Einsterel: A Dynamically Scheduled Compiled Event-Driven Simulator for Esterel

Michael O'Malley Halas
Columbia University
IBM

Vimal M. Kapadia
Columbia University
IBM

Abstract
The performance of compiled Esterel code is sub-optimal. Faster simulation of Esterel programs is needed. This paper presents a high-performance compiled event driven simulator for the Esterel language.

The main contribution is the scheduling of events at run-time; contrasting other Esterel compilers, which scheduled at compile-time. Only a portion of the Esterel language is implemented, but it is a large enough portion to decisively demonstrate the advantages and potential of using compiled event-driven simulation for the Esterel language.

Einsterel is able to simulate Esterel code faster Berry's V5 compiler. The speed increase is in the range of 21% to 171%, and the executable code size is slightly smaller. Our compile time is significantly longer. However, future work will be able to reduce it.

Introduction

We propose a new compiler for Esterel language, called Einsterel. It is the first compiled event-driven simulator that schedules events at run-time for the Esterel language.

Some aspects of EVCF (Event-Driven Conditional-Free) simulation, introduced by Maurer [1], are used for performance gains. EVCF offers improved performance over other event-driven simulation techniques, by avoiding loops and conditionals. Instead of having a type code for each node it has a list of function addresses. This is used to distinguish the node type. The function addresses are branched to directly using computed goto's. This avoids the decoding of the type code and the overhead of function calls [1].

Edwards [2] wrote ESUIF, an open Esterel compiler built on the SUIF 2 system. The ESUIF front-end builds a very high-level abstract syntax-tree-like representation of the source Esterel program [2]. This is stored in SUIF format. Because the front-end is separate, it allows for different compilation paths, making it a good front-end for Einsterel, as well. The SUIF is then used as input to Einsterel, which produces autonomous C code.

Being event-driven enables Einsterel to skip work on threads that are waiting for a signal, as they do not have to be scheduled until that signal changes. For blocks of code that are not being exercised, Einsterel will do essentially no work. The work by Bertin, et al [3] and any statically scheduled compilers suffer from wasting time on code that does not need to be run.

All of Esterel's kernel statements are implemented in Einsterel with the exception of trap and exit. This includes: nothing, emit, present, loop, ";", pause, suspend/when, and "||". The derived statement, await, is also implemented in its simple form as well as the Boolean operations (and, or, not).

Event Graph Structure
The event graph for Einsterel is called EinGraph. It consists of an array of EinNode's. An EinNode consists of an array of fan-ins and fan-outs, two integers giving the number of fan-ins and fan-outs, an integer storing the level, and a pointer to the address of a label. Fan-ins are inputs to a node; they are used to calculate its next value. The expression (A or B) would create a node of type "or" with fan-ins of nodes A and B. Fan-outs indicate which nodes a node can schedule. If a node "D" has a fan-in of "C" it does not imply that "C" has "D" as a fan-out. The "present" node in (present A then emit B) would have the "emit" as its fan-out, and A as its fan-in. However A would not have a fan-out of "present" and emit would not have a fan-in of "present". This is because A cannot schedule the "present" and emit once scheduled doesn't care about "present's" value. Figure 1 shows an EinNode.

```
struct EinNode
{
  int numFanIns ;
  int numFanOuts ;
  int level ;
  int *fanIns ;
  int *fanOuts ;
  void *pFunc ;
} ;
```

Figure 1: EinNode

Building the Event Graph

To further ESUIF, its pass on SUIF that decomposes the SUIF into Esterel code was used as a skeleton for constructing Einsgraph. As ESUIF's print code passes over the SUIF, Einsterel constructs and inserts the nodes into EinGraph. The fan-ins and fan-outs are also assigned during this process. When this pass on the SUIF is complete the levelization can occur because all nodal relationships have been resolved. The levelizing algorithm is as follows:

```
changed=1;
while someone's level has changed
   changed=0;
   for every node
     for every fanout of this node
       FixLevel(parents_level*);
```

Figure 2: Levelizing Algorithm

* - Exceptions exist.

```
highest_level_seen = parents_level
for each fanin
   if fanin_level > highest level_seen
      highest_level_seen = fanin_level
endfor
if highest_level_seen >= my_level
   my_level = highest_level_seen + 1
   changed=1;
```

Figure 3: FixLevel(parents_level)

There are some special cases that are not shown in the above figures. Some nodes, such as pause, have fan-outs in a lower level than themselves. When a node has a fan-out in a lower, level it means it schedules that fan-out for the next cycle. These fan-outs should not be moved to a higher level, and thus, are treated specially.

