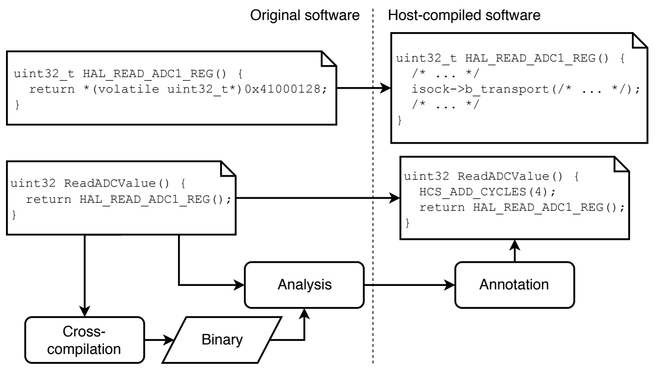
Automated Redirection of Hardware Accesses for Host-Compiled Software Simulation

Rafael Stahl, Daniel Mueller-Gritschneder, Ulf Schlichtmann
Chair of Electronic Design Automation, Technical University of Munich, Germany
{r.stahl,daniel.mueller,ulf.schlichtmann}@tum.de

***Abstract*—For host-compiled software simulation it is required that accesses from the target software to memory-mapped hardware are identified, so that they can be redirected to a virtual prototype. This is straight-forward if the software uses a hardware abstraction layer as interface. If such an interface is not used by existing or third-party source code, the rewriting of the code for host-compiled simulation involves a considerable manual effort. In this paper, we present a method to automate this process with the help of a symbolic execution engine. With our approach the time to adjust software for host-compilation is significantly reduced. We show that the most memory-mapped hardware accesses are correctly rewritten in a real-world application by comparing recorded access traces on a virtual prototype. Additionally, a test suite has been developed to cover edge-cases.**

***Index Terms*—source-level simulation, hardware accesses, static analysis, host-compiled simulation**



```
uint32_t HAL_READ_ADC1_REG() {
  return *(volatile uint32_t*)0x41000128;
}
```

```
uint32_t HAL_READ_ADC1_REG() {
  /* ... */
  isock->b_transport(/* ... */);
  /* ... */
}
```

```
uint32 ReadADCValue() {
  return HAL_READ_ADC1_REG();
}
```

```
uint32 ReadADCValue() {
  HCS_ADD_CYCLES(4);
  return HAL_READ_ADC1_REG();
}
```

Fig. 1. HCS overview

I. Introduction

With the continuous growth of chip complexity the demand for even faster integrated system development is high. Complex systems have to be produced at low cost and a diverse set of applications has to be developed for different products. So considering that software development occupies large parts of the total project time, it is essential that this process can start as early as possible. Therefore it has been the norm to start with software development before the target hardware is available by simulating the software on a Virtual Prototype (VP) of the target platform. This allows early design space exploration and testing.

One way to achieve simulation is to first cross-compile the target software and then running it on an Instruction Set Simulator (ISS). It loads instructions just like the processor would and then simulates the behavior of the target instructions by updating a target processor state representation. This process can achieve very high accuracy, but has the disadvantage of being slow. The low simulation speed makes it unsuitable for simulation of soft real-time and long running applications.

It is possible to trade some accuracy off for much improved simulation speed with host-compiled simulation (HCS) [1], [2]. Instead of interpreting target instructions, the whole target software is compiled directly for the architecture of the simulation host.

This work is partly funded by the German ministry of education and research (BMBF) under grant number 01IS17028F as part of the ITEA3 project COMPACT with reference number 16018.

