2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Intelligent Virtual Platforms

Ola Dahl, ola.dahl@ericsson.com
Mikael Eriksson, mikael.k.eriksson@ericsson.com
Udayan Prabir Sinha, udayan.prabir.sinha@ericsson.com
Daniel Haverås, daniel.haveras@ericsson.com

Ericsson AB, Stockholm, Sweden

***Abstract*—Ericsson develops SystemC/TLM-based virtual platforms for digital ASICs and boards. The virtual platforms are used to accelerate software development in early project phases prior to board bring-up, but also in the continued development, for software regression testing. In this paper, we give an overview of recent developments where we add dynamic analysis functionality to our virtual platforms, for monitoring the actual software execution, and for detecting problems in software. We have built prototypes for instrumentation and monitoring functionality that can detect problems with uninitialized data in software. We have also investigated how to detect problems with concurrency in software, such as data races that occur when more than one software process access shared data without synchronization. We think that such monitoring functionality can enable new use cases for virtual platforms, with diagnostics and error reports for detection and pinpointing of errors in software. Long term, this should lead to higher product software quality.**

Keywords—virtual platforms, dynamic analysis, memory management errors, uninitialized memory, concurrency errors, software quality

I. INTRODUCTION

The functionality of our virtual platform SVP has been expanded to include detection of uninitialized memory usage, and detection of concurrency errors. For this work, we have taken inspiration from available tools. These tools, such as Valgrind, MemorySanitizer, and ThreadSanitizer, work well for host applications e.g. in Linux based systems. We have implemented features that are present in these tools, such as shadow memory and propagation of uninitialized values through the execution of target instructions, but within the virtual platform itself. The result is an instrumented virtual platform which executes, and monitors, software, in a non-intrusive way. In this work, we have adapted the features to our architecture, where dedicated processors run software using a real-time operating system.

Valgrind [1] provides a variety of tools for debugging and profiling of target programs. Clang [4] is a compiler front-end for various programming languages including C and C++. Both provide comprehensive tool suites for a variety of purposes, including automated memory management error detection, and are widely used by programmers. MemorySanitizer [5] and AddressSanitizer [2] are tools provided by Clang for detecting memory management errors. MemorySanitizer is used to detect usage of uninitialized memory while AddressSanitizer [6] can detect errors related to accessibility of a memory address such as out-of-bounds and out-of-scope accesses, memory leaks (if LeakSanitizer is enabled) and erroneous use of memory management APIs (such as *malloc()*, *calloc()* and *free()*). Valgrind provides Memcheck which combines the functionality provided by AddressSanitizer (including LeakSanitizer) and MemorySanitizer into one single tool.

As an example, consider a program as

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int x;
    printf("x = %d\n", x);
    return 0;
}
```

Running Valgrind on this program results in an error report, as

```
Use of uninitialised value of size 8
   at 0x3EAAC4390B: _itoa_word (in /lib64/libc-2.12.so)
   by 0x3EAAC464C2: vfprintf (in /lib64/libc-2.12.so)
   by 0x3EAAC4EFF9: printf (in /lib64/libc-2.12.so)
   by 0x4004F8: main (test.c:6)

Conditional jump or move depends on uninitialised value(s)
   at 0x3EAAC43915: _itoa_word (in /lib64/libc-2.12.so)
   by 0x3EAAC464C2: vfprintf (in /lib64/libc-2.12.so)
   by 0x3EAAC4EFF9: printf (in /lib64/libc-2.12.so)
   by 0x4004F8: main (test.c:6)

Conditional jump or move depends on uninitialised value(s)
   at 0x3EAAC44F53: vfprintf (in /lib64/libc-2.12.so)
   by 0x3EAAC4EFF9: printf (in /lib64/libc-2.12.so)
   by 0x4004F8: main (test.c:6)

