Hardware design space exploration using HercuLeS HLS

Nikolaos Kavvadias*
Ajax Compilers
Voutieridi 7 Rd
11525 Athens, Greece
nkavvadias@ajaxcompilers.com

Kostas Masselos†
Department of Computer Science and Technology
University of Peloponnese
22100 Tripoli, Greece
kmas@uop.gr

ABSTRACT

HercuLeS is an extensible high-level synthesis (HLS) environment. It removes significant human effort by automatically mapping algorithms to hardware, providing a valuable design assist to software-oriented developers. To enable accessibility and easiness of hardware design space exploration (DSE), HercuLeS overcomes limitations of known work: non-standard source languages, insufficient representations, maintenance difficulties, necessity of code templates, lack of usage paradigms and vendor-dependence. Specific aspects that are highlighted in this manuscript are: a) the innerworkings of the HercuLeS hardware compilation engine, b) manipulation of SSA (Static Single Assignment) form, c) automatic third-party IP integration, d) backend C code generation for compiled simulation, and e) an exemplary case of DSE. HercuLeS enables efficient hardware generation that can closely match the quality of results of a manually-developed implementation with much reduced human effort and time requirements.

Categories and Subject Descriptors

B.5.1 [**Hardware**]: Register-Transfer-Level Implementation-Design[Styles]; B.5.2 [**Hardware**]: Register-Transfer-Level ImplementationDesign Aids[Automatic Synthesis, Hardware Description Languages]; B.7.1 [**Hardware**]: Integrated CircuitsTypes and Design Styles[Algorithms implemented in hardware]

General Terms

Design, Languages, Performance

Keywords

high level synthesis, HLS, FPGA, application-specific integrated circuit, ASIC

*Cofounder and CEO.

†Holding advisory role at Ajax Compilers.

PCI 2013, September 19 - 21 2013, Thessaloniki, Greece

http://dx.doi.org/10.1145/2491845.2491857

1. INTRODUCTION

Current VLSI technology allows the design of sophisticated digital systems with ever-growing requirements in performance and power/energy consumption. Rapidly changing user demands, unprecedented applications, evolved existing or newly-introduced standards, shape the computational continuum.

It has long been observed that human designers' productivity does not escalate sufficiently enough to match the corresponding increase in chip complexity. Notably, the annual increase of chip complexity is 58%, while human designers' productivity increase is limited to 21% [12]. This technology-productivity gap is a significant obstacle in the industrial development of innovative products. A dramatic increase in designer productivity is only possible through the adoption and practicing of methodologies that raise the specification abstraction level, ingeniously hiding low-level, time-consuming, error-prone details. New EDA (Electronic Design Automation) methodologies aim to generate high-performance digital designs from high-level descriptions, a process called High-Level Synthesis (HLS) [29]. HLS [28] aims at eliminating human errors and shortening time-to-market. The input to this process is usually an algorithmic-level description, generating synthesizable register-transfer level (RTL) designs that can be implemented on FPGAs (Field-Programmable Gate Arrays) or ASICs (Application-Specific Integrated Circuits).

HLS approaches have been developed by academic groups, startups, established FPGA and EDA vendors. Still, there is need to tackle important shortcomings, inefficiencies and omissions such as: a) the devise and use of insufficient and inflexible intermediate representations (IRs), recording only partial information; b) difficulty in maintaining features and interfacing optimizations; c) mandating the use of code templates to obtain decent results; lack of easy to follow paradigms; d) use of closed formats and e) succumbing to vendor and technology dependence.

In this work, specific aspects of HercuLeS[1] are presented. HercuLeS enables a seamless user experience from algorithm to implementation. To achieve this, it confronts the aforementioned problems: a) HercuLeS uses the NAC IR [33] which is a bit-accurate typed-assembly language for whole program descriptions and supports the manipulation of a number of SSA-like (Static Single Assignment) forms; b) optimizations can be added as self-contained external modules upon a moderately-sized HLS kernel; c) HercuLeS does not rely on code templates since it uses a graph-based back-

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

end; d) open specifications such as Graphviz [9] and NAC are used throughout the HLS process, and e) the generated HDL code is completely vendor- and technology-independent. It is human-readable and allows for automatic third-party, IP integration through an open process.

The remainder of this paper is organised as follows. Section 2 overviews previous research on the subject. In Section 3, interesting aspects of HercuLeS are introduced. Classic SSA, pseudo-SSA and ϕ-preserving forms are discussed in Section 4. Backend C code generation issues are presented in Section 5. Section 6 presents efficient DSE with a well-known number-theoretical algorithm as the test vehicle. Section 7 evaluates HercuLeS in terms of speed and chip area optimization. Finally, Section 8 summarizes the paper.

