Automated synthesis of FSMD-based accelerators for hardware compilation

Nikolaos Kavvadias
Ajax Compilers
Athens, Greece
Email: nikos@nkavvadias.com

Kostas Masselos
Department of Computer Science and Technology
University of Peloponnese
Tripoli, Greece
Email: kmas@uop.gr

***Abstract*—In this work we extend the FSMD (Finite-State Machine with Datapath) model to encompass synchronous memory accesses, intermodule communication and hardware-optimizing transformations. A lightweight typed assembly language, N-Address Code (NAC), is used as a designer-friendly representation of FSMDs, simplifying the adaptation of hardware synthesis with existing frontends.**

The quality (computation time, chip area) of the generated FSMDs has been evaluated on modern FPGAs. Our approach overcomes the C code limitations of four HLS tools while maintaining a good speed/area balance.

I. Introduction and related work

The annual increase of chip complexity is 58%, while human designers productivity increase is limited to 21%[1]. A drastic increase in designer productivity is only possible through the adoption of methodologies and tools that raise the design abstraction level, ingeniously hiding low-level, time-consuming, error-prone details. EDA methodologies as High-Level Synthesis (HLS) [1] aim to generate synthesizable and verifiable RTL (Register Transfer Level) designs from algorithmic descriptions.

An overview of FSMD modeling [3] can be found in [4], where synchronous communication among modules is not discussed. Cycle-accurate RTL modeling in [5] identifies the lack of established, timed, RTL models. In a relevant ASM (Abstract State Machine) formalism [6], data races are possible and it is difficult to perform low-level optimizations such as operation chaining.

Commercial HLS offerings include AutoESL[2], CatapultC, ImpulseC, Synphony HLS and C-to-Silicon. Attaining binaries for their evaluation was not possible.

LegUp[3] generates MIPS coprocessors, however in low-level, vendor-specific HDL. Publicly accessible tools producing generic HDL include ROCCC[4], GAUT [5], SPARK[6], C-to-Verilog[7] and TransC[8]. ROCCC targets streamable C applications on a feed-forward pipeline. It is restricted to perfectly nested constant-bound loops. GAUT is incapable of handling non-static loops. SPARK only handles loops with fixed constant iteration counts and assumes that all data is transferred to the chip before the computation starts, rendering some designs infeasible. C-to-Verilog is an LLVM[9] Verilog backend, however presents limitations in accessing arrays within functions. TransC supports streaming constructs for data exchange and process synchronization, through non-standard C constructs.

Existing approaches have certain drawbacks: a) most frontends donnot emit self-contained FSMD specifications; b) mandating the use of code templates to detect memory accesses and intermodule communication and c) succumbing to vendor and technology dependence.

In this paper, NAC [2] serves as both a compiler IR and natural FSMD specification. To support this, a frontend from GIMPLE[10] dumps to NAC is used throughout the text. Relevant RTL facets to memory accesses, hierarchical modules and hardware-oriented optimizations such as operation chaining, are automatically generated. FSMD synthesis does not rely on code templates since it uses a graph-based backend. Further, the generated HDL code is human-readable and completely vendor- and technology-independent. We have implemented our approach as part of the HercuLeS[11] high-level synthesis tool.

II. NAC (N-Address Code)

NAC supports n-input/m-output mappings, user-defined data types (integer, fixed-/floating-point arithmetic), SSA form, and scalar, single-dimensional array and streamed I/O procedure arguments. NAC statements are n-address operations or procedure calls. An (n, m)-operation specifies a mapping from a set of n ordered inputs to a set of m ordered outputs:

```
o1, ..., om <= operation i1, ..., in;
```

where `o1, ..., om` are the m outputs and and `i1, ..., in` the n inputs of the operation. NAC uses the notions of "globalvar" (a global scalar or array variable), "localvar" (a local variable), "in" (an input argument to the given procedure), and "out" (an output argument).