Conditional jump or move depends on uninitialised value(s)
   at 0x3EAAC44F71: vfprintf (in /lib64/libc-2.12.so)
   by 0x3EAAC4EFF9: printf (in /lib64/libc-2.12.so)
   by 0x4004F8: main (test.c:6)
```

We see, in the error report, that the usage of uninitialized memory was detected in the printf statement. This illustrates a central idea, in Valgrind as well as in our tools, that usage of uninitialized memory should only be reported when it has externally visible consequences. This strategy is used to prevent false positives. Valgrind uses shadow memory, and propagation of uninitialized values through instructions. See e.g. [1] for additional information regarding the technical aspects of these features.

Concurrency analysis for a host-based Linux program can be performed using ThreadSanitizer [8]. Also in this area, we have taken inspiration from host-based Linux tools, and implemented the strategies used in our virtual platform. For concurrency detection, a detection of happens-before relations [7] can be used. When doing this, tracking of memory accesses is done, and a shadow memory representation is used. Checking the happens-before relation for every memory access event is very computationally expensive if full vector clocks are compared. It can be noted that happens-before detection has the advantage that every race reported is real (i.e. no false positives occur). An optimization was introduced in FastTrack [9], where each event only stores its scalar clock and thread ID as a tuple known as an epoch. It was shown that it is sufficient to compare an epoch with a new vector clock for write-write and write-read data races. FastTrack only guarantees detecting the first data race of a variable, which is often sufficient. In ThreadSanitizer v2 [8], most read-write data races can also be detected if a fixed number of epochs are stored for each memory address (an epoch is stored for both writes and reads, and up to 4 epochs are stored for a 64-bit address range). It has a slightly larger risk of missing a data race if there are many accesses to the same address, but has the advantage of having a fixed memory overhead and a ceiling on computational complexity.

II. APPLICATION

The Ericsson virtual platform, for baseband and digital radio processing, has been instrumented for detection of uninitialized memory usage and concurrency detection. The virtual platform simulates hardware used in Ericsson Radio System products, as illustrated in Figure 1. We are developing and maintaining virtual platforms for a variety of baseband ASICs and radio ASICs. We also develop and maintain board-level simulators where we add models for a selection of components available on the respective boards. The specific hardware architecture is illustrated in the left part of Figure 2. The DSP architecture, as shown in the left part of Figure 2, is a Harvard architecture with local memories for program and data. It is a VLIW architecture, with a CISC instruction set. The DSPs have several execution units that execute operations in parallel. The DSP architecture is simulated in our virtual platform using SystemC and TLM, in combination with a QEMU-based instruction set simulator. This is illustrated in the right part of Figure 2. The virtual platform is invoked through a debugger (fladb, for FlexASIC Debugger), which is also used for debugging the real target HW. It is also possible to further connect SVP to a special version of GDB, referred to as EMCA-GDB.

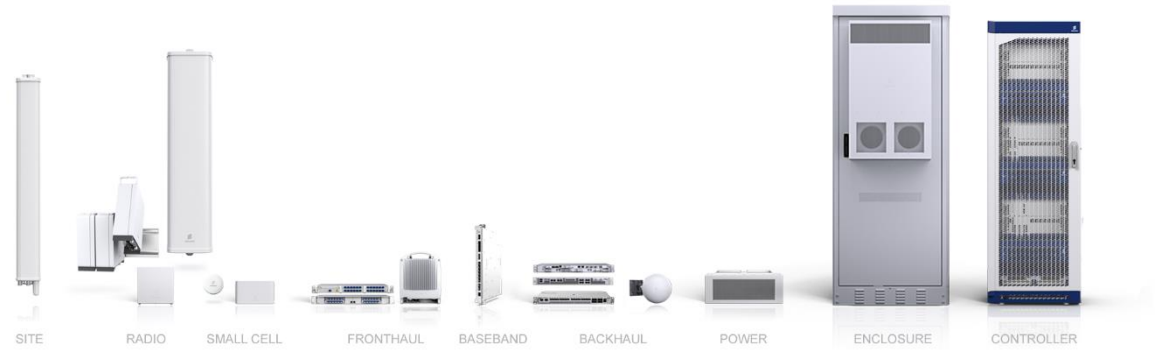


Figure 1 - Ericsson Radio System products.

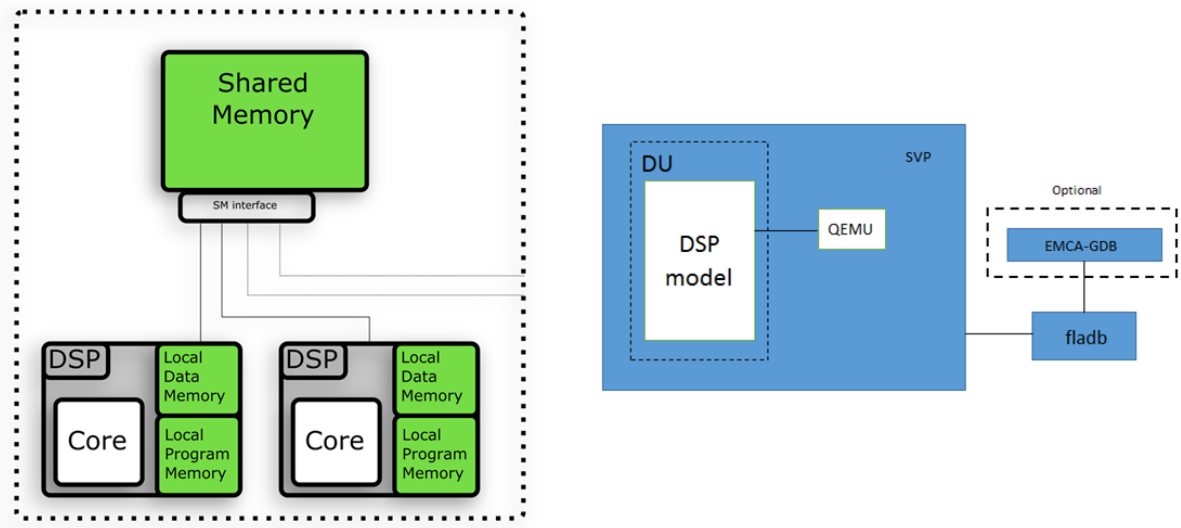


Figure 2 - Ericsson DSP architecture (left) and Virtual platform architecture for DSP simulation (right).

Implementing detection of uninitialized memory in a virtual platform requires the following:

