Master's thesis defense

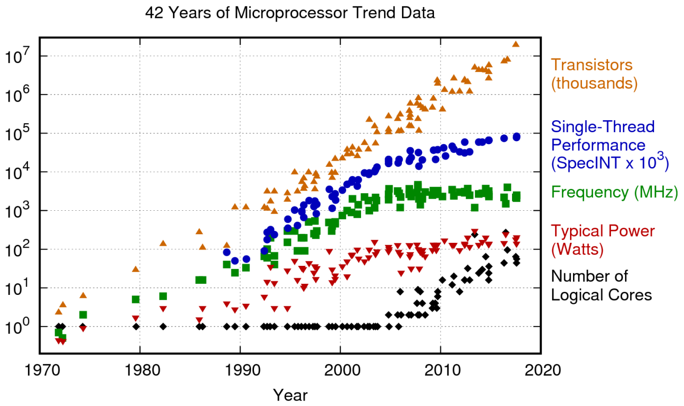
A Domain Specific Language for Synchronous Message Exchange Networks

Truls Asheim — `truls@asheim.dk`

Introduction

Current tendencies

Stagnant per-core CPU performance

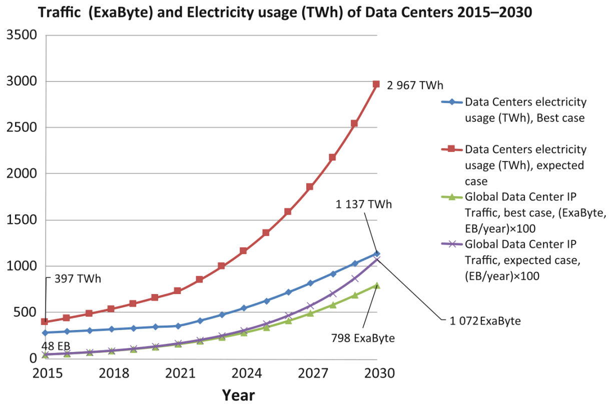


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallelism must be exploited in order to achieve performance improvements.

Current tendencies

Increasing data center power consumption



Increased efficiency is needed[1]

[1] From: The Internet: Explaining ICT Service Demand in Light of Cloud Computing Technologies by Hans Jakob Walnum and Anders S.G. Andrae

Custom hardware

- CPUs are slow and inefficient due to their versatility
- GPGPUs achieve significantly better performance and efficiency for SIMD workloads
- Field-Programmable Gate Arrays (FPGAs) offer good performance, while maintaining a much higher efficiency than GPGPUs and especially CPUs. But, extremely difficult to program.

Custom hardware

- CPUs are slow and inefficient due to their versatility
- GPGPUs achieve significantly better performance and efficiency for SIMD workloads
- Field-Programmable Gate Arrays (FPGAs) offer good performance, while maintaining a much higher efficiency than GPGPUs and especially CPUs. But, extremely difficult to program.

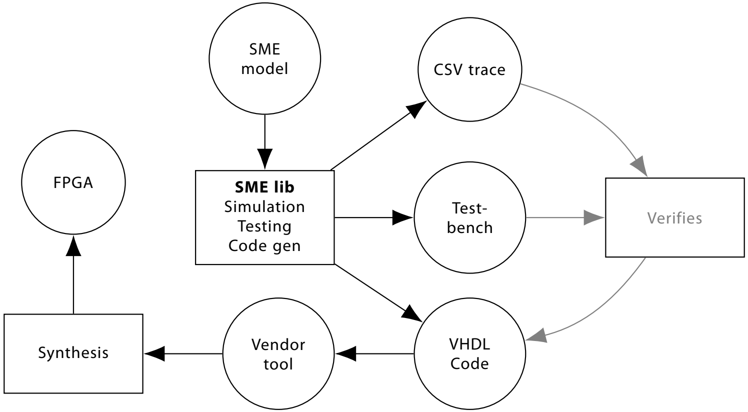
The utilization of FPGAs is being held back (partially) by poor design tools and languages. SME was introduced to alleviate this.

The Synchronous Message Exchange Model

Pure CSP was found to be unfit for modeling hardware due to the overhead induced by enforcing global synchrony by modeling a clock.