To enable host-compiled simulation one needs a clearly defined interface in the source code that separates behavioral software code from accesses to memory-mapped hardware [3], [4]. For directly accessing memory-mapped hardware the functions of the Hardware Abstraction Layer (HAL) will contain the direct access operation. When adapting the software for host-compiled simulation, only the HAL implementation needs to redirect to the virtual prototype model of the hardware e.g. written in SystemC/TLM. Figure 1 shows a rough overview of an HCS annotation process where the HAL function implementation was changed from direct memory access to SystemC interaction. However, if an HAL interface does not exist in the first place, it is required to manually rewrite the software. This can be very challenging, because it is not straightforward to pick out the accesses to memory-mapped hardware from other statements. For example an access might occur through a C-struct that contains pointers to both memory-mapped hardware and variables in memory that should not be redirected. The memory might also be accessed through an aliased or offset pointer.

In this paper, a method to automatically locate and rewrite all accesses to memory-mapped hardware in arbitrary source code is presented. For this purpose the symbolic execution engine of the Clang Static Analyzer is utilized [5], [6]. This work significantly reduces the effort to make software compatible with host-compilation. As a side-effect it improves readability. The accesses are replaced with HAL functions that may also be used for other purposes such as tracing of access patterns. The nature of symbolic execution ensures that even when multiple different static addresses are accessed at the same source code location, all possible values can be determined. This makes the presented approach reliable and generally applicable. The method is demonstrated by simulating a rewritten exemplary application with a virtual prototype and comparing the memory access patterns with those recorded on an ISS. The results show that most accesses are found and replaced. To our knowledge few accesses are missing due to limitations in the used symbolic execution engine, which we will tackle in the future. Moreover, extensive test cases have been developed that also aided in testing corner cases.

The remainder of this paper is structured as follows. In Section II an overview of related work in source-to-source transformation is given. Required background is given in Section III. The underlying methodology of this work is presented in Section IV and our implementation is detailed in Section V. Experimental results are presented in Section VI.

II. Related work

Automatic source-to-source transformation has been used heavily in the context of host-compiled simulation for timing annotation [7], [8], [9]. These methods add new source code statements to existing code blocks to associate timing information with it. To retrieve the correct timing information for the target architecture, the software is first cross-compiled for the target and then analyzed on binary level for cycle counts and memory access patterns. The gathered data is mapped back into the source code where it is annotated as instrumentation code. When now compiling and running the software on a simulation host, it will accumulate the target execution time through the instrumentation functions. In these works it is however required that the source code is already using HAL functions for redirecting hardware accesses to a hardware model. Figure 1 illustrated the redirection with the SystemC/TLM interface function `b_transport`, which initiates a transaction on a socket.

The necessity for an HAL interface is outlined in [3], [4] and it also shows how the HAL implementation can be used to switch between native embedded and simulation environment.

III. Background

A. Symbolic Execution Engines

A symbolic execution engine is a static analysis tool for determining whether a certain state can be reached in a piece of software [10], [11]. It attaches symbolic values to variables and then tries to constrain these values as much as possible with the given static code. The software is simulated linearly with each statement, building up a path constraint formula with values that are used. Each time the software branches, the complete simulation state is forked into the possible paths while constraining the variables further. At relevant points of interest the constraint formula can then be solved to determine which input values lead to the current state or whether the program point is reachable at all. The current state of the art to solve these formulas are Satisfiability Modulo Theories (SMT) solvers [12], [13].

B. Clang Static Analyzer

The Clang compiler front-end project provides a free and open-source implementation of a source-code-level symbolic execution engine for C and C++. As part of the LLVM project [14], Clang is backed by a large community that helps mature the static analyzer. Users of the analyzer can write so called checkers that integrate very tightly with the symbolic execution engine itself using callbacks. These callbacks provide high flexibility for usages of the engine, because execution paths and symbolic values can be controlled fully while the engine is running.

It has become especially interesting for the goals of this topic when a contribution to add cross-translation unit (CTU) analysis appeared. This was introduced by Ericsson and ELTE for their static analysis tool ”CodeChecker” that is based on the Clang Static Analyzer [15]. When the engine encounters a function call, it will look for a function definition within its translation unit and inline the body of the called function into the caller in order to more accurately model the function parameters. However, in well-designed modular software a function definition may not always be available in the translation unit where it is called. With the new contribution by them the analyzer will first gather all functions that may take part in linking and then during analysis will resolve, import and inline function bodies across translation units.