1. *Execution of target code on a simulated hardware platform*: a virtual platform provides a simulated hardware environment for target code to be executed on, and the models within the virtual platform can be exploited for information about the hardware (such as accumulators, registers, and memories). This is needed to correctly propagate information about uninitialized memory for each instruction.
2. *Knowledge of target program source code:* This includes names (and addresses) of variables and functions being used in the program. If a program is compiled with debugging symbols, the line numbers in the source file where the error occurred can be reported to the user. This information can be accessed from debuggers (in this case, fladb and EMCA-GDB) connected to the virtual platform. This is needed to report detected errors to the user in an understandable way, e.g. via the source file name and line number in the file where the error occurred, along with other useful details such as the name of the uninitialized variable.
3. *Knowledge of the operating system*: This includes context-switching information, location of global variables and stack variables, heap allocations/deallocations and information about system calls. The operating system already has this information, and since the virtual platform is executing the operating system kernel, it is possible to use this information when detecting usage of uninitialized memory. The information retrieved in this way can also be used to track heap memory leakage.
4. *Detection, and reporting, of uninitialized memory usage:* The actual error detection algorithms and associated framework (such as shadow memory).

For detection of concurrency problems, we have implemented an algorithm, similar to FastTrack and Google's ThreadSanitizer v2. The algorithm detects synchronization by checking program counter and tracking memory accesses. It uses a vector clock library, a TLM transaction monitor, and shadow memory. It provides EMCA-GDB back-traces, using a report formatter.

III. IMPLEMENTATION

A. *Memory Management Error Detection*

For each memory address, and for each register, we use validity bits (V-bits) for indication of validity. For each memory address, we use an accessibility bit (A-bit) for indication of accessibility. When software runs, V-bits and A-bits are updated. This is done, for each instruction executed, so that validity and accessibility are propagated. The propagation continues until an instruction performs actions, based on invalidity or inaccessibility, that are externally visible. When this happens, an error report is issued. During system boot, we initialize the V-bits and the A-bits, based on information from actions taken by software during the boot. In this way, we arrive at a situation where the states of the V-bits and the A-bits correspond to memory segments that should be regarded as valid, and accessible, before the actual software application starts. The initialization of V-bits and A-bits is done incrementally, during the boot, and we do propagation of V-bits and A-bits as well, in this stage, for the purpose of detecting errors also in boot code.

Our approach for propagating invalidity and inaccessibility is similar to the approaches taken in Valgrind Memcheck [1] and MemorySanitizer [5]. In some areas, however, we believe that we have taken a somewhat different approach. One such area is handling of flags, where we track individual bits in the flag registers. We have taken this approach in order to avoid false positives. There are several flag registers, and multiple criteria for updating the flags. In addition, the flag updates are controlled by logic related to execution of several instructions in parallel. Another area where we believe we have a somewhat more detailed approach than Valgrind Memcheck and MemorySanitizer is V-bit propagation in arithmetic and logic instructions. Here, we have chosen to propagate V-bits in a way that takes as much as possible into account of the actual operation performed. As an example, for an addition, MemorySanitizer approximates the V-bit propagation by combining all uninitialized bits of both operands with a logic OR, and ignores carry [12]. In contrast, we propagate V-bits based on an algorithm that does carry propagation on each bit-pair of the operands.