[1]http://www.itrs.net/reports.html
[2]http://www.xilinx.com/tools/autoesl.htm
[3]http://www.legup.org
[4]http://www.jacquardcomputing.com/roccc/
[5]http://www-labsticc.univ-ubs.fr/www-gaut/ (2011)
[6]http://mesl.ucsd.edu/spark/
[7]http://www.c-to-verilog.com
[8]http://cgi.tu-harburg.de/~ti6hm/
[9]http://www.llvm.org
[10]http://gcc.gnu.org/wiki/GIMPLE
[11]http://www.nkavvadias.com/hercules/

For instance, an addition of two scalar operands is written as: `a <= add b, c;`. Control-transfer operations include explicit conditional and unconditional jumps. An example of an unconditional jump would be: `BB5 <= jmpun;` while conditional jumps always declare both targets: `BB1, BB2 <= jmpeq i, 10;`. Multi-way branches corresponding to compound decoding clauses can be easily added.

The memory access model defines dedicated address spaces per array. For an indexed load in C (`b = a[i];`), a frontend would generate the following NAC: `b <= load a, i;`, while for an indexed store (`a[i] = b;`) it is: `a <= store b, i;`, both using the array identifier as an explicit operand.

Procedures are non-atomic operations; in `(y) <= sqrt(x);` the square root of an operand `x` is computed.

III. Extended FSMDs

The FSMD [7] introduces embedded actions within the next state generation logic of an FSM. Our extended synchronous FSMD model supports: array input and output ports, streaming I/O, communication with embedded block and distributed LUT memories, design of a latency-insensitive local interface between caller and callee FSMDs, and design of memory interconnects for the FSMD units.

A. Conventions

The FSMDs are organized as computations allocated into $n+2$ states, where n is the number of required computational states. The two overhead states, `S_ENTRY` and `S_EXIT`, correspond to the source and sink nodes of the CDFG of the given procedure, respectively. One possible optimization is merging the sink state with its immediate predecessors. Input registering is supported although this intent has to be made explicit in NAC. The control interface is simple:

- clk (I): signal from external clock
- $reset$ (I): synchronous or asynchronous reset
- $start$ (I): activates the FSMD so that in the next cycle, the first computational state is reached
- $ready$ (O): the block is ready to accept new input
- $valid$ (O): asserted when the corresponding data output port is streamed-out from the block
- $done$ (O): end of computation for the block

`ready` signifies only the ability to accept new input (non-streamed) and does not address the status of an output.

B. Communication with embedded memories

We assume a RAM model with write enable, and separate data input (`din`) and output (`dout`) sharing a common address port (`rwaddr`). A `store` operation raises write enable (`mem_we`) in a given single-cycle state so that data are stored in memory and made available in the subsequent state/machine cycle.

Synchronous `load` requires the introduction of a `waitstate` register. This register assists in devising a dual-cycle sub-state for performing the load. Fig. 1 illustrates its implementation. During the 1st cycle of `STATE_1` the memory block is addressed. In the 2nd cycle, the requested data are made available through `mem_dout` and assigned to register `mysignal`. This data can be read from `mysignal_reg` during `STATE_2`.

```
when STATE_1 =>
  mem_addr <= index;
  waitstate_next <= not (waitstate_reg);
  if (waitstate_reg = ’1’) then
    mysignal_next <= mem_dout;
    next_state <= STATE_2;
```

```
  else
    next_state <= STATE_1;
  end if;
when STATE_2 =>
  ...
```

Figure 1. Wait-state communication for loading data from a block RAM.

```
when STATE_1 =>
  isqrt_start <= ’1’;
  next_state <= SUPERSTATE_2;
when SUPERSTATE_2 =>
  if ((isqrt_ready = ’1’) and
      (isqrt_start = ’0’)) then
    m_next <= m_eval;
    next_state <= STATE_3;
```

```
  else
    next_state <= SUPERSTATE_2;
  end if;
when STATE_3 =>
...
isqrt_0 : entity WORK.isqrt(fsmd)
  port map (clk, reset,
    isqrt_start, x_reg, m_eval,
    isqrt_done, isqrt_ready);
```

Figure 2. State-superstate communication of caller and callee procedure instances in VHDL.

C. Hierarchical FSMDs

Hierarchical FSMDs define entire systems with caller and callee CDFGs. A two-state protocol describes proper communication, using a “preparation” state and an “evaluation” superstate where the entire computation applied by the callee FSMD is effectively hidden.

