Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Some Assembly Required – Program Analysis of Embedded System Code

Ansgar Fehnker, Ralf Huuck, Felix Rauch and Sean Seefried
National ICT Australia*, University of NSW, Locked Bag 6016, Sydney NSW 1466, Australia

Abstract

Programming embedded system software typically involves more than one programming language. Normally, a high-level language such as C/C++ is used for application oriented tasks and a low-level assembly language for direct interaction with the underlying hardware. In most cases those languages are closely interwoven and the assembly is embedded in the C/C++ code. Verification of such programs requires the integrated analysis of both languages at the same time. However, common algorithmic verification tools fail to address this issue. In this work we present a model-checking based static analysis approach which seamlessly integrates the analysis of embedded ARM assembly with C/C++ code analysis. In particular, we show how to automatically check that the ARM code complies to its interface descriptions. Given interface compliance, we then provide an extended analysis framework for checking general properties of ARM code. We implemented this analysis in our source code analysis tool Goanna, and applied to the source code of an L4 micro kernel implementation.

1 Introduction

Embedded system software differs from general application software in that it is typically a part of the trusted computing base, i.e., embedded system software has direct access to the memory and the underlying hardware. This makes its correctness crucial for the overall correctness of the embedded system. However, embedded system software is difficult to analyze as it is often highly optimized, uses programming constructs that are hard to analyze (such as pointer arithmetic), and typically implements the interaction with hardware as embedded assembly code. Guaranteeing full functional correctness requires proof-based methods such as interactive theorem-proving which is labor- and time-intensive. However, product cycles in the embedded system industry are shrinking and a high pressure-to-market calls for automated methods. Model checking and static analysis are two algorithmic verification techniques which are used to prove limited correctness or at least to detect as many bugs as possible automatically.

Algorithmic analysis approaches for C/C++ programs with embedded assembly code have mainly been pursuing either of the following two approaches:

1. Whenever checkers encountered embedded assembly code they have been optimistic in the analysis, i.e., do not report any warnings around those embedded code fragments, or they have been conservative by estimating any potential effect the embedded code can have and raised many (unnecessary) warnings [10, 14, 17, 18]. Both approaches are not fully satisfactory.

2. Other checkers perform the analysis on the assembly level for the whole program by analysing the disassembled object code [2, 21] and, if possible, taking debugging information into account. While this allows one to deal with the embedded code, it creates problems of re-discovering structures and data types of the original high-level C/C++ code, which might be impossible in the presence of optimizing compilers, and results in an analysis that has to deal with partial information. It also results in a much larger state space which can significantly slow down model-checking.

This work presents a model-checking based static analysis approach for C/C++ programs with embedded assembly code that is a compromise of the aforementioned solutions. Our analysis is performed at the level of the higher-level C/C++ code, while fully accounting for the connection between the two languages. We use the *interface description* of the embedded assembly code and map this by *operand aliasing* to the higher-level C/C++ code. This enables a seamless integration into the high-level analysis and remedies many issues mentioned in item 1 of the previous paragraph. Moreover, we extend the analysis framework to assembly code and check for compliance of the embedded code with its defined interface, which allows us to check

*National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.15

for violations not addressed by the approaches 1 or 2. And, finally, we introduce the possibility for statically checking arbitrary properties of the assembly code by mapping the assembly code to C, rather than mapping the surrounding code to the lower assembly level.

This work specifically addresses embedded ARM assembly [5], i.e., the ARMv6 instruction set, for the simple reason that ARM has gained by far the largest market share in the embedded systems sector. However, the general principles are valid for most other architectures.

The remainder of this paper is organized as follows: In Section 2 we give a presentation of our general analysis approach and extend this approach to embedded ARM assembly in Section 3. We present qualitative results of our implementation in the tool Goanna in Section 4, followed in Section 5, by a discussion of related work. Finally, in Section 6, we discuss current limitations of our tool, ideas for future work and our conclusions.

2 Syntactic Software Model Checking

This section describes how to statically check properties of source code. We follow a model-checking based approach to solving static analysis problems as first introduced by Schmidt et al. [23, 22]. A similar approach has also been taken by [16, 11]. Using a model checker for solving static analysis problems has a number of advantages. All properties can be expressed in a single, flexible analysis engine, making it easy to add new checks by adding new checking properties. In addition, the analysis scales well with increasing number of properties. The details of our path-sensitive, intra-procedural analysis can be found in [13].

The basic idea is to annotate the control flow graph (CFG) of a program by atomic propositions of interest. In order, to check, e.g., for uninitialised variables, we can identify atomic propositions $decl_x$, $read_x$ and $write_x$, representing program locations where a variable x is declared but not initialised, read from, or written to, and we mark those locations in the CFG accordingly. The resulting model can be viewed as a Kripke structure and we can model check it for properties of interest.