Event Wheel Structure

The event wheel is implemented as a two dimensional array. Each level in the array has space allocated only for the number of nodes that reside in that level. Accompanying the wheel are two arrays used for managing the wheel. The first, inWheel[nodeNumber], marks if the node is currently present in the wheel. The Second, numSched[levelNumber], tells the number of nodes scheduled in a level, and thus, doubles as a pointer to the last node in the level. Figure 4 shows the wheel structure loaded in an example state.

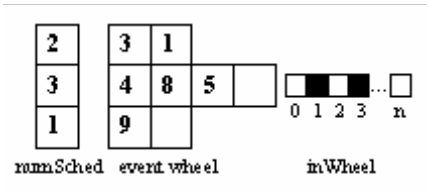


Figure 4. Loaded Event Wheel

Using this structure gives a significant performance advantage over a using a link list for each level. Memory doesn't need to be constantly allocated and deallocated when adding and deleting nodes from the wheel, and fewer operations are performed. An insertion of a node consists of two assignments and an addition. Only one assignment and a decrement are necessary for deletion of a node. Testing if a node is scheduled consists only of checking the value of inWheel[nodeNumber]. Finally the label pointer points to the address of the code that evaluates the node. Insertion and deletion of a node are shown in figures 5 and 6.

```
numScheduled[level]++;
wheel[level][numScheduled[level]]=nodeNumber;
inWheel[nodeNumber] = 1;
```

Figure 5: Inserting a node to the wheel

```
numScheduled[level]--;
inWheel[nodeNumber] = 0;
```

Figure 6: Deleting a node from the wheel

Scheduling and Running

The simulation of a cycle starts with the primary inputs being driven from <stdin>. A cycle is defined as one iteration over the wheel, and begins when a semicolon is received from <stdin>. If an input is driven with a different value then it had been driven with in the previous cycle, then its value is updated, and its fan-outs are scheduled. Then, each level in the wheel is iterated over; for each node in a wheel level, its new value is calculated. If this new value is different than the last cycle's value, then its fan-outs are scheduled.

The new value of a node is calculated by calling a node's "function". To avoid the overhead of a function call for evaluating every scheduled node, we use indirect goto's. To get the new value of a node, the code residing at the address pointed to by funcLabel is jumped to. This "function" sets a shared variable with the new value, and then jumps back. This is especially helpful because for most nodes the number of instructions run to calculate it's new value is very small and thus the function calls would largely dominate the evaluation time

```
for each level in the wheel
  while there are nodes scheduled in this level then
    for each node in this level
      delete it from the wheel
      goto (its evaluation code) and set eval
      if (eval != last cycle's eval) then
        schedule it's unscheduled fan-outs
```

Figure 7: Basic simulation algorithm

The scheduling algorithm discussed up to this point does not cover all node types. For these types, extra work must be done. The scheduling code assumes that if the new value of a node is the same as the old value, the node need not be scheduled.

While this works very well for logic gates, it does not cover the behavior of all Esterel statements.

Furthermore some nodes require the ability to schedule nodes in the next cycle. Allowing a node to schedule other nodes in a level less than its own accomplishes this. A node that is scheduled in a level less than the current level is thus being scheduled in a level that was already processed this cycle. As such, it will not be evaluated until the following cycle.

By creating a reschedule buffer, the ability for nodes to schedule themselves for the following cycle is also added. If a node needs to be rescheduled, its evaluation code pushes it into the reschedule buffer. When a level is done being processed, any nodes in the reschedule buffer get pushed back into the level, where they will be waiting until next cycle.

Nodes may not schedule nodes, other than themselves, in their own level. The levelizing code makes this ability unnecessary. If a node needs to schedule a fan-out in the current cycle, the fan-out is guaranteed to reside in a higher level. If it needs to schedule a fan-out in the next cycle, it is guaranteed to be in a lower level. Two examples will be explained to illustrate this.

Pause is a node requiring the ability to schedule another node in the next cycle. A pause is broken down into a pause1 node and a pause2 node. When a pause1 is scheduled, it schedules its pause2 into a level below its own. The next cycle the pause2 schedules the statement following the original pause in the Esterel code. This makes that branch of the event graph pause here and continue the following cycle.