IV. Redirecting accesses

The approach to redirecting accesses to memory-mapped hardware starts with taking the original software and running a custom analysis with a symbolic execution engine on it. This analysis has the goal of finding source code expressions that cause read or write operations which statically fall into memory regions mapped to non-memory hardware. A prerequisite is that the full memory map of the system-on-chip platform is known. The results are then used for source-to-source transformation of the original access expression to HAL function calls.

A. Locating accesses

In order to locate accesses to memory-mapped hardware in the target software source code, symbolic execution was chosen to be the most fitting tool, because these accesses do by definition have a statically determinable address. Their path-sensitive nature makes them more suitable for finding accesses to specific memory addresses than other data-flow analysis methods. Listing 1 shows an example where path-sensitivity provides increased accuracy. The access operation at line 5 is clearly only ever referring to `ADC1_PTR` because the conditions `x` and `!x` are exclusive to each other, but common data-flow methods like reaching definition analysis or constant folding are not able to model the constraints in every possible execution path and therefore can only say that `p` is either `ADC1_PTR` or `ADC2_PTR`. Symbolic execution on the other hand will model each execution path with all constraints and when reaching the access operation will find that !x implies that p is always equal to ADC1_PTR, because line 3 was not explored on this path.

Listing 1. Path-sensitivity example
```
1  int *p = ADC1_PTR;
2  if (x)
3    p = ADC2_PTR;
4  if (!x)
5    *p = 0;
```

Exploring every possible execution path has exponential complexity. However, to be useful for their typical usage as a bug finding tool, they have to deliver their findings to the programmer within a short response time. For this purpose or to even make analysis of large software possible, symbolic execution engines implement mechanisms of limiting path exploration at the cost of accuracy. Another way to put an upper bound on execution time is to discard some of the recorded program state. Assuming a perfect implementation of the engine and a full view of the analyzed program, symbolic execution achieves perfect accuracy.

Hardware addresses are not coming from external input to the program, but are hard-coded into the source code. This means that it is possible to use a symbolic execution engine for finding all existing accesses to memory-mapped hardware, as long as accuracy-impairing features are disabled. The resulting long execution time is not a significant issue, because the purpose is not immediate feedback while writing software, but automating a process that might otherwise take weeks or months of manual work. In the embedded systems domain it is also highly unlikely that driver code which interacts with hardware is as complex as web browsers or operating systems which are analyzed in reasonable time already. A solution on source code level is required, because rewriting of hardware accesses to use HAL functions happens on source code level and tracing back from binary or intermediate code is very difficult - especially after optimization.

Finding all expressions that cause read or write operations is achieved by determining which constructs dereference a pointer. There are three ways in which a pointer can be dereferenced in C:

- Explicit dereference: `*p`. This is the trivial case where the symbolic value of p can directly be used as access address. Even if the expression is not directly a named variable as in `*(p + n)`, the full expression should already have a bound value in the engine.
- Member expression: `p->x`. In this case the pointer to struct p is dereferenced on source code level. However, in order to redirect the access to a virtual prototype, the address of p is not sufficient. The correct register can only be referred to when also including the offset of the field x within the struct pointed to by p. This case was resolved by checking if the dereferencing expression is a member expression and then retrieving the statically determinable offset of the field from the compiler front-end. This offset is added to the symbolic value of p to get the final dereferenced memory address.
- Array subscript expression: `p[index]`. Here we also only get the symbolic value of p and not the address of the resulting dereference. This case is a bit more problematic, because the subscript expression can be arbitrarily complex and it has already been processed and discarded by the symbolic execution engine when the dereference of p is found. To solve this issue, symbolic values of array subscript indices have to be added to the program state of the engine whenever they are encountered. These values can now be retrieved when encountering a dereferencing expression that is caused by an array subscript expression. The final dereferenced memory address is set equal to the formula $p + sizeof(pointee\_type) \cdot index$ where p and index are symbolic values.