We specify these properties in Computation Tree Logic (CTL). CTL uses path quantifiers $\mathbf{A}$ and $\mathbf{E}$, and temporal operators $\mathbf{G}, \mathbf{F}, \mathbf{X}$, and $\mathbf{U}$. The (state) formula $\mathbf{A}\phi$ holds in a state if ϕ holds on all paths starting in that state, while $\mathbf{E}\phi$ means that ϕ holds on some path. The (path) formulae $\mathbf{G}\phi, \mathbf{F}\phi$ and $\mathbf{X}$ hold for a path, if ϕ holds globally in all states, eventually in some state, or in the next state of the path, respectively. The (path) formula $\phi\mathbf{U}\psi$ holds if until a state occurs along the path that satisfies ψ, property ϕ holds. We also use the *weak until* $\phi\mathbf{W}\psi$. It differs from the *until* in that ϕ until ψ holds, **or** ϕ holds globally along the path. The *weak until* operator does not require that ψ holds eventually along the path, as long as ϕ holds everywhere. The *weak until* can be expressed in terms of the other operators. In CTL a path quantifier is always paired with a temporal operator. For a formal definition of the CTL syntax and semantics we refer the reader to [8].

Checking for uninitialised variables, for expamle, can be expressed in CTL as:

$$\mathbf{AG}\ decl_x \Rightarrow (\mathbf{A}\ \neg read_x\ \mathbf{W}\ write_x) \tag{1}$$

This formula prescribes that on all program paths, if a variable x is declared, it must not be read until it has a value assigned or it is not read at all. The latter means that the variable is unused, which is checked separately.

In the same style properties on correct pointer handling, variable usage, or memory allocation and deallocation can be expressed. Moreover, it allows specifying application specific properties to handle programming guidelines, API-specific rules or even hardware/software interface rules for device drivers. For the sake of simplicity, however, we use the example of uninitialised variables throughout this paper.

We now describe formally how to map programs to transition systems labeled with atomic propositions, and how to derive the labels themselves from a program. To construct a model from the program, requires a formal notion of an abstract syntax tree (AST) and a labeled graph (the annotated CFG). We define a labeled graph/tree and an attributed tree as follows:

Definition. A *labeled graph* (L, E, μ_L) over the alphabet Σ_L is a finite and directed graph, where L is a set of nodes, $E \subseteq L \times L$ is an edge relation between the nodes, and $\mu_L : L \to \Sigma_L$ is a node labeling function.

A *labeled tree* is a labeled graph $T = (L, E, \mu_L)$ if it has a single root node $root(T)$ for which we have the following: For each node $l \in L$ there exists exactly one path from the root to the node, i.e. exactly one sequence $l_0, \ldots, l_n$, such that $l_0 = root(T)$, $l_n = l$, and $(l_{i-1}, l_i) \in E$, for $i = 1, \ldots, n$.

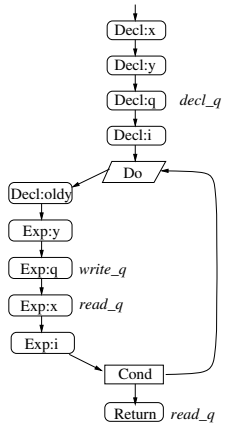
An *attributed tree* (L, E, μ_L, μ_A) over the alphabets Σ_L and Σ_A is a labeled tree where there is an additional labeling function $\mu_A : L \to \Sigma_A$, assigning *attributes* to nodes.

Given two nodes l_1 and l_2 of a labeled tree (L, E, μ_L), we call l_1 the *parent* of l_2 and l_2 the *child* of l_1 if $(l_1, l_2) \in E$. If there exists a (non-empty) path from l_1 to l_2, l_1 is called *ancestor* of l_2, and l_2 is a *descendant* of l_1. ■

An AST can be seen as an attributed tree where the nodes are labeled with program statements and (sub)expressions while the attributes describe the role of a node's branch.

```
1    int fibonacci(int n)
2    {
3       int x = 0, y = 1, q, i = 0;
4       do
5       {
6         int oldy = y;
7         y = x;
8         q = x + oldy;
9         x = q;
10        i++;
11      }
12      while(i < n);
13      return q;
14   }
```

(a)



(b)

Figure 1. (a) the original C program, and (b) a fragment of the annotated control flow graph (CFG).

From the AST we can construct in a straightforward manner the control flow graph (CFG). Note, that a CFG does not contain all the information available in the AST, only the control structure down to the level of statements, but not the structure of expressions, types, and constant values. Later we add labels making it a labeled graph. We denote the labeled CFG of an AST T by CFG_T. The labels show in which node certain atomic propositions hold, for example, if a particular variable is assigned a value, if it is used on the right-hand side of an assignment, or if it is dereferenced.

