Safe Virtual Machine

for C in less than 3 KiBytes

Bernhard H.C. Sputh, Eric Verhulst, Artem Barmin,
and Vitaliy Mezhuyev
Email: {bernhard.sputh, eric.verhulst,
Artem.barmin, vitaliy.mezhuyev}@altreonic.com

http://www.altreonic.com

From Deep Space to Deep Sea

Push Button High Reliability

Outline

- History of Altreonic
- Design of the Safe Virtual Machine for C (SVM)
- Integrating the SVM in the OpenComRTOS-Suite
- Performance Figures
- Conclusions
- Further Work

History of Altreonic

- Eonic (Eric Verhulst): 1989 – 2001
  - Developed Virtuoso a Distributed RTOS
  - Communicating Sequential Processes as foundation of the “pragmatic superset of CSP”
  - Open License Society: 2004 – now
  - R&D on Systems and Software Engineering
  - Developed OpenComRTOS using Formal Methods
- Altreonic: 2008 – now
  - Commercialises OpenComRTOS
  - Based in Linden (near Leuven) Belgium

Design goals for the SVM

- Distributed heterogenous systems are hard to maintain
  - Each target runs different binary, staticly compiled
  - Different compilers, different versions, different CPU variants
  - If system is large, probability that some nodes will fail or present issues is high
  - Still, system must remain operational during maintenance
- Dynamic code loading, independently of target CPU
- Back-up functionality for mis-behaving tasks
- Boundary conditions:
  - C is most often used embedded programming language
  - Memory resources are strictly limited

Design of the SVM

- Execution engine for OpenComRTOS Tasks, written in ANSI-C.
  - The C-Code gets translated into a binary using a standard C compiler
  - The SVM interprets this binary format.
- One Instance of the SVM executes in one OpenComRTOS Task, as Guest-Task.
- OpenComRTOS virtualises hardware for the SVM Guest-Task

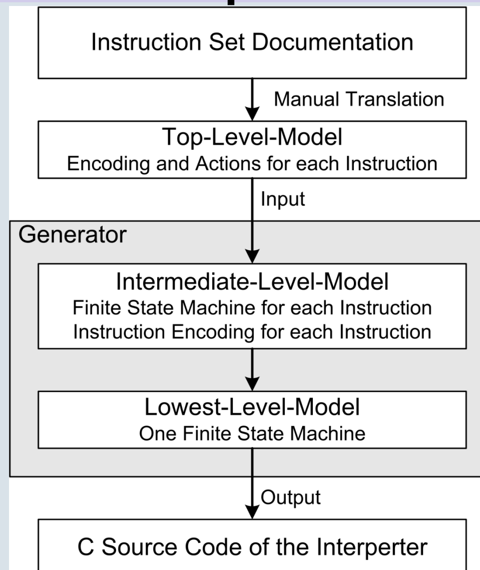
Selecting a VM Instruction Set

- **Criteria on the Instruction Set (IS):**
  - Compact byte code, i.e. small binaries
  - Not too many instructions
  - Availability of good toolchains / compilers
- **Instruction Sets evaluated**: MIPS, ARM Thumb-1, and Xilinx Microblaze
- **Chosen Instruction Set:** ARM Thumb-1:
  - Compact instruction set, all instructions, except one, only have 16bit
  - Widely used within the industry
  - But it can be any other IS as well

From the IS to the Interpreter Source Code – step 1

- The complete Instruction Set (IS) was modeled in a Domain Specific Language.
- The VM source code gets generated.
- The generator tool chain was written in Haskell.

From the IS to the Interpreter Source Code – step 2



Top Level Model of Push 1

```
Command "push"
  (Opcode ["1011","0","10"])
  [Operand "LR" 8 8,
   Operand "reg_list" 7 0]
  push
```


Top Level Model of Push 2

```
push = do
  when' (v "LR" .== i1) $
    do
      r "SP" .-= i4
      m [r "SP" .+ i0] .= r "LR"
  for' ([("i",i0),("j",i1)], v "i" .< i8,
   do {v "i" .+= i1;v "j" .<<= i1}) $

    when'((v "reg_list" .& v "j") .> i0)$
      do
        r "SP" .-= i4
        m [r "SP" .+ i0] .= rv "i"
```

Integrating the SVM in the OpenComRTOS-Suite

- Building the binary
- Virtualising the underlying OS and hardware
- Using the SVM inside an OpenComRTOS-System

Building the Binary

- OpenComRTOS is designed to support heterogeneous systems, i.e. systems that consist of different CPU-Architectures, which are called a `Platform’
- The SVM is just another platform that OpenComRTOS must support
- The SVM-Platform provides all the necessary support code in order to compile and link an OpenComRTOS-Task to form a binary file which can be loaded into the SVM.
- Furthermore, access libraries for all components that have been developed for OpenComRTOS can be used from within an SVM-Task.