The software application typically runs in several processes and on several processors. The propagation of V-bits and A-bits, and the associated error reporting, is done on all processes and on all processors. We propagate V-bits when messages are sent between processes executing on different processors. We also monitor process switches. When a process switch is detected, and the process to be resumed has not executed before, its stack is initialized, with respect to V-bits and A-bits. In addition, a selection of system calls, such as sending of messages between processes, are detected.

Our real-time operating system has the concept of kernel space and application space. The boundary between them, however, is not as distinct as for larger operating systems, such as Linux. From a dynamic analysis viewpoint, this makes it possible to do propagation of V-bits and A-bits also inside a system call. In this way, we do not have to detect all system calls. For several system calls, we can instead continue the dynamic analysis, which in turn makes it possible to detect errors also inside the operating system.

B. *Concurrency Error Detection*

Our implementation is based on an extended happens-before algorithm, where a pure happens-before detection is extended with monitoring of memory unlock/lock arcs [8]. For every memory access, metadata about the access is stored in a struct called an *access block entry*, with the following contents: *PID (process ID)*, an unsigned 32-bit integer used to identify the source of the access; *is_write*, an 8-bit integer with value 1 for write events and value 0

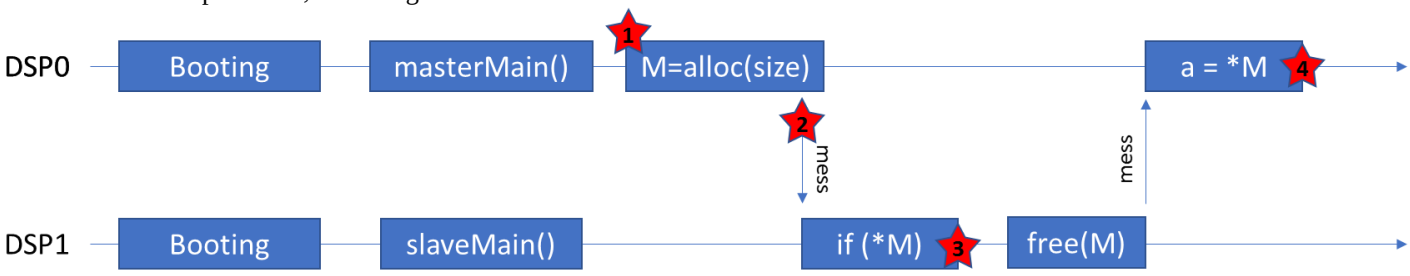
for read events; *clock*, a 64-bit unsigned integer storing the epoch of the accessing process. For a process P, the epoch is retrieved from P's vector clock.

This struct is nearly identical to the ones used in ThreadSanitizer v2, with only the bit widths of the fields being different. For every memory address, we map an access block holding up to N access block entries. The access blocks are stored in an array with one element per memory address. For each access block, the memory address is used as the access block's index in the array. If a memory access happens at an address that already has N populated access block entries, an entry that happened-before the current entry will be evicted. If no such entry exists, a random block is evicted. The number N of access block entries per address can be configured. A higher value should theoretically decrease the risk of missing a race when an unsynchronized access is preceded by a large number of synchronized accesses from the same thread, as the preceding offender would be evicted in such a case. This comes at the cost of greater memory overhead.

Although using the same basic algorithm as in ThreadSanitizer [8], there are some differences. As an example, our algorithm for concurrency detection detects data races in a shared physical address space in a many-core system. This is different from ThreadSanitizer [8], which detects data races within a virtual address space that belongs to a single process. Another difference is that we do detection during run-time, on an unmodified executable program, while ThreadSanitizer adds instrumentation code during compilation. As for the case of detection of memory management errors, we rely on software awareness, in the sense that we need to detect the execution of certain system calls. For the case of concurrency detection, we detect usage of semaphores, and sending and receiving of messages between processes.

IV. RESULTS

A scenario where memory management errors are detected is shown in Figure 3. The scenario illustrates two software processes, executing on two DSPs.