In our framework, these labels are associated with tree patterns on the AST. We define the syntax of a query language to match tree patterns as follows:

$$\begin{array}{lcl} P & ::= & \epsilon \mid \emptyset \mid \sigma_A \mid \sigma_L \mid \downarrow \mid \downarrow^* \mid P/P \mid P \cup P \mid P[Q] \\ Q & ::= & P \mid label = \sigma_L \mid attr = \sigma_A \mid Q \wedge Q \mid Q \vee Q \end{array}$$

where $\sigma_L \in \Sigma_L$ and $\sigma_A \in \Sigma_A$.

Given an (attributed) tree T and a node l, a pattern defines a set of nodes in the subtree rooted in l. The pattern ϵ defines the node l itself, $\emptyset$ the empty set, and pattern σ_A and σ_L children labeled σ_A or σ_L respectively. The patterns $\downarrow$ and $\downarrow^*$ stand for the children and descendants of l, $/$ and $\cup$ for concatenation and union. Finally, pattern $P[Q]$ filters all nodes satisfying Q.

This tree query language is the downward, recursive fragment of the language defined in [4]. We refer the reader to this paper for formal semantics and a discussion on expressiveness. The only difference is that we allow for two types of labels, which however, does not add expressivity.

Given an atomic proposition p, we associate a tree pattern P with it. We label every node l that matches P, in the AST T with respect to the root node of T, with p. In the case that l is not in CFG_T, we label its closest ancestor in T which is in CFG_T.

Example. Assume an atomic proposition $decl_x$ used to label declarations of variable x that omit initialisation. This proposition is associated with pattern $\downarrow^*$ $(Decl[$*Var:x* $\wedge$ *Nil*$[attr = \texttt{init}]]$, i.e. it matches all nodes (descendants of the root node) in the AST labeled $Decl$, that have a child labeled *Var:x*, and a child labelled *Nil* with attribute *init*. The latter filters declarations that do not include initialisation. □

Once the patterns relevant for matching atomic propositions have been defined and the CFG has been annotated, it is straightforward to translate the annotated graph automatically into a Kripke structure which can be analyzed by a model checker. Adding new checks only requires to define the property to be checked and the patterns representing atomic propositions. All other steps are fully automatic.

Example. Consider the contrived program of Figure 1.a. In the case of uninitialised variable analysis, our approach introduces at most three atomic propositions for each variable. The declaration of variable `q` in line 3 of the code fragment omit the optional declaration. The corresponding node in the CFG will therefore be labelled $decl_q$ (Figure 1.b). Similarly, nodes that correspond to a use and an assignment of variable `q` are labelled $read_q$ and $write_q$, respectively. These labels are identified as patterns on the AST as described in Example 2 for label $decl_q$. Each of these labels will be an atomic proposition in the model checking framework. Model checking of property (1) shows that variable `q` is correctly initialised before use. □

3 Analysing Embedded Assembly

In this section we describe how assembly is embedded in C/C++ code, why it breaks the standard analysis, how we can use readily available interface information to rescue most of our checks, and how we can extend our model-

```
1   int fibonacci_asm(int n)
2   {
3     register int x = 0, y = 1, q, i = 0;
4     do {
5       asm(" mov r1, %[reg_y]           \n"
6           " mov %[reg_y], %[reg_x]     \n"
7           " add %[reg_r], %[reg_x], r1 \n"
8           " mov %[reg_x], %[reg_q]     \n"
9           " add %[reg_i], %[reg_i], #1 \n"
10      :  [reg_x] "+r" (x), [reg_y] "+r" (y),
11        [reg_q] "=r" (q), [reg_i] "+r" (i)            /* out reg     */
12      :  "[reg_x]" (x), "[reg_y]" (y), "[reg_i]" (i)  /* in reg      */
13      :  "r4"                                         /* clobber reg */ );
14    } while(i < n);
15    return q;
16  }
```

Figure 2. Example including assembly

checking based framework to check compliance of embedded code with its interface definition, which, in addition allows us to check arbitrary, often architecture specific, properties of the embedded assembly.

For the remainder of the paper we will use the ARM architecture as our driving example. ARM is a particular RISC architecture most common for embedded systems. Typically, ARM has 16 32-bit processor registers, of which three are special: register 13 is typically the stack pointer, register 14 holds the return address from the current function, and register 15 is the program counter. We implemented the checks we discuss below for the ARMv6 instruction set. However, the general principles of the approach are applicable to most inline assembly programs supported by modern compilers.

3.1 Motivation for Extending the Analysis Framework

Consider the example code presented in Figure 2. It is essentially the same program as the one in Figure 1, the core of the loop has been replaced by an assembly block. Although this example is contrived, in embedded real-time systems where every clock cycle counts, code that is executed frequently may be replaced by more efficient assembly code in a similar fashion. Anything in the `asm()` block is typically not visible by a C/C++ parser and is ignored in standard C/C++ static analysis.