Virtualising the underlying OS 1

- OpenComRTOS has two principal entities:
  - Tasks:
    - Prioritised (256 priorities are available);
    - Tasks do not share memory;
    - Tasks communicate with Hubs using Packets.
    - Also the Kernel is a task
  - Hubs:
    - Generic synchronisation primitive in OpenComRTOS
    - Hubs operate system-wide, but transparently → Virtual Single Processor (VSP) programming model.

Virtualising the underlying OS 2

- Services offered by Hubs
  - Event – Synchronisation on a Boolean value;
  - Semaphore – Synchronisation with a counter;
  - Port – Synchronisation and exchanging a packet, i.e. data transport (CSP Channel like);
  - Resource – Locking mechanism, with ownership;
  - FIFO – Buffered packet communication;
  - User defined Hubs are possible!

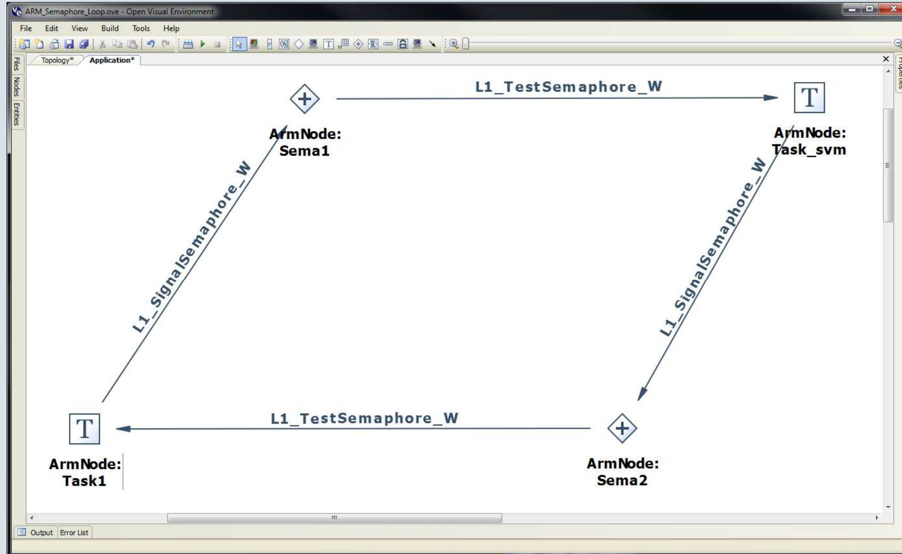
Virtualising the underlying OS 3

- Accessing to the underlying OS-API is done by using a Software Interrupt (SWI instruction ARM Thumb-1)
- Due to the OpenComRTOS API, being represented by the exchange of L1_Packets, only one function had to be provided: L1_buildAndInsertRequest()
- Additionally the SVM-Platform gets currently an extension to allow Guest-Tasks direct access to the memory of the Host
- Using the native OpenComRTOS, the SVM tasks have full access to underlying hardware, including other processing nodes in the network