The caller FSMD performs computations where new values are assigned to $\star$`_next` signals and registered values are read from $\star$`_reg` signals. To avoid the problem of multiple signal drivers, callee procedure instances produce $\star$`_eval` data outputs that can be connected to register inputs by hardwiring to $\star$`_next` signals.

Fig. 2 illustrates a call (`(m) <= isqrt(x);`) to an integer square root. `STATE_1` sets up the `isqrt_0` callee instance which reads `x_reg` and produces `m_eval`. In `SUPERSTATE_2` control is transferred to the component instance of the callee. When the callee instance terminates, `isqrt_ready` is raised. Since `isqrt_start` is kept low, the generated output data can be transferred to the m register via its `m_next` input. Control then is handed over to `STATE_3`.

D. Streaming ports

Streaming suits applications with absence of control flow. In a prime factorization algorithm ($pfactor$), a streaming output can be used, `outp`, to produce successive factors. The streaming port is accessed based on `valid`. Thus, `outp` is accessed periodically in context of basic block `BB4` as shown in Fig. 3.

E. Operation chaining

Operation chaining assigns dependent SSA operations to a single control step. Simple means for selective operation chaining involve merging successive ASAP states. In

```
procedure pfactor (in u16 x, out u16 outp) {
  localvar u16 i, n, t0;
BB1: n <= mov x; i <= ldc 2; BB2 <= jmpun;
BB2: BB3, BB_EXIT <= jmple i, n;
BB3: t0 <= rem n, i; BB4, BB5 <= jmpeq t0, 0;
BB4: n <= div n, i; outp <= mov i; BB3 <= jmpun;
BB5: i <= add i, 1; BB2 <= jmpun;
BB_EXIT: nop;}
```

Figure 3. NAC code for a prime factorization algorithm.

```
...
when S_1_3 =>
  t3_next <= "000"&x_reg(15 downto 3);
  t4_next <= "0"&y_reg(15 downto 1);
  next_state <= S_1_4;
when S_1_4 =>
  t5_next <= x_reg - t3_reg;
  next_state <= S_1_5;
when S_1_5 =>
  t6_next <= t4_reg + t5_reg;
  next_state <= S_1_6;
```

(a) VHDL code without chaining.

```
when S_1_1 =>
  ...
  t3_next <= "000"&x_next(15 downto 3);
  t4_next <= "0"&y_next(15 downto 1);
  t5_next <= x_next - t3_next;
  t6_next <= t4_next + t5_next;
  ...
```

(b) VHDL code with chaining.

Figure 4. Chained computations.

successive states, intermediate registers are eliminated by wiring assignments to ⋆_next signals and reusing them in the subsequent chained computation, instead of reading from the stored ⋆_reg value. To avoid excessive critical paths, a heuristic is defined for disallowing flow-dependent multiple occurrences of expensive operators in the same newly defined state. In Fig. 4 states S_1_3 to S_1_5 comprise intermediate computations in a merged S_1_1 state.

F. *High-level optimizations*

A set of grammatical transformations has been developed using TXL[12]. As proof-of-concept, matrix flattening and argument globalization are examined.

Matrix flattening deals with reducing the dimensions of an array from N to one. This optimization creates multiple benefits: addressing, interface and communication simplifications, and direct mapping to physical memory. Argument globalization replaces multiple copies of a given array by a single-access globalvar array. It prevents exhausting interconnect resources for single-threaded applications. Through a bus-based hardware interface, globalvar arrays can be accessed by any procedure.

IV. Performance of FSMDs

HercuLeS is used for C-to-VHDL synthesis with the help of a prototype translator from GCC GIMPLE dumps to NAC. It extracts Graphviz CDFGs from NAC, which are then synthesized to vendor-independent self-contained RTL hardware descriptions.

Computation- (C) and memory-intensive (M) benchmarks have been selected from public domain. In Table I, for each benchmark (Bench.), a short description is given (column 3), and its type (C, M or both) is shown in column 2. Source lines are given in columns 4-7, respectively for the user C code, NAC, Graphviz CDFGs and VHDL.