All methods together ensure that even most complex expressions like `(p + 2)->x[n]->[i]` can be resolved to the final memory address that is accessed. Along with the expression that caused the dereference and the static address value, the access size and type (read or write) is retrieved. This information is required for emitting the correct HAL function call.

Accesses to memory-mapped hardware are not always statically constrained to a single constant address. For example there might be a function taking a base address of a peripheral memory block and this function is called with different base addresses, because there could be multiple instances of that peripheral. In this case the analyzer is able to find that the access happens to either of the resulting addresses. All result values need to be stored with the access expression that caused it.

B. Virtual memory

Software that accesses hardware through mapped virtual memory pose an issue to this work. The virtual addresses that are accessed have a different or even statically unknown value. In Linux for example the kernel API ioremap is used to map a given physical address range to a virtual address that is returned. Determining the return value with the static analyzer is not helpful, because it would not match the sought physical hardware address.

A possible solution is to hook into function calls that interact with virtual memory and change symbolic values. The Clang Static Analyzer allows this. To solve the issue with ioremap, the symbolic value of the return value can be set equal to the symbolic value of the parameter that was passed in as hardware address to map. During analysis this means that the mapping function is treated as empty and every access to the returned virtual address will be recognized as accessing the mapped physical address. Of course it still has to be considered whether bypassing the virtual memory system at this point still accurately simulates the target.

C. Rewriting

After the analysis has been executed on the original source code, the results can be passed to a rewriting tool. This tool goes through all results of each translation unit and replaces them with HAL function calls. In a first step the original translation unit is prepended with an include directive for a support header that defines all code required for calling the HAL functions, notably the function declarations for all available HAL functions. An example of HAL function names would be `HAL_READ_PERIPHERAL_REGISTER()` and `HAL_WRITE_PERIPHERAL_REGISTER(value)`.

The rewriter already has the exact access expression available from the analyzer in order to replace it with the HAL function call. One situation where it is not sufficient to just replace the access expression is on write accesses through assignments. If it was replaced, the resulting expression would assign the write value to the function call, which would not make any sense. The write value has to be available to the HAL function implementation, because this is where it is forwarded to the virtual prototype. Therefore the correct transformation passes the right-hand side expression of the assignment as argument to the HAL write function as shown in Listing 2. The code also shows how compound assignment operators have to be expanded into separate read and write operations.

Listing 2. Rewriting assignments

```
ADC1->REG = 5;
ADC1->REG &= 0x10;
// Rewritten to:
HAL_WRITE_ADC1_REG(5);
HAL_WRITE_ADC1_REG(HAL_READ_ADC1_REG() & (0x10));
```

If the emitted HAL functions have to follow a certain naming convention, there must be a mapping between register addresses and their name. That way the rewriter can look up the correct name with addresses obtained from the analysis.

As explained earlier, there might be accesses that have multiple possible static address values. During the rewriting process it is now possible to emit code that checks whether the original access expression address is equal to one of the result addresses and calls the corresponding HAL function. The access expression is typically not standing by itself as shown in Listing 3 so that it cannot be replaced by an if-statement. Instead, the ternary conditional operator can be used to avoid more complex rewriting. These conditional operators can be chained indefinitely to handle cases with more than two possible values. This example also shows how another branch can be emitted for safer operation. In case the analyzer has missed a possible access value due to limitations in the constraint solver, the program will stop at the exact point of failure and the issue can be resolved with manual intervention. Even when only one result is found, the safer code can be emitted by utilizing the same technique with the conditional operator.

