Safe Virtual Machine for C in less than 3 KiBytes

Bernhard H.C. Sputh, Eric Verhulst, Artem Barmin, and Vitaliy Mezhuyev
Altreonic NV; Gemeentestraat 61A bus 1; 3210 Linden; Belgium
Web: http://www.altreonic.com
E-mail: {bernhard.sputh, eric.verhulst, vitaliy.mezhuyev}
@altreonic.com
artem.barmin@openlicensesociety.org

From Deep Space to Deep Sea

Abstract

Altreonic is using a formalised approach to embedded software engineering. One recent example is Altreonic's novel Safe Virtual Machine for C (SVM). Tuned to the needs of embedded systems it allows to dynamically download C compiled binary code to OpenComRTOS nodes independently of the target processor. Yet, the Virtual machine requires less than 3 KiBytes of program memory (measured on an ARM Cortex M3).

Every processing node in an OpenComRTOS supported system can host multiple Safe Virtual Machine tasks, each of which can use the native kernel services and hence communicate system wide. SVM tasks can also be unloaded, updated at runtime as well as moved between networked OpenComRTOS nodes. A next release of the Safe Virtual Machine can verify memory accesses and catch boundary violations and numerical exceptions at runtime. The VM itself was generated in ANSI C from a higher level description. This has the benefit of correctness and allows to enhance the instruction set with little overhead. In addition the VM can be redefined using a different instruction set to address different functional needs.

As the VM is based on the ARM Thumb-1 instruction set, it is also possible to execute the binary images in native mode on most of the ARM processors. While VM tasks execute slower than native code, the performance is adequate given the intended range of applications. Typical applications for the SVM are remote diagnostics, fail safe & fault tolerant control, and processor independent programming.

1 Introduction

As embedded systems are increasingly becoming safety critical as well as networked, the designer is confronted with conflicting requirements. His system might include high end processors running a general purpose OS, as well as very small micro-controllers with no memory protection. Being able to program such systems transparently was the prime requirement of OpenComRTOS [1, 2]. Being formally developed, it features a very clean architecture and uses a packet switching communication layer. The result is that the code size on an OpenComRTOS networked node is typically in the area of 5 KiBytes.

Such systems however also need increasingly support for dynamic code and isolation between the application tasks. Traditional solutions, often derived from the high-end OS environments, exist but are not compatible with the use of small, memory constrained processing nodes. Virtual Machines will typically require 100's of MiBytes, whereas solutions like Java Virtual Machines (JVMs) still require memory in the order of MiBytes. The other issue is that will VMs provide isolation, they are problematic for predictable real-time performance.

The reality is also that most embedded programming is still done in C. This is largely due to the wide availability of C compilers and the capabilities of C to generate small code sizes while being very flexible, in particular for accessing hardware features of the processor. While C has a number of inherent safety issues, many of these issues disappear when code generators are used. The latter is the approach taken in the OpenComRTOS-Suite of tools. The programmer specifies the system graphically, while all RTOS parametrisation code being generated by our tools.

Hence, it was natural to look at a C based solution for a dynamic code support. Rather than developing a proprietary solution we opted for being able to use existing C compilers and hence the user can stay in his embedded RTOS development environment. Tasks written in C using the OpenComRTOS services can then also be executed either natively on the hardware or on the Safe Virtual Machine. As the Safe Virtual Machine itself is generated C code, it can be installed on any node that is supported by a C compiler, while the SVM tasks can be executed on any node in the system.

1.1 OpenComRTOS Architecture

OpenComRTOS uses a layered architecture which is based on semantic layering. The lowest functionality level is limited to priority based preemptive multitasking. On this level tasks exchange standardized Packets using an intermediate entity we call a Port-hub, a special instance of the general purpose 'Hub' used by OpenComRTOS for synchronisation and data exchange between tasks. Two tasks rendezvous by one task sending a 'put' request and the other task sending a 'get' request to a hub. The Port-Hub behaves similar to the JCSP Any2AnyChannel [3]. Hence, tasks can synchronise and communicate using packets and hubs. The packets are the essential workhorse of the system. They have header and data fields and are exclusively used for all services, rather than performing function calls or using jump tables. Hence, it becomes straightforward to provide services that operate in a transparent way across processor boundaries. In fact for a task it makes no difference whether a hub exists locally or on another node, the kernel takes care of routing the packet to the node which holds the Port-Hub. Furthermore, packets are very efficient, because kernel operations often come down to shuffling packets around (using handlers) between system level data structures.