What remains visible is that a variable `q` is declared in the function and its value is returned at the end. However, on the C/C++ level it appears that `q` has never been initialized. Typically, a warning will be raised such as the following by the GNU C compiler:

```
% arm-linux-gcc -Wall -g -c fibonacci.c
In function 'fibonacci_asm':
fibonacci.c:3: warning: 'q' might be used
               uninitialized in this function
```

As we will see later, `q` is in fact initialized before being returned, however, this is done within the `asm()` block. Therefore, the warning is a false alarm. Note, the assembly block also contains a subtle bug involving register `r4`, leading to register corruption. We will explain this in detail later in this section.

3.2 Embedded Assembly Interfaces

As shown in the example in Figure 2 assembly is embedded through compiler extensions into the source code. These compiler extensions define *interfaces* between the variables of the high-level source code and the embedded assembly. In particular, they define mappings between variables in the C/C++ source code and the registers or memory locations used in the assembly. To define interfaces we introduce the following notations: We denote the set of possible variable names in C/C++ by $Vars$ and the set of hardware registers names in ARM by $Regs$. Moreover, for a function f we write $Vars_f$ for the set of variable names occurring in f and $Vars_{reg}$ for the set of variable names occurring in an embedded assembly block. Note, that every variable name $r \in Vars_{reg}$ is implicitly mapped to a processor register by the compiler. In assembly terminology the set of $Regs \cup Vars_{reg}$ are called *operands*. We denote this set by Ops.

Definition. An *assembly interface* is defined by

$$(Output, Input, Clobber)$$

where

- $Output : Ops \rightarrow Vars_f$ is a partial function mapping

registers and register variable names in the assembly to variable names in C/C++ code, and

- $Input : Vars \rightarrow Ops$ is a partial function mapping variable names in C/C++ to register names and register variable names,
- $Clobber \subseteq Reg$ is a subset of hardware register names. ■

The input function passes a variable name to some assembly register or register variable, like a parameter in a call-by-name function call. The output function, on the other hand, is used to pass a register name or register variable from the assembly code back to a C/C++ variable name. The clobber function defines the additional registers that are used inside the assembly block. This tells the compiler that any values in these registers may be changed by the assembly code, and are not conserved across the block. In addition, it tells the compiler that it may use registers that do not appear in the clobber list for other purposes. In particular it may map register variables to any of those registers. This means, if a register name is part of the output, but not clobbered, it can be internally overwritten at any time.

For the GNU C compiler those interfaces have the following syntactic format:

```
asm(<code>
    : <output operands>
    : <input operands>
    : <clobber list>);
```

In the example in Figure 2 the variables `x`, `y`, `q`, and `i` are the output of the corresponding register variables `reg_x`, `reg_y`, `reg_q`, and `reg_i`. The r-flag specifies that a variables maps to a register and additionally indicates whether a variable is write only ("=") or read and write ("+"). Moreover, `x`, `y`, and `i` are also input variables, which are mapped to the correspondingly named register variables. The register `r4` is part of the clobber list and is not guaranteed to be preserved by the assembly block. The compiler can use all other registers for other purposes. The code itself consists of a number of copy instructions and additions and intends to mimic the original C/C++ program from Figure 1.

Next, we define how inline assembly code is checked in two complementary ways: by introducing *operand aliasing* and checking for *interface compliance*. Operand aliasing makes the links between C/C++ variables and the interface explicit in the C/C++ code. Subsequent analysis then takes automatically into account that all inputs to the interface are initialized properly, and assumes that all outputs are written. Note, that operand aliasing relies upon an accurate interface and does not check whether the output operands are actually initialised in the assembler block—this is done by the interface compliance check.

Interface compliance complements operand aliasing. It checks that the assembly code conforms to its interface by examining the assembly instructions themselves. It checks that the input operands are actually used by an assembly instruction, that the output operands are assigned a value by an assembly instruction and that the assembly code only changes the registers in the clobber list.

3.3 Operand Aliasing

The idea of operand aliasing is to create stub C code[1] which mimics the interface. For each operand (input or output) we introduce a local declaration of a fresh auxiliary variable to the C/C++ code. Each fresh variable represents an operand in the interface. For each input operand we initialize the auxiliary variable with the value of the corresponding C-variable as defined through $Input$ and for each output operand according to $Output$ we introduce an assignment mapping it to the corresponding auxiliary variable. This will enable us to check if all inputs are initialized properly and carry over the information that all outputs have at least been modified once.

We denote the set of fresh variables by $Vars_{fresh}$ and define a mapping between the variables in the program to the fresh ones by $aux : Vars_f \rightarrow Vars_{fresh}$. Let $Vars_{out} = \{v \mid \exists r.\ v = Output(r)\}$ and $Vars_{in} = \{v \mid \exists r.\ r = Input(v)\}$ denote the sets of output and input variables for a given interface. Moreover, let $decl(x)$ be a function generating a C declaration for a variable x and $assign(x, y)$ be a function generating a C assignment of the form `x = y`. We create the stub code adhering to the following rules:

- for all $v \in Vars_{in} \cup Var_{out}$ we create $decl(aux(v))$,
- for all $v \in Vars_{in}$ we create $assign(aux(v), v)$, and
- for all $v \in Vars_{out}$ we create $assign(v, aux(v))$.