Await nodes are also composed of 2 nodes (await1 and await2). Await1, when scheduled, simply schedules the await2 into a level below itself. The await2 is what checks the await condition, needs to be able to reschedule themselves. The algorithm for this is simple. If an await2 evaluates to false, then it reschedules itself using the reschedule buffer, so it can check the condition again next cycle. If it evaluates to true, then it is done waiting, and does not schedule itself; instead, its fan-outs will be scheduled.

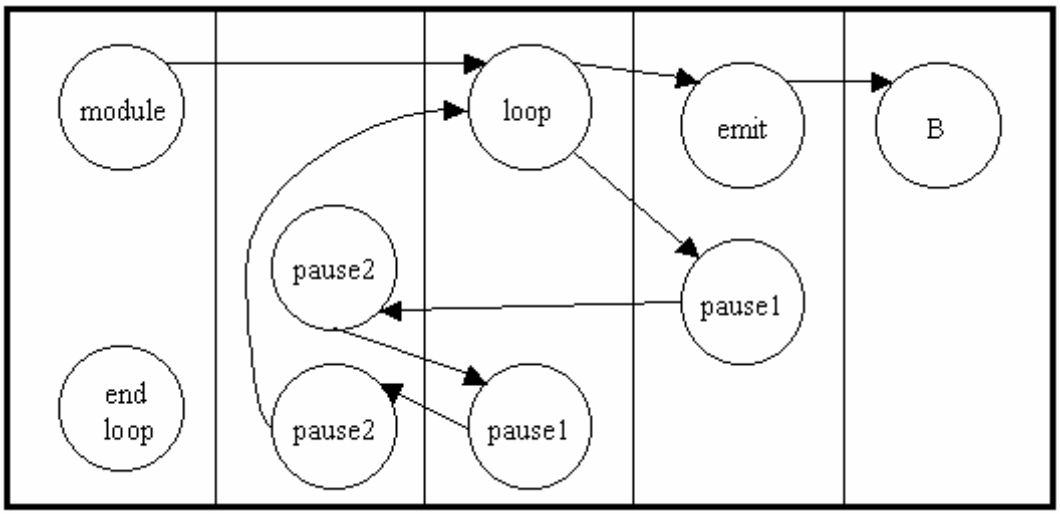
Figure 8 gives a cycle-by-cycle analysis of the scheduling of a loop containing an emit and two pauses.

Synthesizing C

Einsterel produces C code broken up into two main sections. The first section builds the compiled simulator's EinGraph by looping over the compiler's EinGraph. The event wheel and EinNode's get declared and initialized.

The simulation kernel is the second part of the generated C. The Esterel input file does not affect the simulation code or the "functions" for evaluating the nodes. This code is simply printed out at the end of the target C.

Comments are also outputted in the resulting code, and debug flags are included as preprocessor directives, but are turned off by default. This allows someone to recompile the code in "trace mode", allowing the user to see what code actually gets exercised, and to watch the scheduling process. This makes its easier for others to understand the inner workings of the scheduler, and makes for a more open compiler. Since trace code is "hidden" by preprocessor directives, it has no affect on normal performance.




```
loop
  emit B ;
  pause;
  pause;
end;
```

cycle 1: module schedules the loop and never runs again.
loop schedules the emit and the pause1
emit schedules B, and B is outputed
pause1 schedules it's pause2 to run the following cycle
cycle2: the pause2 schedules the next pause1
the pause1 then schedules its pause2 for the next cycle
cycle3 the pause2 schedules the loop and the process repeats

Figure 8 : Running a sample program cycle-by-cycle

Results

Figure 9 shows the percentage speed increase of Einsterel over Berry's v5 compiler. The testbench consists of 34 programs of varying size and parallelism. Two trends can be observed by examining this chart. The first demonstrates that Einsterel's performance advantage grows as the number of cycles run increases. This is due to Einsterel having a larger start up time then v5, and as the programs runs this initial load-time becomes less important.