At the next semantic level OpenComRTOS provides more traditional RTOS services like events, semaphores, resources, etc., all implemented using hubs. Finally, the architecture was kept simple and modular by developing kernel and drivers as tasks. All these tasks have a 'Task input Port' for accepting packets from other tasks. This has some unusual consequences like: a) the possibility to process interrupts received on one processor on another processor, b) having multiple kernel tasks on a single node. It also keeps the architecture simple and homogeneous.

1.2 SVM as an extension to OpenComRTOS

In the context of OpenComRTOS the Safe Virtual Machine for C (SVM), is nothing else than a specific set of tasks and hubs, hence the OpenComRTOS itself was not modified. The SVM is used by defining a protocol at the level of the SVM itself with interface services giving a SV task access to all underlying RTOS services and hubs, including communication to other nodes.

1.3 Structure

Section 2 details the design and development process of the SVM and describes its current state. This is followed by an explanation of the integration effort of the SVM into the OpenComRTOS-Suite ecosystem, in Section 3. Section 4 discusses in which situations the SVM is applicable, and n which not. This is followed by conclusions and further work in Section 5.

2 Design of the SVM

The Safe Virtual Machine for C (SVM) is, as the name states, meant as an execution engine of C programs. This means that the design goal was not to virtualise a complete CPU / MCU with all its hardware and intrinsic behaviours, but to provide a software component which can run a compiled and linked C program in binary format. To be precise the goal was not even to execute arbitrary C source code, but 'just' tasks that were written for OpenComRTOS, our own RTOS [1, 2]. These design goals make the SVM a type of process level Virtual

Machine, such as used to execute Java [4] or Erlang [5] programs. However the SVM provides at the same time a form of Operating System level virtualisation, such as for instance Xen [6].

2.1 Thumb 1 Instruction Set

As C is a complex programming language, which results in a complex and therefore big parser which is furthermore hard to verify, it was out of the question to directly interpret C programs at the source level. Instead, we decided to leave this complex task to readily available compiler toolchains, and only interpret the result of the compilation and linking process, i.e. the binary bytecode. The tricky bit was now to choose the instruction set of this bytecode. There were a number of criteria the instruction set had to fulfil:

- Compact bytecode — OpenComRTOS is used in highly resource constraint environments, thus the compiled and linked tasks for the SVM should not be as compact as possible.
- Not too many instructions — Every instruction that is in the instruction set must be implemented in the interpreter. This results in more code, and a more complex instruction decoder.
- Readily available and good compilers — Even the most compact bytecode is not useful if there are no compilers available to translate the C source code into bytecode.

We evaluated the following instruction sets: MIPS, ARM Thumb-1 [7, 8], and Xilinx Microblaze [9]. During this evaluation it became clear that the ARM Thumb-1 instruction set, is most suitable for the SVM. It has a limited set of instructions, of which all except one are 16 bit large[1]. This makes the decoding phase relatively simple. Furthermore, it is known to produce very compact bytecode and there are many commercial and free compilers available that can be used. An additional benefit is that the ARM processor is widely used.

2.2 Safe Virtual Machine development process

To achieve a compact, efficient, and maintainable solution for the virtual machine we decided not to hand code the binary code interpreter but instead to develop a Domain Specific Language (DSL) using Haskell. In this DSL we then modelled the ARM Thumb 1 instruction set and then generated the C source code that represents the binary code interpreter, from it.