- SME properties:
  - Globally synchronous message propagation
  - Implicit clock
  - Shared-nothing
- Implementations for Python†, C#†, C++
- Key features:
  - Explicit concurrency using a model mimicking signal propagation in hardware
  - Allows cycle-accurate simulation of the resulting hardware
  - Automatic generation of VHDL (from †), test bench and test vectors for continuous verification of hardware description

SME Workflow



Initial motivation

Challenges

- Most SME development focused on C#, though the SME model was not intended to be specific to a single language. Thus, wider language support was desirable.
- Maintaining feature-parity of several independent SME implementations is infeasible due to duplication of code and effort.
- Interaction between SME components written using SME implementations for different languages was not possible.

Initial motivation

Challenges

- Most SME development focused on C#, though the SME model was not intended to be specific to a single language. Thus, wider language support was desirable.
- Maintaining feature-parity of several independent SME implementations is infeasible due to duplication of code and effort.
- Interaction between SME components written using SME implementations for different languages was not possible.

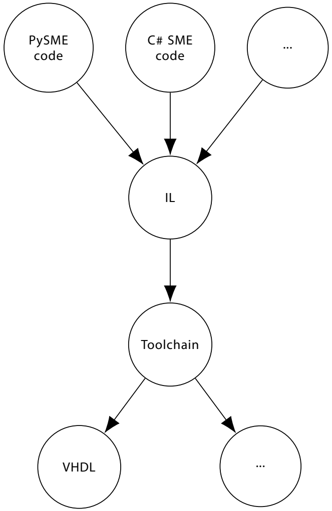
Solution

Introducing a common intermediate language and common tooling for the SME model.

Initial motivation: Common IL

Current

Initial motivation: Common IL



Current

Desired

The SME Implementation Language (SMEIL)

The SMEIL language

- SME primitives as first-class constructs
- Statically checked bit-precise typing
- Intentional commonality with general-purpose languages (e.g. Python and C#) (enable straight-forward mapping from high-level languages)
- C-like syntax — A tradeoff between simplicity of parsing and readability
- Purely a hardware modeling language — no simulation-only features[2].

[2] Some exceptions

The SMEIL language

- SME primitives as first-class constructs
- Statically checked bit-precise typing
- Intentional commonality with general-purpose languages (e.g. Python and C#) (enable straight-forward mapping from high-level languages)
- C-like syntax — A tradeoff between simplicity of parsing and readability
- Purely a hardware modeling language — no simulation-only features[2].

Usable both as a primary implementation language for SME models and as an IL for other SME implementations.

[2] Some exceptions

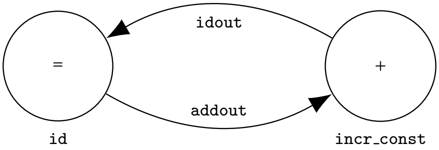
Example: the addone network

```
proc id(in inbus)
  bus idout {
    val: int = 0;
  };
  var it: uint = 0;
{
  idout.val = inbus.val;
  trace("Iteration: {} Value: {}",
    it, inbus.val);
  it = it + 1;

}
```

```
proc incr_const(in inbus, const val)
  bus addout {
    val: int = 0;
  };
{
  addout.val = inbus.val + val;
}

network addone() {
  instance addone_inst of
    incr_const(id_inst.idout, val: 10);
  instance id_inst of
    id(addone_inst.addout);
}
```



Supported types

`int`, `uint`	Unconstrained (unlimited size) signed/unsigned integer
`i4`, `u54`	Constrained signed/unsigned integers
`bool`	Booleans
`f32`, `i64`	Single/double floats[3]
Arrays	Static length non-nested arrays

[3] Currently only for simulation

Co-simulation

Co-simulation

Why?

- SMEIL is purely intended for expressing hardware models — not for test benches. Thus, no support for, e.g., file I/O or advanced visualization
- Therefore: method for providing external interactions is needed

Co-simulation

Why?

- SMEIL is purely intended for expressing hardware models — not for test benches. Thus, no support for, e.g., file I/O or advanced visualization
- Therefore: method for providing external interactions is needed

Key goal: providing seamless integration of SME networks written in SMEIL and another language enabling them to be co-simulated as single entity

Co-simulation (continued)

How?

Exposing a co-simulation API from libsme enabling external access to simulation control and reading/writing from/to bus channels

Co-simulation (continued)

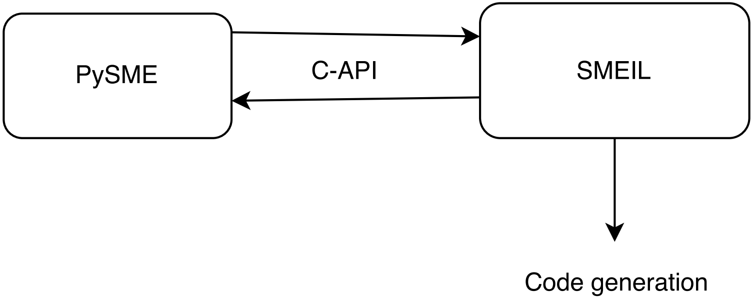
How?