Example. For the example in Figure 2 operand aliasing inserts the following stub code before the ASM block:

```
int asm_operand_0;
int asm_operand_1;
int asm_operand_2;
int asm_operand_3;

asm_operand_0 = x;
asm_operand_1 = y;
asm_operand_2 = i;
```

and the following after:

[1] Stub code is used to simplify the presentation. To keep our analysis non-intrusive and only modify the AST, not the source code itself.

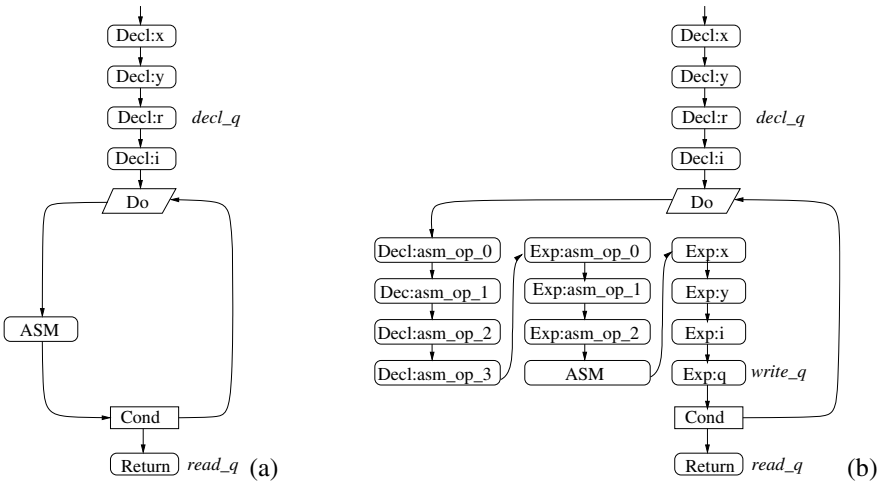


Figure 3. (a) CFG before operand aliasing (b) CFG after operand aliasing

```
x = asm_operand_0;
y = asm_operand_1;
i = asm_operand_2;
q = asm_operand_3;
```

The generated stub code ensures that `q`, as an output operand, is identified as initialized before its value is returned (Figure 3.b). □

3.4 Interface Compliance

While operand aliasing lifts some of the interface behavior to the C/C++ level, it relies on the assumption that the interface is accurately defined and the assembly code adheres to the input, output, and clobber specifications. The compiler sees only the connections between variable names and register names, but it does not check whether the assembly code complies to the interface. By *compliance* we mean:

- all registers mapped to input variables are read
- all input variables that are not output variables are not written
- all output variables are written
- no non-clobber register is written to

We check interface compliance using the framework defined in Section 2 by generating a CFG for the embedded assembly code, and identifying syntactically for every register name and register variable `r` if it writes to or reads from `r`. Assume, $read_r$ or $write_r$ are the two propositions stating that register `r` is read or written, respectively. In addition, we introduce a label asm_{end} to mark the end of the assembly block.

For ARM assembly the read and writes are determined in a straightforward manner using the semantics of the instructions. For example, the `MOV`, `MVN`, `TEQ`, `TST`, `CMP`, and `CLZ` instructions all overwrite the first operand and use the second. Other instructions, such as `MLA` overwrite the first operand while using the second, third and fourth. For a register `r` all nodes are labelled $read_r$ that match the query $\downarrow^* (\texttt{MOV} \cup \cdots \cup \texttt{CLZ})[\textit{Reg:r}[attr = \text{"}2nd\_op\text{"}]] \bigcup \downarrow^* \textit{MLA}[\textit{Reg:r}[attr = \text{"}2nd\_op\text{"} \vee attr = \text{"}3rd\_op\text{"} \vee attr = \text{"}4th\_op\text{"}]]$. Writes to this register are matched by query $\downarrow^* (\texttt{MOV} \cup \ldots \cup \texttt{MLA})[\textit{Reg:r}[attr = \text{"}1st\_op\text{"}]]$. Building and annotating the CFG is obviously platform specific.

Given an annotated CFG for the assembly block we define interface compliance as follows.