For the virtual machine we have in fact used a hierarchy of DSL, to increase the level of abstraction. Each of the levels describes the virtual machine at a different level of abstraction.

1. Top-level — Each instruction of the instruction set is represented as a tuple of:
   - Instruction encoding — Listing 1 shows the representation of the push instruction at the opcode-level. First the name of the command is specified, in this case "push", then the opcode itself, followed by the operands that this instruction uses. Finally the name of the Action is specified.
   - Action — Listing 2 shows the specification of the action that should be performed when the `push` instruction is detected. The specification is written in a subset of the C programming language.

   The programmer only has to provide the top level to the generator, the following levels are constructed automatically from the top level model.

2. Intermediate-level — the VM is defined as set of commands, which each command defined as a tuple of:
   - A finite state machine, with each state containing the code and a set of possible transitions to other states. Each instruction is represented by two states:
     - Operand fetching;
     - Execution of the instruction;
   - A model which only contains the instruction encoding. This is used to generate the code for the instruction dispatcher.

[1] The instruction that is larger than 16bit is the long jump instruction.

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

Listing 1: Instruction encoding for the `push` instruction

```
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"
```

Listing 2: Specification of the Action of the `push` instruction

3. Lowest-level — In this model the VM is represented as a finite state machine (FSM). Each command is represented as a path in the FSM graph representation. The start state of the FSM performs the instruction fetching and dispatching. After executing every instruction the FSM returns to the start state. Having this combined FSM makes it possible to identify multiple usages of the same state and thus reusing their definition eliminating multiple copies, which in turn reduces the memory consumption.

Figure 1 illustrates the whole process of generating the SVM source code in form of a flow chart.

2.3 Interfacing with the underlying OS

For any process virtual machine it is vital to offer a way for the guest to access services from the underlying host. The following first explains how user tasks issue service-requests to the OpenComRTOS kernel, followed by an explanation on how the SVM virtualises the service-requests.

2.3.1 Service Requests in OpenComRTOS

OpenComRTOS is based on packet-switching, this means that all service requests from user tasks are represented by packets in the kernel domain. Thus to access any service a user task has to insert an L1-Packet into the input queue of the kernel task, the so called kernel input Port. For this purpose each task has a so called Request-Packet. Once a task has issued its Request-Packet it has to wait for it to be returned by the kernel task. Upon return the Request-Packet contains the result of the service request. The whole process of issuing the request has been implemented in the function: `L1_buildAndInsertPacket`.

2.3.2 Virtualisation of Service Requests

The interpreter component of the SVM task is itself an OpenComRTOS user task, which means that the other entities of the system are not aware that this task is actually a virtual machine. Furthermore, this means that the SVM task can issue service-requests like any other user task. The task being run inside the SVM has naturally also an Request-Packet available to itself, and a customised implementation of the function `buildAndInsertPacket`. The interfacing is thus done by copying the Request-Packet of the task being executed by the SVM task to the Request-Packet of the SVM task, and then calling the native implementation of `L1_buildAndInsertPacket`.

The SVM executed task informs the SVM about the need to issue a service request using a software interrupt. A software interrupt instructs the CPU, or in this case the interpreter, to execute code at a previous registered address, i.e. like an interrupt handler. The task running inside the SVM calls an implementation of the function