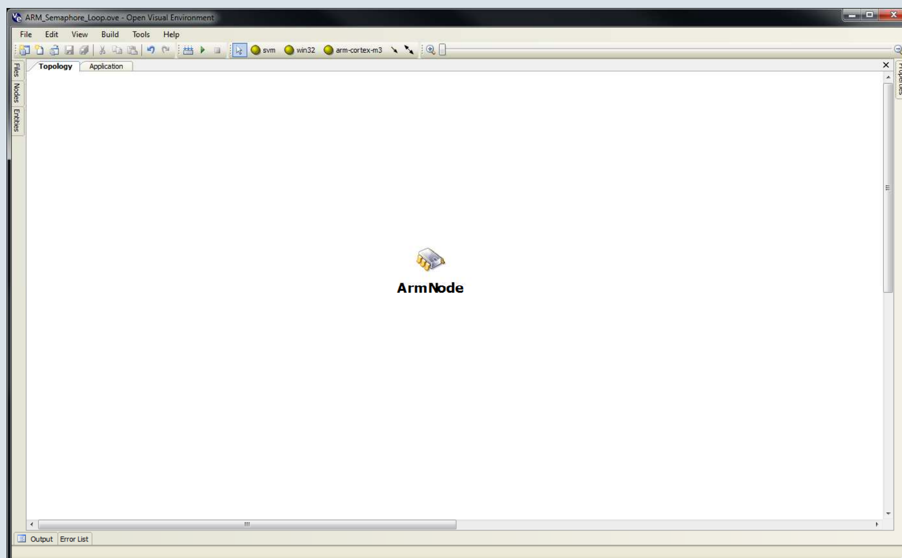
Using the SVM

- The SVM consist of two elements:
  - Topology: SVM-Platform:
    - OpenComRTOS Virtualisation
    - Build System
    - Component access libraries
  - Application: SVM-Component:
    - Virtual Machine Task
    - Control Task

