reduce the verification effort further. We do not yet understand why our proofs are so much shorter: this is something we would hope to investigate.

Foundational proof checking [16], Typed Assembly Language [17, 5, 4] and Certified Assembly Programming [3] mostly aim to check relatively weak safety properties. In contrast, our work aims at proof of full functional correctness.

The most significant current work in the second category – verifying compilation – is the research, discussed above, by Professor Slind and his group at the University of Utah [11].

Somewhat tangentially related to our goals is certified compilation of subsets of C. Leroy's certification (verification), using Coq, of a compiler from a subset of C (Cminor) to PowerPC code is a milestone achievement [10]. The Verisoft project [19] aims to use Isabelle to verify a compiler from another subset of C (C0) to a generic academic RISC machine language (DLX). This work is impressive, but it does not address the verification of the assembly code that actually runs, rather it provides a way of converting C code to 'equivalent' machine code. Proof of correctness of C code is a major challenge, since getting an accurate formal semantics of a non-trivial subset of C is notoriously hard. Finding trustworthy and practical methods of verifying C source code is certainly a valuable goal, but not one we are addressing here. Like Boyer and Yu, we can verify code generated from C (by running a compiler and then verifying the resulting assembly code), but we do not propose to attempt reason about C source code directly.

The proposed PhD work is closer to bare metal than most other work we know of, but will benefit from interaction with higher-level operating system verification projects, such as the L4 project at NICTA in Australia [18, 6] which aims to certify significant functional properties of an operating system microkernel written in C using the Isabelle proof assistant.

## Beneficiaries and Collaborators

This work should interest anyone creating trustworthy implementations, on bare metal, either from scratch or using existing code. The cryptography case study will be of particular interest to those developing assembly code software for high assurance certification (e.g. Common Criteria EAL7), especially if they need to support their IP on multiple architectures.

Our storage management and functional programming case studies are relevant to implementers of high assurance software written in functional languages like ML, F# or Haskell. We hope to liaise with Galois Inc, who have a business interest in this.

We will collaborate with Professor Slind, as described above, and have included a request for funds to support visits for research discussions. We hope also to visit and exchange ideas with the groups doing related work in Coq at Yale, Princeton and Harvard (see *Related Work* above). The planned PhD research will benefit from liaison with the L4 operating system verification work at NICTA in Australia.

## Project management

The project will be managed by the Principal Investigator, Mike Gordon, who will meet regularly with the Researcher Co-Investigator, Magnus Myreen, and the PhD student. Myreen is to be the second supervisor to the student, which will provide him with experience and training in research supervision.

## Dissemination and Exploitation

Dissemination of the results of the project will occur via the normal channels of conference presentations and journal publications. In addition, we will establish and maintain a web site devoted to making results and progress reports available during the project.

## Resources requested

Support for one full-time postdoctoral researcher (Magnus Myreen) and one PhD student (to be recruited) is requested, together with standard equipment, consumables, project-specific infrastructure and travel resources. We also request funds to enable face-to-face research meetings with Prof. Slind and others at Utah (see the separate *Justification of Resources*).

## References

- [1] Ralph Back and Magnus Myreen. Tool support for invariant based programming. In *12th Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, Dec 2005.
- [2] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
- [3] The FLINT Group. Yale University. <http://flint.cs.yale.edu/flint/research.html>.
- [4] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 208–219, New York, NY, USA, 2003. ACM.
- [5] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 160–171, New York, NY, USA, 2006. ACM.
- [6] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA, May 2007. Online proceedings at <http://www.usenix.org/events/hotos07/tech/>.
- [7] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. Technical Report YALEU/DCS/TR-1336, Dept. of Computer Science, Yale University, New Haven, CT, November 2005. <http://flint.cs.yale.edu/publications/sbca.html>.
- [8] Anthony Fox. Formal specification and verification of arm6. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40, Rome, Italy, September 2003. Springer.
- [9] The Cryptol domain specific language for Cryptographic applications, Galois Inc., Portland, Oregon, USA. <http://www.cryptol.net/>.
- [10] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
- [11] Guodong Li, Scott Owens, and Konrad Slind. Structure of a proof-producing compiler for a subset of higher order logic. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2007.
- [12] Magnus Myreen and Michael Gordon. Verification of machine code implementations of arithmetic functions for cryptography. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07, pages 170–179, 08 2007.
- [13] Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for ARM machine code. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.
- [14] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2007.
- [15] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19, London, UK, 2001. Springer-Verlag.
- [16] Certified Program Verification with Coq. <http://proofos.sourceforge.net/>.
- [17] Typed Assembly Language Compiler. <http://www.cs.cornell.edu/talc/>.
- [18] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification — now! In Margo Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.
- [19] Verisoft: a long-term research project funded by the German Federal Ministry of Education and Research. [http://www.verisoft.de/index\\_en.html](http://www.verisoft.de/index_en.html).