Exposing a co-simulation API from libsme enabling external access to simulation control and reading/writing from/to bus channels

Co-simulation is a commonly use technique for supplying test vector to hardware design (e.g., CoCoTB, VPI, VHPI). The advantage of the SMEIL approach is that SME is used on both sides of the co-simulation

Co-Simulation with PySME

PySME extended with co-simulation support:



- Interaction with the SMEIL simulator through a C-API
- Exchanged values recorded in trace file for use with VHDL test bench

PySME co-simulation interface

Simple interface enables co-simulation with minimal intrusion on PySME code

Python side

```
@extends("collector.sme",
        ["-f", "--trace",
"trace.csv"])
class ColorBin(Network):
  def wire(self, result):
     img_out_data =\
        ExternalBus("image_input")
     result_bus =\
        ExternalBus("bin_count_out")
     self.add(img_out_data)
     self.add(result_bus)
```

SMEIL side

```
proc collector (in image_input)
  exposed bus bin_count_out {
     valid: bool;

     low: u32;
      med: u32;
      high: u32;
  };
// [..]
network coll_net() {
  exposed bus image_input {
    valid: bool;
    last_pixel: bool;

     R: u8;
     G: u8;
     B: u8;
};
// [..]
```

Constraining bit-widths from value observations

Constraining bit-widths from value observations

- Unconstrained integers are not representable in hardware, so unconstrained types (e.g. `int`) must be constrained (e.g. `i8`) before we can generate a hardware description
- No registers of pre-defined sizes (bit-widths) exists in hardware, so we are free to define the sizes we need
- Synthesizing unnecessarily wide wires will lead to wasted space and an inefficient implementation
- But, knowing the minimal number of bits needed for a value in advance is sometimes hard
- Most existing hardware description languages either require explicit sizing of all integer types or use bad defaults (e.g. an `int` is 32-bits)

Constraining bit-widths from value observations

- Unconstrained integers are not representable in hardware, so unconstrained types (e.g. `int`) must be constrained (e.g. `i8`) before we can generate a hardware description
- No registers of pre-defined sizes (bit-widths) exists in hardware, so we are free to define the sizes we need
- Synthesizing unnecessarily wide wires will lead to wasted space and an inefficient implementation
- But, knowing the minimal number of bits needed for a value in advance is sometimes hard
- Most existing hardware description languages either require explicit sizing of all integer types or use bad defaults (e.g. an `int` is 32-bits)

Idea

Constrain unconstrained channels based on the range of values assigned to them during simulation

Example

Our example from before. Note the unconstrained integer types.

```
proc id(in inbus)
    bus idout {
        val: int = 0;
    };
    var it: uint = 0;
{
    idout.val = inbus.val;
    trace("Iteration: {} Value: {}",
        it, inbus.val);
    it = it + 1;

}

proc incr_const(in inbus, const val)
    bus addout {
       val: int = 0;
    };
{
    addout.val = inbus.val + val;
}

network addone() {
    instance plusone_inst of incr_const(id_inst.idout, val: 1);
    instance id_inst of id(plusone_inst.addout);
}
```

Lets simulate it for 10 cycles.

Output from simulator after simulation for 10 cycles

```
proc id (in inbus)
    var it: u6 = 0 range 0 to 10;
    bus idout {val: i4 = 0 range 0 to 5;};
{
    idout.val = inbus.val;
    trace("Iteration: {} Value: {}", it, inbus.val);
    it = it + 1;
}

proc plusone (in inbus, const val)
    bus plusout { val: i4 = 0 range 0 to 5;};
{
    plusout.val = inbus.val + val;
}

network plusone_net ()
{
    instance id_inst of id(plusone_inst.addout);
    instance plusone_inst of plusone(id_inst.idout, val: 1);
}
```

What about 50 cycles?