Listing 3. Rewriting dynamic access

```
int x = Base->REG;
// Rewritten to:
int x = (&Base->REG == (void*)0x41000124 ?
  HAL_READ_ADC1_REG() :
  (&Base->REG == (void*)0x42000124 ?
    HAL_READ_ADC2_REG() :
    HAL_ASSERT()));
```

V. Dereference Analyzer

The tool "Dereference Analyzer" was implemented based on the symbolic execution engine of the Clang Static Analyzer. The tool implements a checker and enclosing stand-alone command-line tool that loads the checker as plugin into Clang. It receives as inputs the address ranges that are mapped to hardware, target specific information such as type sizes and the software project to be analyzed. Since the analyzer operates at source code level, the memory map is the only platform-specific input to the analyzer, so that even platforms in early design stages without an LLVM backend can be supported. After running the analysis the tool will return a list of results with the expression that caused the access, the statically determined address that is accessed, the size of the access and whether it is a read or write.

As mentioned in Section III, the Clang Static Analyzer provides various callbacks to hook into the running process of the symbolic execution engine. One of these callbacks is called every time a "location" is encountered. Among other things, such a location can be a memory address. This callback is sufficient to ensure every possible load and store operation is handled. When combined with handling the different dereference cases described in Section IV, many different code constructs can be determined to be accesses to memory-mapped hardware with certain static addresses.

A. Changes to Clang

In the context of this work, some patches to upstream Clang were required. Most importantly this includes the cross-translation unit analysis patches by Ericsson and ELTE [15] with follow up fixes of issues discovered through this work. The following other patches were contributed by us to significantly improve accuracy and results of the analysis:

- Pointer arithmetic on constants: The execution engine did not model pointer arithmetic with constant operands. This was added to correctly model expressions such as `p += 2`.
- More constant bindings: For every expression and declaration the analyzer encounters it should bind any symbolic values to constant values if those values can be determined from the program structure. These changes added more cases where this was possible, namely: casted constants, constant array initialization, constant in-class initializers and constant element-wise struct initialization.
- Extend CTU importing to variables: The existing CTU analysis works by importing function definitions from other translation units into a translation unit that only has the function declaration available. As mentioned in the previous point, it is important that the analyzer binds symbolic values to constant ones whenever possible. A common code pattern in tested code was the constant initialization of a global variable in an own translation unit, while users of that variable only had the declaration available (with the `extern` keyword). Therefore the CTU importing mechanism was extended to not only import function bodies, but also variable initializers.

All required changes are available with the source code and contributed back to Clang, with the goal to make this tool work with an unmodified version. The custom checker can be integrated as a plugin without modifications to Clang. This way any future improvements to Clang will also improve our tool.

B. Tuning execution time

As outlined in Section IV a symbolic execution engine that is used for finding bugs is limited in path exploration to allow faster response times. For the use-case of locating accesses to memory-mapped hardware it is however important to maximize accuracy of the analysis. The Clang Static Analyzer provides several configuration options to tune it appropriately. These are flexible enough that no modification to the analyzer itself are required. Most importantly the maximum number of program state nodes generated from a single function needs to be raised significantly as this is the main bail-out mechanic limiting maximum execution time. Related to this, the maximum function inline depth needs to be raised to support modeling propagation of a pointer argument through a chain of functions, as well as more fine-grained inlining settings that must be relaxed. Generally all functions must be inlined to not lose any information conveyed via function arguments. Another important configuration option is the maximum number of times a block may be visited on the same path, because this causes exploration to stop during loops that do not branch into more execution paths. The analyzer does by default not analyze any functions defined in header files to avoid false positives in library headers, but for catching all possible accesses this option needs to be enabled.

It is possible that tuning of these parameters raise the execution time of the analyzer too much for it to be practical anymore. But since we are not interested in searching for bugs in the given complete software, it is not necessary that it is tested in its entirety. If the software is modular enough, it is very beneficial to only analyze the driver code related parts. More involved algorithmic code like media processing has a strongly negative effect on the analyzer execution time, while the inclusion of it serves no benefit to finding accesses to memory-mapped hardware.