[12]http://www.txl.ca

Table I
Benchmark descriptions and line statistics.

Bench.	Type	Description	C	NAC	dot	VHDL
edgedet	M	Edge detection	35	145	873	1921
float2half	C	Convert float-to-half	25	71	157	370
fsme	M	Motion estimation	65	159	1483	2730
half2float	C	Convert half-to-single	12	32	55	174
icbrt	C	Integer cubic root	18	36	83	213
isqrt	C	Integer square root	20	28	84	199
mandel	C/M	Mandelbrot fractal	60	108	259	639
matmult	M	Matrix mult.	40	94	763	1511
sierpinski	C/M	Sierpinski triangle	51	70	300	630
smwat	M	Smith-Waterman kernel	68	159	753	1615
walsh	M	2D Walsh transform	32	71	326	704
yuv2rgba	C	Color space conv.	27	98	240	679

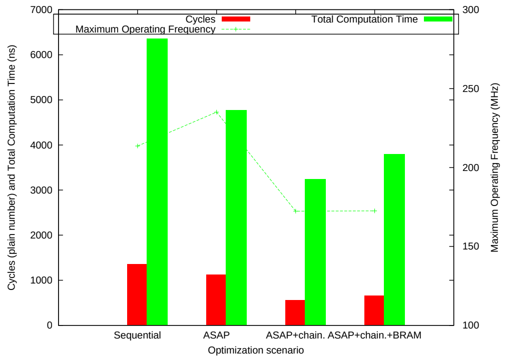


Figure 5. Number of cycles, MOF and TCT (geometric means) for the generated FSMDs.

A. *Speed measurements*

To assess the performance of the generated hardware, the minimum propagation delay (MPD), maximum operating frequency (MOF) and total computation time ($TCT = cycles \times MPD$) are evaluated. Four different scheduling scenarios have been examined: S1) sequential scheduling, S2) ASAP, S3) S2 with chaining, and S4) S3 with synchronous read (block RAM) memories. Such linear complexity schedulers are critical for providing fast compiles on sizable benchmarks.

A graphical view of this information, showing relative TCT metrics is illustrated in Fig. 5. All designs were synthesized on the XC6VLX75T Xilinx Virtex-6 device using Xilinx Webpack ISE 12.3i.

Average TCT is reduced by 47% when comparing S1 to S3. BRAMs impose a fixed cycle readout latency, which limits this gain to about 36.4%. Computer arithmetic problems (*float2half*, *half2float*) achieve improvements of about 4× reduction in TCT. Memory-intensive benchmarks (*matmult*, *smwat*, *walsh*) present lesser opportunities due to the memory accesses activity interfering with chaining of arithmetic operations within the same clock cycle. All benchmarks achieve MOFs in the range of 120-450MHz.

B. *Chip area measurements*

An aspect of the FPGA area measurements is shown in Fig. 6. S1 allows for the generation of smaller hardware in terms of slice LUTs and registers. In S3, LUT and register

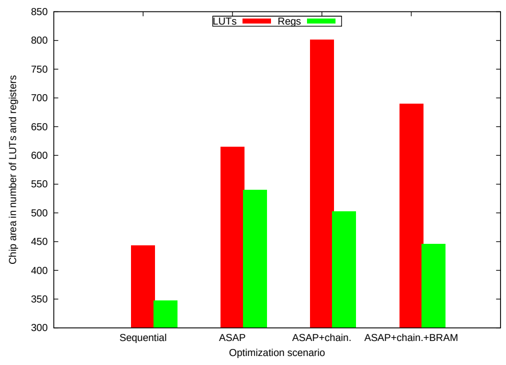


Figure 6. Chip area in number of LUTs and registers (geometric mean) for the generated FSMDs.

demand is increased by 90.1% and 59.1%, compared to S1, a price paid for much higher speed. However, it is more reasonable to compare S2, S3 and S4 which all apply ASAP on the NAC SSA form. Registers are reduced by 17.5% regarding the geometric means for the corresponding metric among S2 and S4. The tradeoff among S3 and S4 is very clear; S3 provides better speed performance in exchange for worse LUT and register utilization, where S4 excels.