Simulation for 50 cycles:

```
proc id (in inbus)
    var it: u6 = 0 range 0 to 50;
    bus idout {val: i6 = 0 range 0 to 25;};
{
    idout.val = inbus.val;
    trace("Iteration: {} Value: {}", it, inbus.val);
    it = it + 1;
}

proc plusone (in inbus, const val)
    bus plusout { val: i6 = 0 range 0 to 25;};
{
    plusout.val = inbus.val + val;
}

network plusone_net ()
{
    instance id_inst of id(plusone_inst.addout);
    instance plusone_inst of plusone(id_inst.idout, val: 1);
}
```

Note: Signedness of original type is preserved

Usage of observed ranges

The SMEIL code with observed ranges has two possible uses:

1. The observed ranges are used for constraining unbounded types in SMEIL such that VHDL with bounded types can be generated

Usage of observed ranges

The SMEIL code with observed ranges has two possible uses:

1. The observed ranges are used for constraining unbounded types in SMEIL such that VHDL with bounded types can be generated
2. The original SMEIL code is rewritten to include the updated ranges – may be used as input for formal verification.

New feature: improved range granularity

- In the handed-in version of libsme, all observed ranges was between 0 and the maximum absolute value
- Sufficient for deriving type bounds
- However, formal verification of observer ranges required more detail
- Therefore, the actual range (e.g. 10 to 55 or -24 to 30) is now tracked
- Non-trivial implementation: Added distinction between undefined and defined values

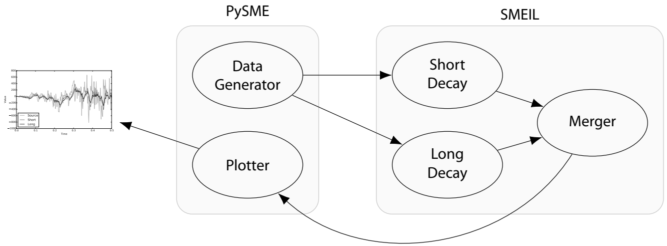
Evaluation

SMEIL has been evaluated through the following examples

- Modification of the PySME to VHDL compiler to generate SMEIL instead of VHDL directly
- An image pixel color intensity bin counter
- **The core of a high-frequency trading chip** (up next)
- An MD5 bruteforcer
- A model 7-segment digital clock

Example: High-frequency trading chip

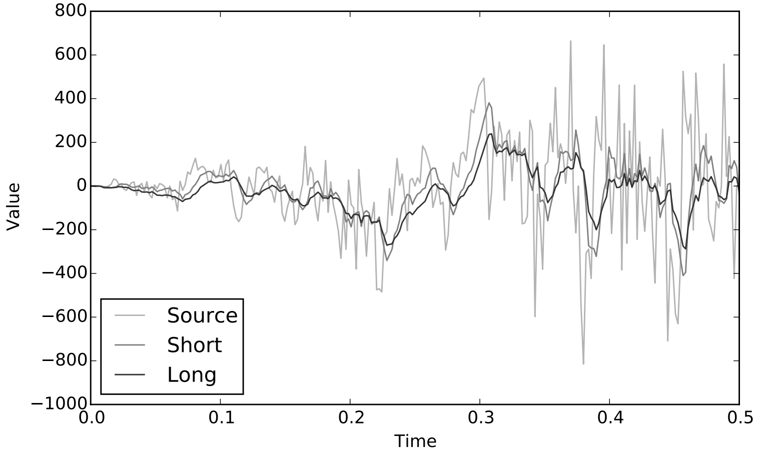
Makes a trading decision based on calculation of Exponential Moving Averages with long and short decays



Uses the co-simulation interface for writing input generation- and verification code in Python and the core processes in SMEIL

Example: High-frequency trading chip (continued)

Result of execution:



Future work

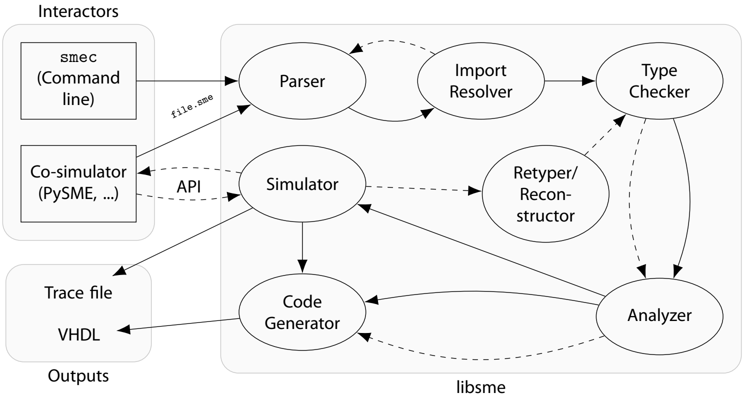
- Feature parity between SMEIL/libsme and C# SME: Make SMEIL fully capable IL for C# SME
- Hardware-software co-design. Add automatically configured opaque interfaces enabling inter-device communication (e.g. between SME processes running on GPGPUs and FPGAs)
- Formal verification of observationally derived value ranges (in progress).
- Parameterized network orchestration
- Sugary SMEIL: Further improvements of the language as a primary implementation language for SME networks