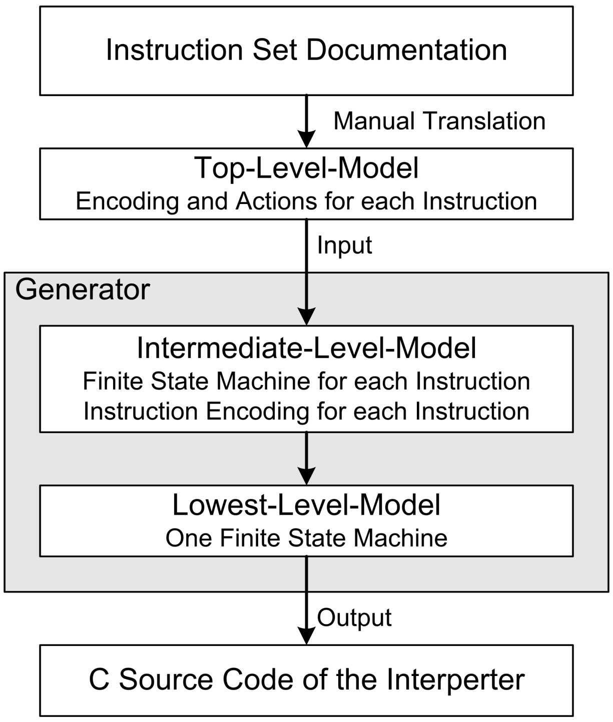
Instruction Set Documentation
Manual Translation
Top-Level-Model
Encoding and Actions for each Instruction
Input
Generator
Intermediate-Level-Model
Finite State Machine for each Instruction
Instruction Encoding for each Instruction
Lowest-Level-Model
One Finite State Machine
Output
C Source Code of the Interperter

Figure 1: Illustration of the Generation of the Interpreter

`L1_buildAndInsertPacket`, that copies the passed parameters into the general purpose registers r1–r4, and the code of the operation to initiate into register r0 (at least on the ARM interpreter, these registers can be different on other VMs). Then it issues the software interrupt, which results in the function `ProcessSwi` to be called, which then handles the software interrupt. In case of `buildAndInsertPacket` this means, to translated the contents of the SVM executed task's Request-Packet, i.e. copy all values into the SVM-Task Request-Packet doing endianness conversions where necessary. This is followed by calling the native implementation of `L1_buildAndInsertPacket`, and wait for the service-request to be completed. Afterwards the contents of the SVM Request-Packet are copied into the SVM executed task's Request-Packed, again endianness conversions where necessary and the call then returns.

3 Integrating the SVM into the OpenComRTOS Suite

There are two different aspects to consider during the integration of the SVM into the OpenComRTOS Suite: a) Building the binary that is executed by the SVM, detailed in Section 3.1. b) The SVM interpreter within an OpenComRTOS project covered in Section 3.2.

3.1 Building the Binary

The file the SVM loads must be a complete binary, with the standard program entry point (`main()`) and it must contain all the instructions necessary to execute the program. From the point of view of the OpenComRTOS Suite this means that the SVM represents its own platform. In the OpenComRTOS Suite a platform consists of: the operating system kernel as a set of libraries, code generator instructions that generate the build system and the parametrisation of OpenComRTOS, and a metamodel describing the available parameters. To use a platform in an OpenComRTOS project a Node will instantiate a specific platform. In case of the SVM this Node instantiates the SVM platform (from now on referred to as SVM-Node). By mapping a task to the newly created SVM-Node this task will be build for the SVM. The result of the compilation will be a standard bin file, such as one downloads directly to an ARM processor, except that it contains SVM specific code to communicate with the underlying OpenComRTOS environment.

3.2 Interfacing with the SVM Interpreter

With a binary that represents the code that should be executed inside the SVM and the Interpreter task in place the last missing bit is the interface between the OpenComRTOS environment and the Interpreter task. this allows to control the Interpreter task. For this purpose the SVM consists of two tightly coupled tasks, i.e. these two tasks may not be executed on two different Nodes. The first task is the Interpreter task, i.e. the task that will actually execute the SVM-Task. Furthermore, there is a Supervisor task, which is able to access the program memory of the Interpreter task and can start and stop the Interpreter task. The Supervisor task offers these services (among others) to the OpenComRTOS environment in form of what we call a Host Service Interface. Using the corresponding function calls, any user task is able to load a program into the SVM, start and stop the SVM, with the Supervisor task ensuring that all operations are performed safely. The use of the client library ensures that the communication between the User tasks and the Supervisor task is done correctly.