C. Limitations

The main limitation of the analyzer is that it cannot guess where accesses are located if there is no reachable code path from any constant address value to the final access. This is commonly the case when analyzing library code without any code that calls its functions. There can be an `init` function for example that returns a pointer to the initialized device and all other driver functions would take that pointer as argument for accessing the hardware. Without some user code that calls `init` and then the other driver functions, the access operations within them cannot be determined. Therefore when testing libraries, they have to be analyzed along with user code that has sufficient code coverage. Ideally this would be unit test code developed in the process of test-driven development that is wide-spread for writing device drivers [16].

An exception to this issue is the case when driver functions have precondition checks like assertions that verify the value of the pointer before using it. The assert provides the analyzer with sufficient information to perfectly constrain the pointer to the value it was expected to be, since any other value would lead to the assert being triggered and any following code not being reachable with these values.

Since the Clang Static Analyzer is based on the Clang compiler front-end, aside from C code it can also handle C++ code. However, the source code importing component that is crucial for cross translation unit analysis is not mature enough to deal with every code construct, yet. C++ can also contain expressions that cause dereferences which are less obvious than the three cases described above. Some of these cases can not be handled by the analyzer, yet.

One implementation issue that could not be resolved so far is initialization of const-qualified pointers with addresses of const-qualified structs. Listing 4 shows a situation in which this can lead to missing results. The global variable `ADC_SUBHANDLE` is initialized with a pointer to a memory-mapped device `ADC_PTR`. After that another global variable `ADC_HANDLE` is initialized with the address of the first global variable. The Clang Static Analyzer does currently not model this relation, so that the symbolic value of `ADC_HANDLE.sub` becomes unknown. Any access in the software that references the `ADC` device via that handle cannot be resolved to a static address. This is a technical limitation that can be resolved in Clang.

Listing 4. Const pointer initialization

```
struct SubHandle {
  ADC *adc;
};
struct Handle {
  SubHandle *sub;
};
const struct SubHandle ADC_SUBHANDLE = {
  .adc = ADC_PTR
};
const struct Handle ADC_HANDLE = {
  .sub = &ADC_SUBHANDLE
};
```

VI. Results

A. Setup

The functionality of the tool was tested by comparing the original software to the rewritten one while running them both in a simulated target environment as follows:

1) In a first reference run the original software was run on an ISS with a VP that contains a data bus trace as shown in Figure 2. The trace module records all accesses that happen on the data bus as long as they fall within an address range that is occupied by memory-mapped hardware. Accesses to memories are not recorded, because accesses to them are not redirected to a VP for HCS.
2) The software is rewritten to use HAL redirection functions that besides performing the original access operation also report the accessed address to a special VP

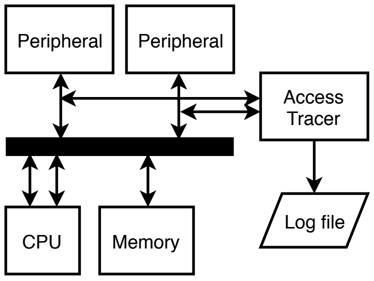


Fig. 2. VP testing setup

trace module. The rewritten software is now simulated to generate a second trace.

While the first trace records all hardware addresses accessed by the original software, the second one only records accesses that happen within HAL redirection functions from the transformed software. This ensures that accesses which would have been missed in the rewriting process do not appear in the second trace file. When finally comparing the two trace files, it is expected that they are equal.

B. Experiments

The primary testing project is an application that controls a robotic arm while running on an Infineon XMC4500 industrial micro-controller. Most of the contained driver source code was automatically generated by Infineon's DAVE^{TM} tool and amounts to 39254 total lines of code with 24896 in headers. The project is ideal for testing, because the software makes use of many peripherals that are connected to the CPU with memory-mapped bus accesses. All of the used peripherals are implemented in a SystemC/TLM based virtual prototype, so that the software can operate properly. As ISS for the simulation ETISS was used [17].

