An Esterel Virtual Machine (EVM)

Aruchunan Vaseekaran
Dept. of Computer Science
Columbia University, New York, NY
av62@columbia.edu

December 16, 2002

Abstract

Esterel is a synchronous, imperative language designed to specify deterministic control systems. However compilers and run time environments for Esterel are not available on low cost microprocessors and microcontrollers. We address this shortcoming by designing and implementing an Esterel Virtual Machine and compiler that will operate on a variety of low footprint target platforms. We will compare the efficiency of this approach to the traditional approach of generating native code. We will show that it requires less memory at the cost of execution speed.

1 Introduction

The Esterel programming language [1] is an imperative control flow language which includes semantics for concurrency and preemption based on a synchronous global clock. Esterel programs are deterministic and its semantics are formally defined. As such Esterel is highly suitable for building deterministic control systems. Unfortunately, Esterel is not widely available on many low cost embedded controllers.

A method of making programs highly portable is to compile then for an abstract virtual machine and then build virtual machine interpreters for all platforms on which the programs are to run. Programs written for the Esterel programming language can also be made highly portable in this manner at the cost of execution speed.

In this project we will design an Esterel Virtual Machine (EVM) and modify an existing compiler to generate code for it. The EVM will be a small C program. It will have instructions for signal handling, multiple threads of execution and context switching between threads.

Finally we will contrast the approach of generating EVM code to that of generating native machine code. We expect to find that the EVM approach is less efficient by 2 orders of magnitude in terms of execution speed, but that it is much more efficient in terms of total memory footprint.

The remainder of this paper consists of the following sections: Related Work, Esterel Primitives, Design of the EVM, Compiler for the EVM, Status of the Compiler and EVM and Conclusions and Future Work.

2 Related Work

The approach of compiling languages for virtual machines is not new. It was used at least as far back as the 1970's in Pascal Compilers [5]. The USCD Pascal Compiler generated code for an abstract machine called the P-Code Machine. The P-Code machine was stack-based and supported a global heap. It understood the primitive types supported by Pascal such as integers and sets. It had no support for concurrency or locking.

A more recent example of the use of virtual machines is the Java Virtual Machine [7] (JVM). THe JVM was designed for use in networked environments to ensure the secure platform independant delivery of programs accross a variety of computers and devices. All Java programs are compiled into Java Byte Code instructions for the Java Virtual Machine. At runtime, the Byte Code instructions are executed by the virtual machine. This process is sometimes optimized using a Just-In-Time (JIT) compiler, which will compile the Byte Codes into native instructions at run-time.

The JVM is stack-based like the P-Code machine but also supports higher level abstractions such as multiple threads of execution, object creation, object de-allocation and monitors for protecting critical sections of code.

The Common Intermediate Layer (CIL) [6] in Microsoft's .NET environment is another example of a virtual machine which is also stack-based. All languages in the .NET environment are compiled into CIL code. Like Java byte Codes, CIL supports high level object abstractions. The fact that all languages are compiled into CIL, means that modules written in different languages can interoperate in the same program.

A very different example of a virtual machine is VMWARE [1], which is used to simulate real hardware on real-platform. This software enables one to run multiple virtual machines on a single real machine instance. So a x86 machine running VMWARE can have running within it a virtual x86 machine running Windows NT and another virtual x86 machine running SUN Solaris. This type of virtual machine reduces hardware costs and server room real-estate at the expense of performance.

Many different compilation strategies exist for Esterel. The original approach by Berry [1] was to compile programs into a single finite state machine. This method produces very fast

[1] http://www.vmware.com

code but suffers from state-space explosion as the program size increases. An improved approach also due to Berry mapped the Esterel program into a network of logic gates and then generated code that simulated the network. This hardware simulation approach scaled very well with program size but generated much slower code than the single finite state machine approach.

The EC compiler by Edwards [2, 3], transforms the program into a concurrent control-flow graph. This graph is then analyzed and transformed into a set of individual program threads which are statically scheduled. We prefer to visualize this as a set of finite state machines which are statically scheduled. The point at which execution of a finite state machine stops and at a which another finite state machine resumes execution can be determined at compile time for most Esterel programs. The EC approach generates smaller and nearly as efficient code as the hardware approach but cannot compile all classes of legal Esterel programs.

3 Esterel Features

We will briefly review some of the features of the Esterel language in order to illustrate the design of the EVM. The features are the Esterel Primitives, Concurrency and Causality and Reincarnation.