4 Discussion

Naturally, running a task inside the SVM brings with it some restrictions. The first obvious restriction is that the interpretation of the instructions results in a performance degradation versus a native execution. Thus one should not use the SVM when one needs raw processing power. However, for control algorithms this is most of the time less of an issue, especially when the SVM task runs a back-up algorithm to bring the system in fail-safe state. Thus one can use a simpler control algorithm to run in the SVM to still have control, even it may not be ideal. This provides graceful degradation of the functionality instead of an immediate loss of functionality. The usage scenario given in section 4.1.3 details this further. Another restriction is naturally, that one cannot directly interface with the underlying hardware, due to the fact that the SVM does not provide hardware virtualisation.

The complexity and the memory resource consumption increases due to the the need to run an additional supervisor task, and an interpreter task additionally to the task that runs inside the SVM. Therefore it was of great importance to keep the SVM as simple as possible while still meeting all other design constraints. The OpenComRTOS packet switching service model greatly contributed to this goal as it was only necessary to implement one gateway between the virtualised SVM-Node and the underlying OpenComRTOS, i.e. the function to build and insert a Packet into the kernel input port. This meant not only that we saved space, but we also saved development time because we only had to implement and test one function, to use all services provided by OpenComRTOS. Furthermore, the chosen instruction set (ARM Thumb-1) also contributed to this goal due to its very few instructions that had to be implemented, while still resulting in compact code. These are the two main factors that contributed to the less than 3 KiByte measured code size on an ARM Cortex M3 processor.

4.1 Usage Scenarios

The following sections give three examples of how the SVM can be used in system development to increase the reliability and diagnostics of OpenComRTOS Nodes.

4.1.1 Task Isolation

Programming mistakes are a human factor that cannot be fully avoided. Especially, with the C programming language it is very easy to make catastrophic errors, such as modifying the memory of another task, stack overflows, or accessing non existing memory. The SVM can prevent these kinds of problems by isolating individual tasks from each other, i.e. providing each task its own private memory. In this respect it acts like a software MMU. It can furthermore prevent out of memory accesses and gracefully stop the task trying to perform them, instead of the CPU throwing an exception, and thus halting the complete node.

4.1.2 Diagnostics

One scenario in which a dormant SVM instance can be of use, is to download a task that performs diagnostics tasks. This is for instance useful if an already deployed system does behave abnormally under specific conditions. In such a case a system specific monitoring code could be added to the system to investigate what happens in the system when things go wrong. This can be for instance recording the trace that leads to the erroneous behaviour.

Without the SVM one would have to modify the source code of the system and then redeploy. With the SVM one just has to write the monitoring task and download it into the running system.

4.1.3 High Reliable Heterogeneous Systems

The OpenComRTOS-Suite allows the construction of distributed heterogeneous systems, i.e. systems which consist of multiple nodes with different CPU architectures are used to fulfil a specific system requirement. To achieve high reliability in systems it is necessary that other nodes can compensate for one or multiple failing nodes. This means that the system reconfigures itself so that the operational nodes take over the core function of the failed node(s), by executing the core tasks. In homogeneous systems this can be achieved by simply adding the tasks to the other nodes. In a heterogeneous system this is not directly possible, as the executable needs to be available in a compatible binary format, i.e. once for each type of CPU used. This is where the SVM simplifies the life of the system designer because he just needs to provide all tasks in the binary format suitable for the the SVM, independently of the number of different CPU-Architectures used in the system. This drastically reduces the amount of storage necessary, and simplifies deployment and management of the system.

4.2 Performance measurements

Although the SVM is targeted at control and diagnostics in the context of safety, hence with less emphasis on the performance, it s still interesting to know how much performance degradation can be expected versus native execution. The native execution time was measured using an ARM Cortex M3 MCU 50MHz executing optimised code for it. Naturally, this is not a completely fair comparison, due to the Cortex M3 having the Thumb-2 instruction set which might increase the efficiency of the system. Despite this difference, it gives some indication of the slowdown caused by using the SVM. The test, we have done, concerns a recompilation of the OpenComRTOS semaphore loop (two task signalling each other in a loop using two semaphore hubs, [2] gives an explanation). For the test, we executed one of the tasks using the SVM. The degradation caused by the SVM is a factor of 7.26.