The analysis took 137.2 seconds and identified 300 accesses to memory-mapped hardware, with their concrete expression in the source code together with one static address. The simulation of the original software resulted in about 50k recorded accesses. The first 36050 recorded accesses of the transformed software match the original one, after which the implementation issue described in Section V-C interferes with the results. Manual inspection showed that a global variable was assigned to as in Listing 4.

Listing 6 illustrates the source code of a rewritten function. Even though the original function shown in Listing 5 only contains two expressions that referred to a register, in total this function contains eight accesses, resulting from two expressions with four possible static addresses each. This is visible through the cascading ternary conditional operators that each of the two expressions can be equal to four different static addresses at run-time. This is also evident by the unchanged switch statement earlier in the function. Note that the HAL function names have been kept simple during testing. Their only purpose is to record the accessed address passed through their first parameter and perform the original operation. This makes the control code look pointless, because it is equivalent to `HAL_READ_32(originalExpression)`. However, a real-world HAL interface would have functions with signatures specific to each register or field, e.g. `HAL_READ_ADC1_REG()`.

Listing 5. Original function
```
int adc0_groupX_channelY_enable(uint32_t X,
                                uint32_t Y) {
  VADC_G_Registers* VDAC0_GroupX_REG;

  switch (X) {
  case 0:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004400U;
    break;
  case 1:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004800U;
    break;
  case 2:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004C00U;
    break;
  case 3:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40005000U;
    break;
  default:
    return -1;
  }

  uint32_t ASSELreg = VDAC0_GroupX_REG->ASSEL;
  ASSELreg |= (1 << Y);
  VDAC0_GroupX_REG->ASSEL = ASSELreg;
  return 0;
}
```

Listing 6. Rewritten function
```
int adc0_groupX_channelY_enable(uint32_t X,
                                uint32_t Y) {
  VADC_G_Registers* VDAC0_GroupX_REG;

  switch (X) {
  case 0:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004400U;
    break;
  case 1:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004800U;
    break;
  case 2:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40004C00U;
    break;
  case 3:
    VDAC0_GroupX_REG = (VADC_G_Registers*) 0x40005000U;
    break;
  default:
    return -1;
  }

  uint32_t ASSELreg =
    (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004528 ?
      HAL_READ_32(0x40004528) :
    (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004928 ?
      HAL_READ_32(0x40004928) :
    (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004d28 ?
      HAL_READ_32(0x40004d28) :
    (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40005128 ?
      HAL_READ_32(0x40005128) :
      HAL_ASSERT())))));
  ASSELreg |= (1 << Y);
  (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004528 ?
    HAL_WRITE_32(0x40004528, ASSELreg) :
  (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004928 ?
    HAL_WRITE_32(0x40004928, ASSELreg) :
  (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40004d28 ?
    HAL_WRITE_32(0x40004d28, ASSELreg) :
  (&(VDAC0_GroupX_REG->ASSEL) == (void*)0x40005128 ?
    HAL_WRITE_32(0x40005128, ASSELreg) :
    HAL_ASSERT())))));
  return 0;
}
```

The execution time of the Dereference Analyzer was also tested with the publicly available DAVE^{TM} example project `BCCU_WHITE_LED_LAMP_XMC12`. Is has the scope of 69676 lines of code with 56806 in headers. The project was processed in about 63.1 seconds and raising the limits did not significantly change the execution time, indicating that the analyzer was already exploring all possible paths. Since this project uses more peripherals than are available in our virtual platform, the rewritten code could not be executed. The analyzer identified 1190 accesses for this project. Even though both projects have about the same number of lines of code in their `.c` files, the higher number of results can be explained, because this projects driver functions contain many assertions on their arguments. As explained earlier this allows the analyzer to constrain unknown pointers to static values without the function having to be called.