Conclusions

- We have introduced SMEIL, a DSL for SME networks shown in practice to be able represent several designs already implemented in SME
- SMEIL is usable both as an intermediate language and as a primary implementation language for SME networks
- Implementation supporting VHDL code generation
- Co-simulation interface enabling writing of test code in a general-purpose language
- The size of values can be constrained based on ranges observed during simulation

Questions?

Implementation



Initial motivation: Easy analysis

Having a SME representation which enables easier analysis of SME networks.
Concrete syntax of other languages is difficult to analyze

EWMA SME code

```
// ewma.sme

sync proc calc (in data, const decay)
    bus result {
        val: i32;
        valid: bool;
    };
    const sub: i32 = 1;
    var prev: i32 = 0;

{
    if (data.valid) {
        result.valid = true;
        my.prev = (data.val >> decay) +
                  (my.prev >> decay) *
                  ((1 << decay) - 1);
        result.val = my.prev;
    } elif (!data.valid) {
         result.val = my.prev;
    } else {
        result.valid = false;
    }
}

sync proc merge (in long, in short) {
    exposed bus output {
        short: i32;
        long: i32;
        valid: bool;
    };
{
    // [..]
}

network ewma (in stream, out result) {
    const decay1: int = 2;
    const decay2: int = 3;

    exposed bus stream {
        val: i43;
        valid: bool;
    };

    instance short of calc (data: stream,
                            decay: decay1);
    instance long of calc (data: stream,
                           decay: decay2);
    instance _ of merge (long: long.result,
                         short: short.result);
}
```

EWMA Python code

```python
@extends("ewma.sme")
class EWMA(Network):
    def wire(self, indata, outdata, *args, **kwargs):
        count = len(indata)
        alpha = 0.1

        stream = ExternalBus("stream")
        output = ExternalBus("output")

        logger = Logger("Logger", [output],
                        [], count, outdata)
        self.tell(logger)
        source = Source("Source", [], [stream],
                        indata, count)
        self.tell(source)

class Source(External):
    def setup(self, ins, outs, data, count):
        self.map_outs(outs, "out")
        self.data = data
        self.count = count
        self.gen = self._datagen()

    def _datagen(self):
        for e in self.data:
            yield e
```

```python
    def run(self):
        self.out["val"] = next(self.gen)
        self.out["valid"] = 1

class Logger(External):
    def setup(self, ins, outs, count, results):
        self.map_ins(ins, "data")
        self.results = results
        self.curpos = 0

    # Save data to array

def main():
    sme = SME()
    with open(data_file, 'rb') as f:
        indata = np.load(f)
    outdata = np.zeros(indata.shape)
    sme.network = EWMA("EWMA", indata, outdata)
    sme.network.clock(255)
    x = np.linspace(0, 0.5, len(indata))
    p = plt.plot(x, indata, x, outdata)
    pylab.show()

if __name__ == "__main__":
    main()
```

MD5 bruteforcer

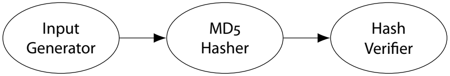


Figure: Structure of the MD5 bruteforcer network.

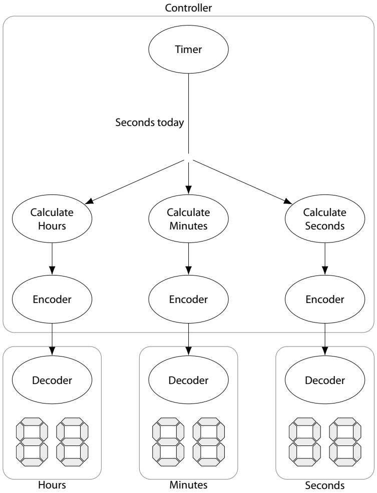


Figure: Model digital clock using a 7-segment display. A timer keeps track of the number of seconds elapsed since midnight and several processes calculates and lights.