3.1 Esterel Primitives

The Esterel language is built around a a few primive operations which can be used to express other language features.

```
present S then p else q end
emit S
loop p end
pause
suspend p when S
trap T in p end
exit T
p || q
```

The **present** statement checks for the existance of a signal and then conditionally executes its body. If the condition fails the **else** part of the statement is executed.

The **emit** statement, makes the signal S present in this instant. The **loop** statement is a conventional loop construct which executes its body repeatedly.

The **pause** statement is unconventional, it causes execution to stop for the current instant. In the next instant or cycle execution resumes at the statement succeeding the **pause** statement.

The **suspend** statement, executes its body in the current instant. If the body does not terminate in the current instance, the body will continue to be executed in later cycles only when the condition S is not present.

The **trap** and **exit** statement together form Esterels preemption mechanism. The body of the trap statement is run in all instants that the trap statement is executed. If an **exit** statement (corresponding to the trap) is executed inside the trap body the trap statement will terminate when the body of the trap has finished for the current instant. The body will finish when it either calls an **exit** or

when it executes a **pause** or when it terminates. When a body contains multiple threads runnning in parallel, then an **exit** in one thread causes all threads to terminate once the other threads have finished for the current instant.

Traps cans be nested and when multiple exits are executed (from threads within a trap) the exit for the outermost trap takes precedence over the other. The effect of the exit statement is equivalent to raising an exception.

The parallel operator is used to run multiple statements in parallel. The parallel statement terminates when all threads in the parallel have terminated. Note that the statements being run in parallel can terminate in the current instant or in later cycles. However the group of statements within the parallel will only terminate when all threads have terminated.

3.2 Concurrency and Causality

Compiling Esterel poses some subtle difficulties because of the instaneous nature of signal propagation and statement execution in the language. Sigals propagate instantly and can have only one value in a cycle. This means that statements that emit a given signal must be run before statements that test the value of that signal in the current instant. It is implied that the two sets of statements are in sibling threads, as it will be illegal to emit and then test the value of a signal within one thread within an instant. The statements in the threads are said to have causal links which must be maintained at run-time.

To compile such programs, the threads in the program must be scheduled so as not to violate causality. This scheduling is done statically at compile time.

3.3 Reincarnation

In general statements in Esterel can execute only once in a given instant, however in some cases there is complicated interaction between concurrency, traps and loops which can cause the statements to be executed more than once in an instant. This behaviour is called reincarnation and is a difficult compilation issue.

4 Design of the EVM

4.1 Design Choices

The main design choice in the EVM is wether to provide explicit instruction support for Esterel features such as threads and exceptions. The alternate approach is to design the minimum virtual machine which supports an esterel compiler which compiles down to a subset of C without any notions of threads or exception handling. The latter approach leads to a program which is more removed from the original Esterel program and will result in greater code size. This is because code needs to be inserted to remember the point where execution left off in a cycle. This is equivalent to inserting code to simulate a program counter.

The approach we have taken is that of providing explicit support for threads and exceptions. We believe this results in shorter machine code and is easier to compile down to.

The approach taken in designing the EVM is to specify a

relatively simple virtual processor with a few additional instructions for thread and exception handling. The EVM also has conventional branching, arithmatic and logical operations. The EVM does not specify a word size as this needs to be varied depending on the application. We can conceive of application which require a 8 bit word size such as one to control the lights in a Automobile to flight control applications which require 32 bit word size size. We envision that the word size will be a run-time parameter to the compiler and a compile time setting for the EVM source code.

4.2 Using the EVM

The EVM is initialized with the EVM Bytes Codes of all the modules comprising the application and a pointer to the topmost module. Once initialized the EVM is ready to be executed as part of a system.

The EVM is called once each cycle. Prior to being called it's state is updated with the state of external input signals for the current environment. Immediately after the EVM is called the values of external output signals are read from it and passed to the environment in some system specific manner.

The cycle time of an EVM program is determined by the maximum time it will take to run for a cycle. This is a function of the program, the efficiency of the EVM implementation and the speed of the processor which is running the EVM. It is the responsibility of the system designer to ensure that the cycle time of the EVM program is within the required response time of the system.

4.3 Registers

The EVM has a simple register set which includes a program counter (PC), 8 general purpose registers (R0-R8), a state register SR, and a flags register FL which holds the results of the last arithmatic or logical operation performed. There is also a stack register (SP).

4.4 Address Spaces & Data Structures

The EVM has a single byte addressable memory space, which is divided into a code area and a data area. The evm instructions are stored and executed from the the code area. The Data area is used to store evm state information, a stack as well as working memory for the executing progam. The state information comprises of signal states, thread states and EVM state. The stack in the EVM is very temporary and is not preserved accross thread invocations. Its purpose is to aid in evaluating expressions within a thread and within a cycle.