C. Comparison against accessible HLS tools

Quantitative comparisons against other HLS tools were investigated on a kernel suite (HLSbench) as shown in Fig. 7.

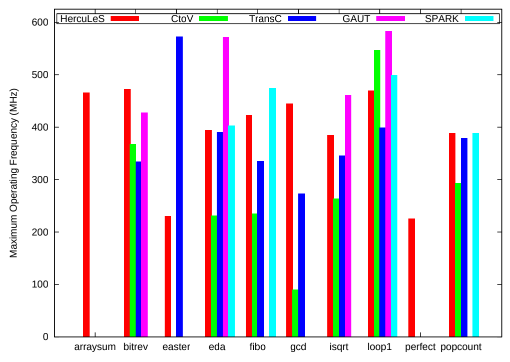
Applications in the HLSbench C suite include array sum, bit reversal, Easter date calculation, distance approximation, iterative Fibonacci series, greatest common divisor, integer square root, a synthetic loop, perfect number detection and population count. All tools except HercuLeS required benchmark source code adaptations. GAUT and SPARK required more effort than C-to-Verilog and TransC. Tweaks were applied to absorb third-party tool issues, for instance, C-to-Verilog produced simulation-only code. All tools except HercuLeS do not support hardware division, due to lack of both respective component libraries and support for state multicycling.

HercuLeS is the only tool that supports the entire HLSbench application suite; it maintains performance comparable to SPARK, but less than GAUT. GAUT can effectively pipeline generated designs for the applications it supports (5 out of 12). SPARK has the lowest ANSI C compatibility (4/12). C-to-Verilog and TransC appear to have closely matched results, supporting 7/12 kernels.

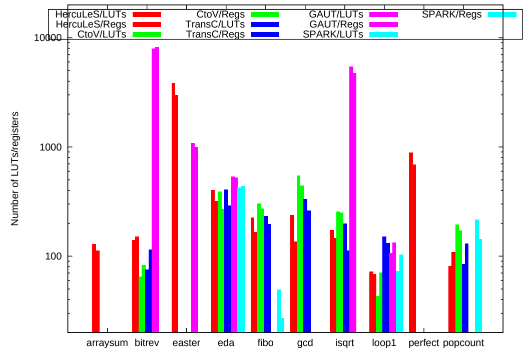
HercuLeS is second only to SPARK in chip area; GAUT introduces impractical LUT and register demands. Since only a limited set of optimizations is currently considered, we expect improved performance in future versions through extra optimizations.

V. Conclusion

An extended FSMD model supporting synchronous module communication, embedded memories, streaming outputs, and hardware optimizations has been presented. Further, an IR that enables the automated synthesis of FSMD-based accelerators using HercuLeS has been discussed. The proposed techniques have been evaluated against four different optimization scenarios on a number of benchmarks using HercuLeS as well as against accessible HLS tools with promising results.



(a) Maximum operating frequency (MHz).



(b) Chip area in number of LUTs and registers.

Figure 7. HLS tools comparison on HLSbench.

References

[1] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuits*. Springer, 2008.

[2] N. Kavvadias and K. Masselos, "NAC: A lightweight intermediate representation for ASIP compilers," in *Proc. Int. Conf. on Engin. of Reconf. Sys. and Applications (ERSA'11)*, Las Vegas, Nevada, USA, Jul. 2011, pp. 351–354.

[3] P. P. Chu, *RTL Hardware Design Using VHDL*. Wiley, 2006.

[4] H. Lehr and D. D. Gajski, "Modeling custom hardware in VHDL," UC Irvine, Tech. Rep. ICS-TR-99-29, Jul. 1999.

[5] R. Dömer, A. Gerstlauer, and D. Shin, "Cycle-accurate RTL modeling with multi-cycled and pipelined components," in *Proc. of the Int. SoC Design Conf.*, Seoul, Korea, Oct. 2006, pp. 21–28.

[6] R. Sinha and H. Patel, "Abstract state machines as an intermediate representation for high-level synthesis," in *DATE*, Grenoble, France, Mar. 2011, pp. 1406–1411.

[7] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 11, no. 1, pp. 44–54, Jan.-Mar. 1994.