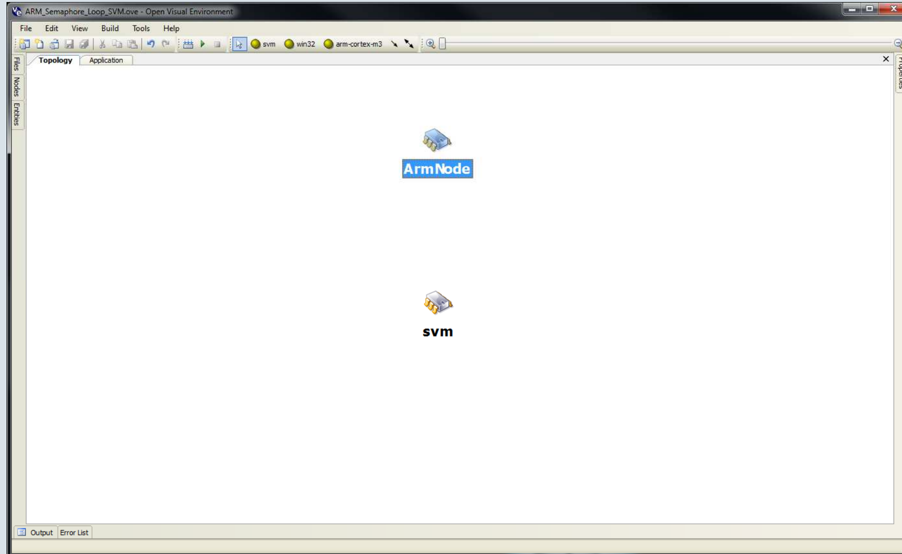
Using the SVM 1



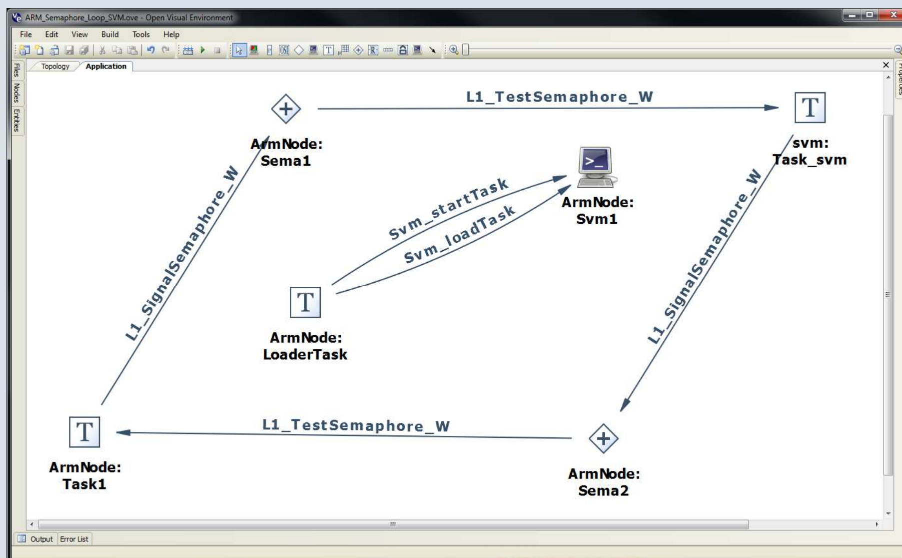
Using the SVM 2



Using the SVM 3



Using the SVM 4



Using the SVM 5

```
 3 #include<SvmService/SvmClient.h>
 4 #include<StdioHostService/StdioHostClient.h>
 5
 6 void Task2_ep(L1_TaskArguments Arguments)
 7 {
 8     L1_BYTE   buffer[1024];
 9     L1_UINT32 readBytes = 0;
10     L1_UINT32 fileHandle = 0;
11     /* Opening the executable file, for reading and in binary format */
12     if(RC_FAIL == Shs_openFile_W(Shs1, "Task_svm.bin", "rb", &fileHandle))
13     {
14         printf("Could not open file.\n");
15         exit(-1);
16     }
17     /* Copying the contents of the executable file into the local buffer */
18     Shs_readFromFile_W(Shs1, fileHandle, buffer, 1024, &readBytes);
19     /* Closing the file after reading */
20     Shs_closeFile_W(Shs1, fileHandle);
21     /* Loading the task into the program memory of the SVM. */
22     Svm_loadTask(Svm1, buffer, readBytes);
23     /* Starting the execution */
24     Svm_startTask(Svm1);
25 }
26
```


Performance Figures

- Memory Requirements of the isolated SVM.
- Comparing the memory requirements of a System with and without using the SVM.
- Performance Degradation, caused by the SVM.

Memory Requirements of the SVM

Optimisation	Code	Data
-O3	3,818 Byte	476 Byte
-Os	2,838 Byte	476 Byte

Compiled for ARM-Cortex-M3, using the Code Sourcery 2009q1 arm-none-eabi toolchain. The figures include the Supervisor and Interpreter tasks, and their helper functions. It does not include any data memory for the Guest-Task.

Impact on Memory Requirements

Semaphore Loop without Safe Virtual Machine:

Filename	Text	Data	Bss	Dec	Hex
ArmNode	4068	892	1708	6668	0x1a0c

Semaphore Loop with the Safe Virtual Machine:

Filename	Text	Data	Bss	Dec	Hex
ArmNode	8140	1736	4736	14612	0x3914
Task_svm	696	24	60	780	0x30c

Compiled for ARM-Cortex-M3, using the Code Sourcery 2009q1 arm-none-eabi toolchain, with compiler optimisation –Os.

Performance impact

- In the previously presented system, the runtime of the Semaphore loop increased by a factor of 7.26
- Depends on:
  1. The native code was compiled for ARM-Thumb-2 which might give better performance
  2. The current Instruction decoder implementation is not ideal, caused by the irregularity of the ARM-Thumb-1 Instruction set (not a restriction in HW, but for SW)

SVM application domain

- Not intended for high speed computations!
  - Dynamic code with small memory requirements
  - Target independent execution
- Typical use:
  - Diagnostics, post-deployment
  - Back-up tasks for natively failing application tasks
  - Run-time monitoring and logging
  - Execute existing binary code without compilation
    - Might require regenerating SVM for other IS.

Conclusions

- Generating the source code of the Virtual Machine, from High-level models made it easy to get the SVM code right.
- We can adjust the SVM to support almost any instruction set, by developing a top-level model of it.
- The OpenComRTOS Architecture made it easy to implement the OS-Virtualisation.
- The SVM brings OS-Virtualisation to deeply embedded systems (minimal overhead of 8kiByte)
- The SVM is useful in many Scenarios:
  - Isolating OpenComRTOS Tasks from each other.
  - Diagnostic of the Host-System
  - Runtime reconfiguration, for high reliability systems.

Further Work

- Automatic Stack monitoring by the SVM.
- Mobility of SVM-Tasks, i.e. moving tasks during runtime from one SVM instance to another instance.
- Native execution of SVM-Tasks, if the CPU supports the ARM-Thumb-1 instruction set.
- Improving the performance of the instruction decoder.

Questions?

Thank You for your attention

Visit us in Hall 11.0 Booth 101

"If it doesn't work, it must be art.
If it does, it was real engineering"