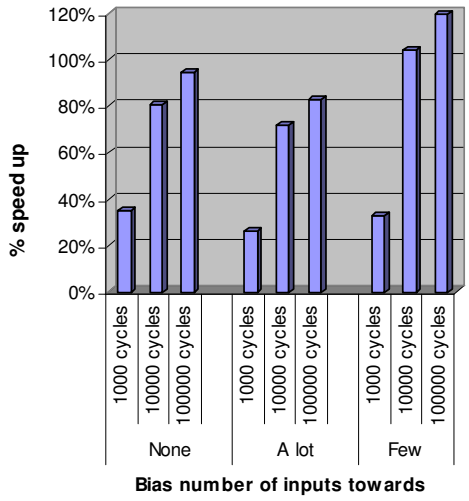


Figure 9

The second trend is that Einsterel's performance advantage grows as the number of inputs driven shrinks. This can be seen by looking at the results for 100,000 cycles. On average, when a program is run with large number of inputs driven most cycles, we are 83% faster. However, when only a few inputs are driven on most of the cycles, we are 120% faster. This jump is explained by the fact that Einsterel is event driven; this speed increase was expected to be seen in the results. When few inputs are driven, less of the Esterel code is exercised. Einsterel will not spend time doing anything to code whose inputs have not changed, v5 will. Thus, the less of the code structure that is exercised the more the performance gap will be apparent. Since most code spends the majority of its time in same segments, this is to Einsterel's advantage.

Figure 10 isolates the situations just described. Programs 100_g1 and 100_g2 are examples of code where most nodes will be scheduled every cycle. This contrasts 10_15, which has 15 concurrent threads, of which, only a small fraction run each cycle. One clearly sees the advantage of being event driven for programs where only small sections run at a time.

It also important to note that even for the programs where most of the code is exercised, Einsterel is still usually faster. This is also shown in Figure 10; even 100_g1 and 100_g2 show Einsterel being quicker.

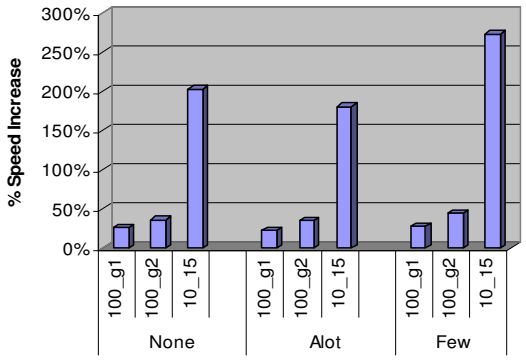


Figure 10

Conclusions and Future Work

This paper has presented a way to compile Esterel into a compiled event-driven simulator. It has demonstrated that event-driven simulation is both a feasible and efficient method of simulating Esterel code. The performance advantage over Berry's v5 compiler was clearly shown.

Compile time is one issue that must be handled; currently, compiling an Esterel file with Einsterel will take an order of magnitude longer than with Berry's v5. This is due to the way we initialize the data structures in the outputted C code. It is overly verbose, making up the vast majority of the resulting C file. The bottleneck in the compile process is the C compiler (g++) processing the initialization code.

Additional future work involves implementing the full functionality of Esterel. One of the challenges involved in that is creating an efficient algorithm for handling trap/exit. It is nontrivial, because it involves unscheduling nodes from the event wheel. To implement this, the trap/exit needs to know who to unscheduled, and who to leave alone. Tradeoffs may have to be made between making the common case fast and reducing the timing penalty of handling an exit for a trap.

Performance can be improved even further, with additional compile-time analysis; presents with only one condition can be treated separate from presents with multiple conditions, as you do not need to loop over your fan-ins if you have only one condition. Boolean/expression nodes can be collapsed into one node, to avoid the overhead of extra scheduling. Currently expressions are a series of binary/unary expression nodes. An optimizing pass can be added to collapse those nodes into a smaller number of nodes.

Lastly, work needs to be done on the detection of illegal cycles and instantaneous loops. Part of this analysis will most likely be done during the levelization process, and the rest may require a separate pass. Currently, the compiler assumes any code passed to it is valid code.

References

1. Peter M. Maurer: Event Driven Simulation Without Loops or Conditionals. ICCAD 2000: 23-26

2. Stephen A. Edwards, "ESUIF: An Open Esterel Compiler in Proceedings of Synchronous Languages, Applications, and Programming (SLAP)," *Electronic Notes in Theoretical Computer Science (ENTCS)* 65(5), April 2002

3. BERTIN, V., POIZE, M., AND PULOU, J. 1999. Une nouvellem´ethode de compilation pour le language ESTEREL [A new method for compiling the Esterel language]. In Proceedings of GRAISyHM-AAA. (Lille, France, March 1999).