Definition. We say an interface is *compliant* if its embedded assembly block satisfies the following checks:

Input I: for all $v \in Vars_{in}$, $r = Input(v)$ check:

$$A\ \neg asm_{end} W\ read_r$$

Input II: for all $v \in Vars_{in} \backslash Var_{out}$, $r = Input(v)$ check:

$$AG\ \neg write_r$$

Output I: for all $v \in Vars_{out} \backslash Var_{in}$, $r = Output(v)$ check:

$$A\ \neg read_r W\ write_r$$

Output II: for all $v \in Vars_{out}$, $r = Output(v)$ check:

$$A\ \neg asm_{end} W\ write_r$$

```
1    int f(int x)
2    {
3        int i, y;
4
5        asm(" mov %[reg_i], #1                    \n"
6            " cmp %[reg_i], %[reg_x]              \n"
7            " bge .end                            \n"
8    ".loop:  add %[reg_i], %[reg_i], #1           \n"
9            " mul %[reg_y], %[reg_x], %[reg_y]    \n"
10           " add %[reg_i], %[reg_i], #1          \n"
11           " cmp %[reg_i], %[reg_x]              \n"
12           " blt .loop                           \n"
13           ".end:  \n" :
14           [reg_y] "+r" (y),
15           [reg_i] "+r" (i):
16           [reg_x] "r" (x)
17           ); return y;
18   }
```

Figure 4. A C program implementing a loop.

Clobber: for all $r \in Regs \backslash Clobber$ check:

$$AG\ \neg write_r$$

The properties for the output operands are similar to the property of the uninitialized use of a variable of C-variables. They differ, since operand aliasing ensures that all $v \in Vars_{out}$ have been declared before the assembly block, and will be used right after. The property ensures that output registers have been written, before the end of the assembler block.

While operand aliasing can be done for most assembly dialects easily, interface compliance requires the parsing of a specific assembly dialect, the analysis of its AST and the construction of the CFG. As such, interface compliance requires a separate parser for every assembly dialect.

Example. Operand aliasing deals successfully with the false alarm that variable `q` was used uninitialised in the example (Figure 2). Operand aliasing relies on the assumption that the assembly code assigns a value to output register `reg_q`. Interface checking shows that this is indeed the case.

Register `r1` is not on the clobber list, although it is used; the clobber list protects register `r4` instead. The clobber property is violated, as `r1` has been written to. This is a potential bug since the compiler could map, for example, register variable `reg_y` to register `r1`, whose value would then be destroyed by the first instruction in the `asm()` block. □

3.5 High-level Language Mapping

General checks of the assembly code can be performed using the same methodology we use on C/C++ code and interface compliance. To achieve this goal we map the assembly code to C/C++ code. This mapping is strictly speaking not to C/C++, but to an equivalent representation in the annotated CFG.

In a first step every construct on assembly level is directly translated into a corresponding C/C++ construct. However, it is possible to recognize that certain combinations of low level language constructs actually implement a more complex high level language construct. Control-flow statements like if-then-else or loop statements have no simple equivalent in assembly code, multiple low level language instructions are required to implement these constructs. In a second step of the mapping from assembly code to C/C++ these are recognized and collated into a single equivalent C/C++ construct. By transforming the low level code to the high level and thereby detecting the loop structure, the subsequent static analysis can include, for example, loop-related checks.

The mapping to a high level language differers in one important aspect from straightforward decompiling to C/C++, in that it also deals with special assembly instructions that have no equivalent in C/C++. Examples are instructions that read or modify the stack pointer, or modify the program counter. Since we are not mapping to actual C/C++ but only to a representation of it in the annotated CFG, we can map these instruction to statements that express enough about their effects to make them meaningful for further analysis.

All queries that have been developed for C/C++ now apply also to the annotated CFG that was generated from the assembly code. These include general checks for unused or dead variables, unreachable code, superfluous assignments, or problematic operations on loop-counter. Moreover, we can introduce architecture specific checks. For instance, we can check for locations where pipeline stalls may occur in ARM code. Such stalls negatively affect performance and may be caused by branch instructions.

```
1    int f(int x)
2    {
3        int i, y;
4
5        i = 1;
6        if(i >= x ) goto end;
7     loop:
8        i++;
9        y = x * y;
10       i++;
11       if(i < x ) goto loop;
12       end:
13       return y;
14   }
```

(a)

```
1    int f(int x)
2    {
3        int i, y;
4
5        for(i = 1; i < x; i++) {
6            i = i + 1;
7            y *= x;
8        }
9
10        return y;
11   }
```

(b)

Figure 5. The program after (a) the first step, and (b) after the second step of mapping it to C

On some architectures, such as MIPS, it can actually be an error to put certain instructions consecutively, since the processor does not contain logic to resolve pipeline errors. Additionally, on the Intel architecture we could check that a disable interrupts instruction is followed by a corresponding enable interrupts instruction.

Example. Figure 4 shows an example of embedded assembly code, and Figure 5 the two steps of mapping it to C. The assembly block uses a few instruction and conditional jumps to implement a for-loop on loop-counter `i`. After the second step of the mapping to C we can apply all checks that have been developed for this language. In certain applications, for example, one might envision a defensive coding rule that the loop-counter should not be modified in the loop. By mapping the assembly to C, we can apply this check also to the block of assembly code, and discover that this rule has been violated. □

4 Implementation and Experiments

4.1 Implementation

The model-checking based analysis framework presented in this paper has been implemented in our analysis tool called *Goanna*. We use the symbolic CTL model checker NuSMV [8] as the analysis engine. Goanna is implemented in OCaml and can be used in standard development environments such as Eclipse.