C. Test cases

Several test cases were developed to determine corner cases of the analyzers capabilities. A test runner application takes a directory of small test applications and runs the Dereference Analyzer on them. The actual results are compared to expected ones, described by address value, access size and access type (read or write). If there is any mismatch, the test runner stops and presents the failing difference.

The most simple test applications contain the basic dereference expressions with pointer arithmetic, struct member accesses, array accesses and combinations for them. Some deal with ensuring the symbolic execution engine handles constraints properly with asserts or explicit range checks. Others test properties of the results like the access size or type. A large part of the tests are created for reproducing specific bugs that were encountered while testing larger code bases. Many of these are dedicated to issues while importing declarations from other translation units, but also include pointer arithmetic issues, missing initialization values assigned to global variables and results with multiple possible addresses. All tests except for the implementation issue in Section V-C have had their underlying issue fixed and are passing.

VII. Conclusion

In this paper, a method to automatically locate and rewrite accesses to memory-mapped hardware in driver software was introduced. It aids in the process of making software compatible with host-compiled simulation. This method should able to find all accesses with the help of path-sensitive symbolic execution. Our tests with an exemplary test application running in a full simulation environment of ISS and VP can show that this is possible once the implementation is able to model the source code better. Various specialized unit tests demonstrate reliable functionality with difficult constructs.

For future work, the modelling inaccuracy of the analyzer should be fixed so that all accesses can be found. Futhermore, liveness analysis can be applied to the values read from and written to in order to determine which specific bit-field is accessed within a register. The improved accuracy could for example be leveraged by fault-injection analysis. By basing this work on the Clang Static Analyzer, the results will improve as the analyzer matures. For example the cross-translation unit analysis is only an experimental feature right now and the constraint solver is fairly simplistic.

Acknowledgment

This work is partly funded by the German ministry of education and research (BMBF) under grant number 01IS17028F as part of the ITEA3 project COMPACT with reference number 16018.

References

[1] V. Živojnovic and H. Meyr, "Compiled hw/sw co-simulation," in *Proceedings of the 33rd annual Design Automation Conference*. ACM, 1996, pp. 690–695.

[2] D. Mueller-Gritschneder and A. Gerstlauer, "Host-compiled simulation," *Handbook of Hardware/Software Codesign*, pp. 1–27, 2017.

[3] P. Gerin, X. Guérin, and F. Pétrot, "Efficient implementation of native software simulation for mpsoc," in *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 2008, pp. 676–681.

[4] P. Gerin, M. M. Hamayun, and F. Pétrot, "Native mpsoc co-simulation environment for software performance estimation," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2009, pp. 403–412.

[5] T. Kremenek, "Finding software bugs with the clang static analyzer," *Apple Inc*, 2008.

[6] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2010, pp. 535–548.

[7] A. Gerstlauer, "Host-compiled simulation of multi-core platforms," in *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*. IEEE, 2010, pp. 1–6.

[8] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 290–295.

[9] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann, "Control-flow-driven source level timing annotation for embedded software models on transaction level," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 600–607.

[10] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[11] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[12] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli *et al.*, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[13] C. Barrett, D. Kroening, and T. Melham, *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*, ser. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, 6 2014.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[15] D. Krupp, G. Horvath, Z. Porkolab, and P. Szecsi, "Cross translational unit analysis in clang static analyzer: Prototype and measurements," *EuroLLVM*, 2017.

[16] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 34–45.

[17] D. Mueller-Gritschneder, K. Devarajegowda, M. Dittrich, W. Ecker, M. Greim, and U. Schlichtmann, "The extendable translating instruction set simulator (etiss) interlinked with an mda framework for fast risc prototyping," in *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. ACM, 2017, pp. 79–84.