```
1
**************************************************************
Detected invalidity: EOS_alloc parameter "size" @a0
                     Invalid bits: 0xffff
Back Trace:
#0 0x175cd doAlloc () at utils_swu.c:57
#1 0x17486 in masterMain () at master.c:95
#2 0x1035d in EOS_getPid () at eos_process.h:469
**************************************************************

3
**************************************************************
Detected invalidity: Condition (accumulator flags @a3)
                     Invalid bits: 0x1
Back Trace:
#0 0x17505 doExternallyVisibleAction () at slave.c:40
#1 0x1753a in slaveMain () at slave.c:58
#2 0x1035d in EOS_getPid () at eos_process.h:469
**************************************************************

2
**************************************************************
Detected invalidity: Message header "receiver" field
                     Invalid bits: 0xff
Back Trace:
#0 0xa318 EOS_sendMessa  () at eos_core_service_mix.asm:174
#1 0xab45 EOS_sendSignal () at eos_core_service_basic.asm:209
#2 0x174fc in masterMain () at master.c:144
#3 0x1035d in EOS_getPid () at eos_process.h:469
**************************************************************

4
**************************************************************
Detected inaccessibility: Mem location @M=0x1f9aa-0x1f9b3
Back Trace:
#0 0x174a6 in masterMain () at master.c:111
#1 0x1035d in EOS_getPid () at eos_process.h:469
**************************************************************
```

Figure 3 - Detection of memory management problems, such as uninitialized memory, non-accessible memory, and illegal parameters to system calls.

At detection point 1 (the red star with 1), an erroneous memory allocation is performed. The error can be seen in the error print, where an uninitialized value for the size is reported. A message is sent, at detection point 2. The message is sent to a receiver that is not defined (the parameter for receiver is not initialized). As can be seen, this error is also reported. Detection point 3 illustrates how uninitialized data is used in a conditional branch. The detection mechanism has detected that the pointer M, which is dereferenced, is not initialized. In detection point 4, access to invalid memory is detected. The software tries to address a memory which is not allocated, and this error is reported. A source code example, corresponding to the scenario shown in Figure 3, is shown in Figure 4.

```
EOS_PROCESS(MASTER)
{
  U16 size;
  U16 send_pid = EOS_find("SLAVE", 1);
  __sm U16* smp = EOS_allocSm(size * sizeof(U16));
  EOS_sendSM(smp, size, send_pid);
  EOS_receive(EOS_find("SLAVE", 1), &smp, &size);
  printf("smp[1]=%d\n", smp[1]);
}

EOS_PROCESS(SLAVE)
{
  __sm void* smp;
  U16 size;
  EOS_receiveSM(EOS_find("MASTER", 0), &smp, &size);
  U16 tmp = smp[0] ? 1 : 0;
  EOS_freeSmMem((__sm void**)&smp);
  EOS_sendSM(smp, size, EOS_find("MASTER", 0));
}
```

Figure 4 – Example program with memory management errors.

As an example of a bug that was found in production code using memory management error detection, consider a code example as