To integrate assembly into our framework, we internally create new nodes in the AST according to the definitions in Section 3.3 on operand aliasing. For interface checking and general assembly checking we introduce a new ARM entry node in the C/C++ AST and insert the assembly AST there, creating one data structure for analysis.

We currently support almost all of the ARMv6 assembly instruction set. To support different assembly architectures, Goanna requires different parser front-ends, as well as adjustments to the pattern matching queries for the assembly specific checks. While this can entail some non-negligible effort, it would not require us to change our analysis framework as such.

4.2 Experiments

First, we summarize prior evaluation results of Goanna for C/C++ programs, then we present new results for analysis which includes embedded assembly.

An evaluation of our approach to program analysis in general is given in [13], where we analyze OpenSSL, a 260 kLoC[2] software package with an unoptimized prototype of Goanna. For the benefit of the reader, we briefly summarize the results: We found that (a) the analysis time is well within the same order of magnitude as the compile time and that the vast majority of source files of OpenSSL are analyzed in under two seconds, (b) the memory requirements of the analysis fit well in the RAM of current developer machines, and (c) it scales well with an increasing number of properties.

To demonstrate the difference between taking interface information in Goanna into account or not, we first have a look at the example code given in Figure 2. An analysis with assembly checking disabled prints—as expected and similar to other static analyzers—the following warning, which is of course a false positive:

```
% goanna -Wall -g -c fibonacci_asm.c
fibonacci_asm.c:3: Warning: Variable 'q' might
                   be uninitialized
```

The example also nicely shows that we can invoke

[2] LoC = Lines of Code

Goanna exactly like a compiler. Enabling operand aliasing correctly detects that the variable `q` is in fact initialized and no longer reports a warning.

In order to test our additions to Goanna for assembly checking in a more realistic scenario, we chose the L4 microkernel as workload. L4 is a small microkernel that is well suited for—and used in—embedded systems. It is mostly written in C++ (with critical parts in assembly) and has been ported to a number of different architectures, including ARM. More specifically, we analyze the Pistachio 0.4 implementation[3] of L4 when compiling for an ARM SA1100 architecture. In this setup, Goanna analyzes 54 C++ files, two of which have embedded assembly blocks (3.7%). These C++ files use the `#include` directive to include a total of 72 header files, of which 10 have embedded assembly blocks (13.8%).

We first build Pistachio and let Goanna analyze the code with assembly checks disabled, then we build it again with Goanna's operand aliasing enabled and compare the two cases. When enabling operand aliasing, the time to build and analyse Pistachio raises from 75.9 seconds to 77.3 seconds, which is a very small increase of only 1.4 seconds or 1.8%. While not fully implemented yet, interface checking also creates only a minimal overhead.

Comparing the results for Goanna on Pistachio, once without assembly checks, and once with assembly checks, reveals a number false positives that are ruled out by the assembly checks. An example are variables that have been assigned a value, but never used afterwards, i.e. the variable is not read anymore after an assignment[4]. Goanna issues a total of 7 warnings of this property when analysing Pistachio, of which 3 are correct warnings and 4 are false positives due to variables that are only used in assembly blocks after their last assignment. Operand aliasing removes all 4 false positives without creating any additional alarms.

While this is only an indicator of the newly added techniques it demonstrates the value of the solution.

5 Related Work

Model checking has made great advances in the recent years to cover realistic C/C++ programs and led a number of powerful tools [15, 3, 9, 7]. However, they are not yet well-suited for embedded system code containing hardware references and in particular embedded assembly. As shown in [20], these tools either ignore embedded assembly and continue the analysis or abort the analysis all together. The same holds for the most prominent static analysis tools [10, 14, 17, 18] working on C/C++ level.

The picture is different for tools working directly on the assembly level though. Balakrishnan, Reps et al. [2] developed a framework to analyse x86 assembly in the absence of source code and debugging information. The authors create an intermediate presentation of the disassembled object code and use weighted push down systems [19] to model program behavior. The analysis is done using model checking techniques. The advantages of this approach are that embedded assembly can be analysed the same way as the rest of the code, hardware dependencies are taken into account, and compiler specifics are naturally taken care of. On the downside, some structural information available on the C/C++ level is lost, as are type and boundary information, which complicates the analysis [1].

A similar approach relying on less stringent assumptions has been developed by [21]. The authors use disassembled ELF files for hardware-depended program analysis. In contrast to [2] the authors assume the availability of the C program. The analysis framework is based on on-the-fly model checking and can analyse deep semantic properties. This, however, results in very large state spaces and creates the challenge of finding suitable sound abstractions automatically.