4.3 Memory requirements

The memory requirements of the SVM naturally depend on the used compiler and the optimisation level. Table 1 gives the resulting code and data sizes for the SVM host service, compiled for an ARM Cortex M3 using the 2009q1 Code Sourcery arm-none-eabi toolchain. The given code code size values include the following elements: the Interpreter task, the Supervisor task, and additional helper functions. The given data size includes all the operating system data structures in order to execute the SVM as well as the parametrisation structures of the SVM. Please note, these figures do not include the program memory for the executable that will be running inside the SVM.

Table 1: Memory consumption of the SVM in Byte

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

We give the code size at two different levels of optimisation: -O3 which should give the highest execution speed of the SVM, and -Os which is a size optimisation. The results show that the -O3 optimised version requires 3.8 KiByte, which is already very small for the given functionality. However, using -Os it is possible to shave off almost an additional KiByte from this size, thus reducing the code size to below 3 KiByte.

5 Conclusions & Further Work

This paper introduced the SVM, a very small virtual machine, meant for embedded systems, that can execute ARM Thumb-1 instruction set binaries. Section 2 first stated the reasons for the chosen instruction, including a detailed description how the instruction set was translated into ANSI-C source code to represent the interpreter.

Furthermore, this section explained how the interface between the guest running in the SVM and the underlying OpenComRTOS instance works. Section 3 detailed how the SVM was integrated into the OpenComRTOS-Suite of tools. Finally, Section 4 highlighted the restrictions of the SVM and potential usages of the SVM, followed by performance and memory requirement measurements.

5.1 Further Work

At the present moment of time the work on the SVM is not finished, it is in a phase of internal evaluation. While the core functionality is present, we are still working on the improving the usability of the SVM, among other things. The main topics were we are working on at the moment are:

- Mobility of SVM tasks. This is a functionality similar to process mobility in Occam π [10] and JCSP.mobile [11]. It allows to move a task from one node to another node at runtime.
- Stack Monitoring to avoid memory corruption. This is very useful on microcontrollers and embedded processors that have no hardware support to monitor memory access.
- Automatic loader task generation.
- Activating native execution, i.e. disabling the interpretation on the task, when the SVM runs on a MCU that can execute the ARM Thumb-1 binary directly.

Acknowledgments

The OpenComRTOS project is partly funded under an IWT project for the Flemish Government in Belgium.

References

[1] Vitaliy Mezhuyev Eric Verhulst, Gjalt de Jong. An industrial case: Pitfalls and benefits of applying formal methods to the development of a network-centric rtos. In Jorge Cuellar and Tom Maibaum, editors, *FM 2008: Formal Methods*, volume 5014 of *Lecture Notes in Computer Science / Programming and Software Engineering*. Springer, May 2008.

[2] Bernhard H.C. Sputh, Eric Verhulst, and Vitaliy Mezhuyev. OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In *Embedded World Conference 2010*, March 2010.

[3] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[4] Tim Lindholm and Frank Yellin. *Java TM Virtual Machine Specification*. The Java Series. Addison Wesley Publications, 1996. ISBN 0-201-63452-X.

[5] J. L. Armstrong and S.R. Virdin. Erlang - an experimental telephony programming language. Technical report, Computer Science Laboratory, Ellemtel Utvecklings AB, 1990.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] ARM Limited. *Thumb 16-bit Instruction Set Quick Reference Card*, March 2007.

[8] ARM Limited. *ARM10 Thumb® Family*, 2000.

[9] Xilinx. *MicroBlaze Processor Reference Guide*. http://www.xilinx.com.

[10] Fred Barnes and Peter H. Welch. Communicating Mobile Processes. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 201–218, September 2004.

[11] Kevin Chalmers and Jon M. Kerridge. jcsp.mobile: A Package Enabling Mobile Processes and Channels. In *Communicating Process Architectures 2005*, pages 109–127, September 2005.