```
U32 offset = 0;
do {
  U32 block_size = (size > MAX_BLOCK_SIZE) ? MAX_BLOCK_SIZE : size;
  EOS_BlockCopy(dst_p + offset, src_p + offset, block_size);
  offset += block_size;
} while(offset < data_size);
```

This code copies data from shared memory to processor local memory. It does so in chunks of size MAX_BLOCK_SIZE if size (holding the number of words to copy) is larger than MAX_BLOCK_SIZE. In this case, if size is not an even multiple of MAX_BLOCK_SIZE, too much data will be copied (since for the last copy, more data than what is left to copy is copied). The report from our memory management error detection, when detecting this, was

```
Report: DSP=0: PID=65535: Detected inaccessibility in background: SM access @PCE=0x977b
@SM=0xf2e7d-0xf2e8f
Reason:  SM is accessed at a location where there is neither a global variable, nor a heap
allocation.
```

We have also found out-of-bounds errors, for example in assembly code, where a corresponding C code would be

```
for (U16 i = 0; i <= ARRAY_SIZE; i++) {
  value = array[i];
  if (value == requested_value) break;
}
```

In this example, if the value is not found, we will index outside of the array bounds. The report when detecting this was

```
Report: DSP=0: PID=65535: Detected invalidity: Condition (accumulator flags) @PCE=0x8cde (invalid
bits: 0x400)
Reason: An uninitialized value is used to make a decision on which execution path to take.
```

which indicates that we have come to the case where i==ARRAY_SIZE, and since array[ARRAY_SIZE] is uninitialized, and used to take a decision (if we shall continue the loop), an error is reported.

We have measured the overhead when using memory management error detection. Our initial findings point towards an overhead of a factor 1.5-2 with respect to execution time, and a 15-20 percent overhead with respect to memory usage. We can note that the overhead, both for execution time and memory usage, is an overhead for the host on which the virtual platform runs. The target software, i.e. the software for which we want to find memory management errors, is not influenced by this overhead. Its timing will not be affected, since the simulated time is not affected, and it will not see any intrusions on its memory usage, since the error detection runs completely in the host memory space.

Concurrency Analysis

For concurrency analysis, we can detect certain kinds of concurrency errors. It is possible e.g. to detect if a memory area is accessed concurrently without using synchronization, such as semaphores. A simple code example for concurrency analysis, with two processes that can be run on EMCA DSPs, is shown in Figure 5. In the code example in Figure 5, process p1 is performing unsynchronized write accesses to shared memory after the mutex has been unlocked. This is a data race which can be detected through concurrency analysis. An example of concurrency error detection is shown in Figure 6. As can be seen in Figure 6, the error is reported, and a reference to the corresponding source code line is given.

```
#include <eos.h>
EOS_PROCESS(p1)
{
    //setup for message passing
    MSG busmsg;
    U16 selAll[]    =   {0};

    //allocate shared memory, retrieve PID of p2
    __sm U32* value  = EOS_allocSm(4);
    PID p2pid = EOS_find((CHAR*)"p2", 0);
    //send memory address to p2
    EOS_sendSm(1, 3, (__sm void*)value,4,0,p2pid);

    EOS_mtxLock(1);
    //this access is protected by the mutex
    *value = 0x0;
    EOS_mtxUnlock(1);

    //unsynchronized access outside of the mutex
    *value = 0xFFFFFFFF;
    //wait for p2 to finish
    EOS_receive(selAll, &busmsg, NULL, NULL);
}
```

```
#include <eos.h>
EOS_PROCESS(p2)
{
    //setup for message passing
    MSG busmsg;
    U16 selAll[]    =   {0};

    __sm void* value;

    //get shared memory address from p1
    EOS_receive(selAll,&busmsg,&value,NULL);

    EOS_mtxLock(1);
    *(__sm U32*)value = 0x1;
    EOS_mtxUnlock(1);

    PID p1 = EOS_getSender(&busmsg);

    EOS_send(1, 0, &busmsg, p1);
    EOS_yield(); //finish p2

}
```

Figure 5 - Data race between two processes.