At the current stage both approaches above address program analysis on a finer semantic level than we do, but both approaches also have to deal with partial information no longer available at the pure assembly level. It remains to be seen which approach is more practical or more suitable for certain systems. A discussion on the depth of an analysis framework can be found in [12].

Other work on static program analysis problems for assembly code include the following: In [24] the authors follow the disassembling approach and apply abstract interpretation to check for coding conventions such as the absence of hard coded pointers for a specific hardware platform. Yu and Shao developed a type system for the static analysis of concurrent assembly programs applying Hoare style inference for verification [25] and Brylow et al. [6] check for stack sizes and interrupt latencies of Z86 platforms by model checking push-down systems. Those approaches are all on a pure assembly level and as such are less suited for the program analysis of C/C++ source code.

6 Future Work and Conclusions

Summary. In this work we presented an approach to extend static program analysis of C/C++ programs to support embedded ARM assembly code. On the one hand we

[3] `http://l4hq.org/`

[4] Note that this is not strictly a bug in the program, but assigning a value to a variable that is never used indicates a problem in the program's logic, which might well indicate a real bug somewhere else in the program.

took readily available interface information into account, enhancing the precision of the analysis, and on the other hand we demonstrated how to check for the compliance of the embedded code with its interfaces. The overall approach is based on model checking and can be applied to standalone high-level C/C++ code as well as low-level assembly code. The checks have been integrated into our Goanna program analyzer and we presented examples of checking real-life microkernel code.

Current limitations and future work. While at the current prototype stage the Goanna tool is already fast enough for practical use, it still lacks some advanced analysis features. In particular, we are currently not performing any value tracking of variables. This will be important on the C/C++ level for having improved array bounds checking, pointer value approximation, and infeasible path pruning, and on the assembly level for tracking register values. Moreover, while in principle our approach extends to classic interprocedural analysis, we still have to develop heuristics to deal with the combinatorial blow up in order to keep the analysis to a similar speed as it is for intra-procedural analysis. We are in the process of creating a two-pass analysis by making use of summaries which can be generated from the intra-procedural stage.

Acknowledgements. We thank Bernard Blackham, Jörg Brauer, Patrick Jayet, and Michel Lussenburg for their implementation efforts and fruitful discussions.

References

[1] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *8th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI '07)*, LNCS. Springer, 2007.

[2] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. *Verified Software: Theories, Tools, Experiments*, 2007.

[3] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *Intl. Conf. on Computer Aided Verification (CAV '01)*, LNCS, London, UK, 2001. Springer.

[4] M. Benedikt, W. Fan, and G. M. Kuper. Structural Properties of XPath Fragments. In *9th Intl Conf. on Database Theory (ICDT '03)*, LNCS, London, UK, 2002. Springer.

[5] D. Brash. The ARM architecture version 6 (ARMv6). White paper, ARM Ltd, January 2002.

[6] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *23rd Intl. Conf. on Software Engineering (ICSE '01)*, Washington, DC, USA, 2001. IEEE Computer Society.

[7] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate Abstraction with Minimum Predicates. In *Correct Hardware Design and Verification Methods*, LNCS. Springer, 2003.

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Intl. Conf. on Computer-Aided Verification (CAV '02)*, LNCS. Springer, 2002.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, LNCS. Springer, 2004.

[10] Coverity. Prevent for C and C++. http://www.coverity.com.

[11] D. Dams and K. S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Technical Memorandum ITD-04-45263Z, Lucent Technologies, 2004.

[12] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, LNCS. Springer, 2004.

[13] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Intl. Symp. on Theoretical Aspects of Software Engineering TASE '07*, LNCS. Springer, 2007.

[14] Gimpel Software. FlexeLint for C/C++ . http://www.gimpel.com/html/flex.htm.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *34th Symp. on Principles of Programming Languages (POPL '04)*. ACM Press, 2004.

[16] G. Holzmann. Static source code checking for user-defined properties. Pasadena, CA, USA, June 2002.

[17] Klocwork. K7 . http://www.klocwork.com/products/klocworkk7.asp.

[18] Microsoft. Prefast . http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx.

[19] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2), 2005.

[20] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In *Proc. of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*. NASA/CP-2005-212788, Sept 2005.

[21] B. Schlich and S. Kowalewski. An extendable architecture for model checking hardware-specific automotive microcontroller code. In *6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT '07)*, 2007.

[22] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Symp. on Principles of Programming Languages (POPL '98)*. ACM Press, 1998.

[23] D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Intl. Symp. on Static Analysis (SAS '98)*, LNCS, London, UK, 1998. Springer.

[24] R. Venkitaraman and G. Gupta. Static program analysis of embedded executable assembly code. In *Compilers, architecture, and synthesis for embedded systems (CASES '04)*. ACM Press, 2004.

[25] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *9th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP '04)*. ACM Press, 2004.