4.4.1 EVM State

The EVM state is used to restart the execution of the program between cycles. It consists of a pointer to the thread state of the main thread. The main thread is defined as the first thread in the topmost module of the program.

4.4.2 Signal State

The signal state is used to store the current values of signals in the system. The signals are referenced by index and the signal state area is organized as an array of both input and output signals.

4.4.3 Thread State

The thread state area is used to store information about all threads in the system. Each thread in the system has a globally unique id. The per thread information stored is its id,the completion code of the thread, the id of the parent thread, a trap state area, a pointer to the starting instruction for the thread and an area to save the register state of the thread when it needs to call a child thread or when it performs a co-routine style context switch to a sibling thread.

The thread completion code is an integer which stores the completion state of the thread. The completion code can take integer values of -1 and higher. A value of -1 means that the thread has performed a coroutine call to another thread. This means that the thread has not finished executing for this cycle. A value of 0 means that the thread has terminated. A value of 1 means that thread has finished execution for the current cycles (it executed a **pause**). A value higher than 1 means that the thread hit an exception of that value.

4.4.4 Trap State

The trap state is used to store the current traps that are active within the scope of the current thread. For each active trap there is a global trap id and an address to which the thread must branch to after an exception executed. This address is the address following the end of the corresponding trap statement or the address of the trap handler if one has been defined. By remembering which traps are active in this way, the evm knows what to do when the progam hits an exception. If it hits a exception for which there is a handler in the current thread it branches to the corresponding trap address. If there is no trap handler defined, then the EVM causes the thread to terminate with the completion code equal to the trap id. Control then passes to the calling thread and the action is repeated after waiting for any sibling threads to complete. Trap handling will be discussed further in a later section.

4.5 Instructions

The instruction set of the EVM can be divided into the following categories: signal, thread, trap, branching, memory address, arithmatic and logical. Instruction are encoded using variable length codes as some of the specialized instructions (**threadwait** in particular) need to be very large. The memory addressing, arithmatic and logical instructions will not be described in more detail except to say that all memory addressing is direct (for simplicity).

4.5.1 Signals

Signal instructions are used to test for the presence of a signal and to emit a signal. All signals have unique numbers to the EVM. So it is assumed that the compiler will map signal names to globally unique numbers. The instruction **sigtst** *signum* tests for the presence of a signal. The instruction **sigemit** *signum* makes a signal present.

4.5.2 Threads

All threads in the system have a unique id. Threads are defined and undefined by the instructons **threaddef** *tid*, *addr* and **threaddel** *tid*. A thread is run by calling **threadrun** *tid* or by calling **threadwait** which runs and waits for multiple threads. The **threaddone** instruction explictly terminates the current thread and returns control to the calling thread.

The **threadrun** *tid* function is used to run a single thread. This function returns control back to the calling thread in the current cycle in one of three ways: the called thread terminates, finishes executing for the current cycle or terminates by executing an exception. Control returns to the called thread when the callee is either finished for the cycle or has terminated without an exception, the instruction after the **threadrun** will be executed in the calling thread. If the thread terminates with an exception, then the calling thread will throw the corresponding exception.

The **pause** instruction is used to stop the thread for a current cycle. Control goes to the parent thread.

The **threadwait** instruction is used to run and wait for the completion of multiple threads. It is called with the count of child threads to wait for and a list of their id's. When **threadwait** is called in a cycle it goes through the threads id's in its list. It runs each thread in turn and checks their completion status. If all threads terminate, then the **threadwait** instruction finishes for the cycle and in the next cycle the instruction after **threadwait** will be executed. If the threads only finish for the cycle, then the **threadwait** instruction will be executed again in following cycles until all thread terminate or hit an exception. If a thread hits an exception it will terminate with completion code indicating an exception. In this case **threadwait** will wait for all other threads to finish for the current cycle and then cause an exception by executing **exit** *trapid*. If multiple threads that are being waited for by a threadwait terminate with an exception, then the threadwait raises the exception with the id equal to the maximum of the exception id (which are the completion codes) of the child threads. This mechanism in conjunction with the way that the Esterel compiler assign trap id's, ensures that the outermost exception is raised when multiple threads terminate with an exception.

As an example the following esterel code:

```
S1  ||  S2
```

would be compiled to the following EVM instructions:

```
    threaddef 5, S1_start;
    threaddef 6, S2_start;
    threadwait 2,5,6;
    threaddel 5;
    threaddel 6;

S1_start: S1
          threaddone;

S2_start: S2
          threaddone;
```

The example assumes that there are no causal relationships between S1 and S2 and that they can indead be run the order S1,S2.