```
ca_report: DATA RACE DETECTED: previous p2 write at 0x1c3ab
conflicts with current p1 write. Colliding byte addresses:
0x1c3ab
0x1c3ac
0x1c3ad
0x1c3ae
Back Trace:
0x0000000000017598 in p1 () at p1.c:53
```

Figure 6 - Detection of concurrency error.

For concurrency analysis, initial tests show a relative slowdown of roughly 5x in memory intensive applications. The slowdown may increase for more complex applications with a greater number of parallel processes.

CONCLUSIONS

Our virtual platform SVP has been instrumented for detection of software errors. We have an implementation where it is possible to detect memory management errors, such as usage of uninitialized memory and access to non-accessible memory [11]. We can also detect uninitialized parameters to system calls. This version of the virtual platform is now being developed for production usage. In addition, we have a prototype implementation for concurrency detection [10]. We can detect concurrency errors, such as unsynchronized accesses to shared memory.

The implementations have been done by instrumenting an available virtual platform, such that software can run unmodified. Concepts and ideas from available tools, such as Valgrind, MemorySanitizer, and ThreadSanitizer, have been used as sources of inspiration. We note that the usage of a virtual platform, where our implementation is part of the host program that constitutes the virtual platform, implies that memory usage for dynamic analysis is not directly coupled to memory usage by the software application being analyzed. This means that we do not need to consider memory address collisions between the software application being analyzed and the dynamic analysis code, which gives us more freedom in the implementation.

Our implementation has been done on a known architecture, where we have full control over the implementation of each processor instruction. We have tried to make the implementation of dynamic analysis as a separate add-on to our available instruction set simulator, and the overhead when not using dynamic analysis has been minimized. This approach could perhaps be used also when using third-party processor models, either by the processor model vendor itself, or via a plugin-based system, where a user of a processor model could add code for propagation of V-bits and A-bits, and for error detection and reporting.

REFERENCES

[1] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in VEE '07, Proceedings of the 3rd international conference on Virtual execution environments, San Diego, California, 2007.

[2] D. Bruening, A. Potapenko, D. Vukov and K. Serebryany, "AddressSanitizer: a fast address sanity checker," in USENIX ATC'12, Proceedings of the 2012 USENIX conference on Annual Technical Conference, Boston, 2012.

[3] Valgrind Home, Valgrind™ Developers, Available: http://valgrind.org/

[4] Clang C Language Family Frontend for LLVM," LLVM Project, Available: https://clang.llvm.org

[5] MemorySanitizer — Clang 5 documentation, LLVM Project, Available: https://clang.llvm.org/docs/MemorySanitizer.html

[6] AddressSanitizer — Clang 5 documentation," LLVM Project, Available: https://clang.llvm.org/docs/AddressSanitizer.html

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978, issn: 0001-0782. doi: 10. 1145 / 359545 . 359563. Available: http://doi.acm.org/10.1145/359545.359563 .

[8] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in Proceedings of the Workshop on Binary Instrumentation and Applications, ser. WBIA '09, New York, New York, USA: ACM, 2009, pp. 62–71, http://doi.acm.org/10.1145/1791194.1791203 .

[9] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in Proceedings of the 30th ACM SIGPLAN Conference on Program- ming Language Design and Implementation, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 121–133, isbn: 978-1-60558-392-1. doi: 10.1145/1542476.1542490. Available: http://doi.acm.org/10.1145/1542476.1542490.

[10] D. Haverås, "Data Race Detection for Parallel Programs Using a Virtual Platform", Master thesis, Ericsson and KTH, Sweden, 2018, Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-230189

[11] U. Prabir Sinha, "Memory Management Error Detection in Parallel Software using a Simulated Hardware Platform", Master thesis, Ericsson and KTH, Sweden, 2017, Available: http://www.diva-portal.org/smash/record.jsf?pid=diva2:1164148

[12] Evgeniy Stepanov and Konstantin Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++", Available: https://static.googleusercontent.com/media/research.google.com/sv//pubs/archive/43308.pdf