If there is a complicated causal relationship between threads than the compiler can insert context switching instructions into the body of each thread to obtain the correct scheduling behaviour. The **threadswitch** *tid* instruction causes a co-routine style call to a sibling thread. To execute **threadswtch** the EVM notes that the thread is context switching by changing its completion code to -1. It then saves the current register state and jumps to the called thread. The called thread then returns to the calling thread by calling the **threadreturn** instruction.

Here is an example of an Esterel program which has complicated causal links:

```
    output a,b,c,d,e;

    present b then emit a end;
    present c then emit d end;
||
    emit b; present a then emit c end;
    present d then emit e end;
```

This program will be compiled as:

```
    threaddef 1, s1_start;
    threaddef 2, s2_start;
    threadwait 2, 2,1;

s1_start:
                #present b then emit a end;
                ..
                threadreturn ;
                #present c then emit d end;
                ..
                threadreturn;
                threaddone;
s2_start:       #emit b
                ..
                threadswitch 1;
                #present a then emit c end;
                ..
                threadswitch 1;
                #present d then emit e end;
                ..

                threaddone;
```

The **tstthreadfinished** *tid* and **tstthreadterminated** *tid* instructions are logical operators which test if thread is finished for the current cycle or has terminated.

The thread handling instructions for the EVM have relatively low overhead and because the amount of saved state for each thread is small and because there is no memory management operations that needs to be performed.

4.5.3 Traps

The EVM handles traps by recording the currently pending traps in each thread.All traps in the system have a unique id. The instruction **trapdef** *id,handleraddr* is used to tell the EVM the address to execute when the corresponding exception is executed. An exception is caused by executing the **exit** *trapid* instruction. When the exception is executed, the EVM searches for the trap definition in the current thread. When it find the definition it sends control to the instruction pointed to by the definition. This instruction is the one immediately after the corresponding Esterel trap statement or the start of the handler for that trap. If a trap is not defined in the current thread, the thread is terminated with a completion code equal to the trap id.

The **trapdel** *trapid* is used to undefine a trap from a thread. For example the following esterel code fragment:

```
trap T in
     s1;
     present S then exit T end;
     s2;
end
```

will be compiled as:

```
     trapdef 6, trap_end:
     # S1
     ..
     # present S then
     ..
     exit 6
     ...
     ...
      # S2
trap_end:
      trapdel 6
```

5 Compiler for the EVM

A Compiler was built by modifying the EC compiler developed by Edwards. EC was built using a compiler development environment called ESUIF. ESUIF structures the compiler as a set of compiler passes. In the ESUIF environment, compiler passes can be added or deleted independently making it a flexible environment to develop an Esterel compiler.

The EC compiler first transforms the input program graph into a form called the intermediate representation (IR). The IR was chosen to be close to C and also to capture Esterel facilities for premption, exceptions and concurrency. Our approach in building the compiler was to start with the IR and then transform it with new compiler passes that we defined until the resulting representation had a direct mapping to EVM instructions.

We dont use all the high level passes in the orginal EC. In particular we map abort statements to a combination of a trap and parallel statements. We do this because we think that this simplifies the IR.

A (strong) abort statement would be transformed from:

```
abort
     body
when S
```

to:

```
trap T in
     suspend body when S;
     exit T
||
     loop
          pause;
          present S then exit T end
     end
end
```

and the weak abort statement would be transformed from:

```
abort
     body
when S
```

to:

```
trap T in
     body
     exit T
||
     loop
          pause;
          present S then exit T end
     end
end
```

The IR consists of the following constructs:

```
if (expr) { stmts } else { stmts }
label:
goto label:
break n:
resume { stmts }
continue
try { stmts } catch 2 {stmts }
      catch 3 { stmts }
parallel { resumes } catch 1 {stmts } catch 2 { st
```

The **if**, **label** and **goto** constructs have their traditional meanings.

break 1 is **pause** in Esterel. Break at higher levels than 1 represent an exception. A **break** 0 means that the current statment or thread has terminated for the current cycle and control passes to the subsequent statement.

The **resume** statement is a statement that can resume its body in subsequent cycles. A **resume** statement is resumed in subsequent cycles by executing the continue statment. We dont use this mechanism in our compiler as the EVM keeps track of state between cycles using the PC for each thread. So a resume construct is compiled by mapping the statements in its body.

5.1 Compiling the Try statement

The try statement is the IR representation of an Esterel Trap statement and associated handlers. We translate it by adding trap definition instructions at the begining of the trap and putting in trapdel instructions at the end of the trap. So the following IR fragment:

```
try {
    body
} catch 2 {
    stmts_2
} catch 3 {
  stmts_3
}
```

would be compiled down to:

```
        trapdef 2, start_handler_2;
        trapdef 3, start_handler_3;
        body
        jmp end_trap;
start_handler_2:
        stmts_2
        jmp end_trap;
start_handler_1:
        stmts_2
        jmp end_trap:
end_trap:
        trapdel 2;
        trapdel 3;
```

5.2 Compiling the Parallel Statement

The parallel statement is compiled by first adding **threadef** instructions to the begining of the statement and adding **threaddel** instructions to undefine the threads at the end of the statement. Then a threadwait instruction is added to run the threads and collect their termination status.

The Esterel code fragment:

```
S1 || S2
```

is represented in the IR as:

```
        parallel {
           thread {
                  pause
                  emit A
           }
           thread {
                  pause
                  pause
                  emit B
           }

        }
```

and which is then compiled to:

```
        threaddef 1,thread_1_start:
        threaddef 2,thread_2_start:
        threadwait 2,1,2
        threaddel 1
        threaddel 2
        jmp parallel_1_end:

thread_1_start:
        pause
        sigemit 0
        threaddone

thread_2_start:
        pause
        pause
        sigemit 1
        threaddone
parallel_1_end:
```

The ordering of thread id's in a **threadwait** instruction must take into account the causal relationships between threads. If this relationship is complex then threadswitch/threadreturn instructions must be inserted into the body of the threads in order to maintain causality.

5.3 Compiling the Esterel Suspend Statement

The **suspend** statement is primitive operation in Esterel, which is not translated by EC so we compiled it by transforming it into a loop running the body as a thread:

```
suspend body when S;
```

becomes:

```
threaddef 6,thread_6_start;
threadrun 6;
pause;
loop_0_begin:
        #if thread 6 finished
        # then jmp suspend_end:
        tstthreaddone 6;
        jmpeq suspend_end;

        sigtst S;
        jmpeq present_end;
        threadrun 6;
        present_end:
        pause;
goto loop_0_begin;

thread_6_start:
body
..
..
threaddone;

suspend_end:
```

First a thread is defined for the body. It is then run once. If the thread has not terminated it is repeatedly run again within a loop only if the condition is absent.

6 Status of the Compiler and EVM

We have succesfuly modified the ESUIF compiler to perform the more important transformations including transforming parallels into EVM instructons and mapping abort statements in terms of traps, suspends and parallels. Hand simulation of the resulting code demonstrates the correctness of the resulting program. The details of the EVM have also been worked out it and a minimal subset of it has been implemented.

Given what we have learnt so far we believe that implementing a complete compiler and EVM is straitfoward. We believe that a 2-3 man month effort is required to produce a reasonably complete compiler and EVM.

7 Conclusions and Future Work

We have demonstrated how to build an EVM and how to compile code for it by modifying an existing compiler. We have shown that the EVM instructions can support the complex concurrency and exception handling needs of Esterel.

The transformations needed by specialized variants of the abort (such as abort .. immediate) statement need to be understood and we need to verify wether they can be handled correctly by the compiler. Work needs to be done to fully implement the EVM and compiler. Thereafter we need to compare its performance (in terms of speed and memory footprint) with other Esterel Compilers. Once these performance measurements have been made, the compiler and EVM will need to be optimized. One obvious optimization that can be performed is to remove all the branch statements that are needed to jump over the code for a thread in it parent. These branches can be removed by moving all the thread specific code to the end of the program.

We also plan to use the EVM to run an Esterel program on a microcontroller platform. The platform we will target is the LEGO Mindstorms Robot construction kit [4]. This brain of this kit is called the RX brick and is based on a Hitachi H8 microcontroller with 36K of externel RAM and 16K of on-chip ROM [2]. This will demonstrate a real Esterel application based on an EVM approach.

References

[1] Gerard Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics and implementation. *Scientific Computer Programming*, 19, November 1992.

[2] Stephen A. Edwards. An esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), 2002.

[3] Stephen A. Edwards. ESUIF: An Open Esterel Compiler. *Electronic Notes in Theoretical Computer Science*, 65(5), 2002.

[4] Jonathan B. Knudsen. *The Unofficial Guide to LEGO MINDSTORMS Robots*. O'Reilly, 1999.

[5] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[6] Thuan Thai and Hoang Q. Lam. *.NET Framework*. O'Reilly, 2001.

[7] Frank Yellin Tim Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[2] http://graphics.stanford.edu/ kekoa/rcx/#Hardware