2. RELATED WORK

EDA vendor HLS offerings include Vivado HLS [36], CatapultC [4], ImpulseC [11], Synphony HLS [18] and C-to-Silicon [1]. Vivado HLS accepts source input in C, C++ or SystemC and generates RTL hardware in VHDL [24] or Verilog HDL [23]. However, third-party IPs are not automatically integrated; instead, vendor-dependent cores are used. Generally, architectures generated by CatapultC and ImpulseC have increased communication overhead that cannot be alleviated in all cases. Synphony HLS and C-to-Silicon primarily target the ASIC community due to their very high price tags; evaluation versions for them are not available to the public. None of these tools expose information using open specifications; textual IRs are not accessible for processing and manipulation by third-party tools.

Some HLS tools restrict users to specific platforms [3,31]. User designs are bound to be utilized as PICO/ARM and Nios-II coprocessors, respectively. Despite that the convenience of GCC [19] is identified in [31], the actual input to the HLS engine is the low-level, machine-dependent RTL IR. C2H is block-oriented accepting a strict C subset, thus it is unable to process whole programs. LegUp [13] provides a rich environment for experimentation but produces low-level, vendor-specific HDL code.

Publicly released tools producing generic HDL include ROCCC [16], SPARK [17] and GAUT [6]. ROCCC [16] targets streamable C applications on a feed-forward pipeline. It is restricted to perfectly nested constant-bound loops. GAUT [6] accepts a C/C++ subset and user constraints (total latency, maximum clock period) to extract full parallelism. It is incapable of handling non-static loops. SPARK only handles loops with fixed constant iteration counts, rendering most designs infeasible.

Tools with web interface access include: C-to-Verilog [2], TransC [20] and HercuLeS[2]. C-to-Verilog [2] is an LLVM Verilog backend [14], a favorable approach due to the optimization capabilities of LLVM. However, it presents limitations in accessing local or global arrays within functions. TransC [20] supports streaming constructs for data exchange and process synchronization, however through non-standard C-like code requiring the user to significantly divert from C programming.

3. HERCULES BASICS

[2]`http://www.nkavvadias.com/cgi-bin/herc.cgi`

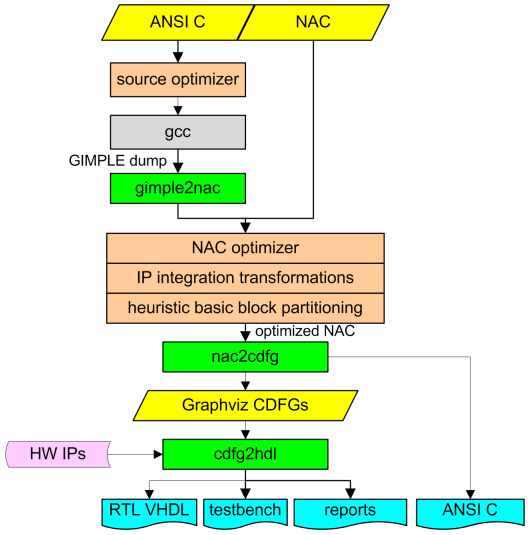


Figure 1: The HercuLeS flow.

HercuLeS automatically generates customized hardware as extended FSMDs (Finite-State Machines with Datapath) [29] in VHDL. Essentially, HercuLeS translates programs in the NAC IR to a collection of Graphviz CDFGs (Control-Data Flow Graphs) which are then synthesized to vendor-independent self-contained RTL VHDL. HercuLeS is also used for push-button synthesis of ANSI C code to VHDL.

Since aspects of HercuLeS have already been covered [32, 34], this work focuses on its DSE capabilities.

3.1 Overview

The basic steps in the HercuLeS flow are shown in Fig. 1. C code is passed to GCC for GIMPLE dump generation [8], following an external source-level optimizer. Textual GIMPLE is then processed by *gimple2nac*; alternatively the user could directly supply a NAC translation unit (TU) [33].

NAC operations specify a mapping from a set of n ordered inputs to m ordered outputs as follows: `o1, ..., om <= oper i1, ..., in;` where: `oper` specifies the IR-level instruction, `o1, ..., om` are the m outputs, and `i1, ..., in` the n inputs of the operation. A declared object is either a `globalvar` (a global scalar or array variable), `localvar` (a local), `in` or `out` (input or output procedure argument). The memory access model defines dedicated address spaces per array, so that both loads and stores require an explicit array identifier. An indexed load in C (`b = a[i];`) is translated as: `b <= load a, i;`, while an indexed store (`a[i] = b;`) as: `a <= store b, i;`. Procedures are non-atomic operations; `(y) <= sqrt(x);` computes $\lfloor\sqrt{x}\rfloor$.

Various optimizations can be applied at the NAC level; peephole transformations, if-conversion, and function call insertion to enable IP integration. Heuristic basic block partitioning avoids the introduction of excessive critical paths due to operation chaining. The core of HercuLeS comprises of a frontend (*nac2cdfg*) and a graph-based backend (*cdfg2hdl*). *nac2cdfg* is a translator from NAC to flat CDFGs represented in Graphviz [9]. *cdfg2hdl* is the actual synthesis kernel for automatic FSMD hardware from Graphviz CDFGs to VHDL and self-checking testbench generation.

nac2cdfg is used for parsing, analysis and CDFG extraction from NAC programs. Multiple approaches to SSA and SSA-like form construction are supported (Section 4). Data

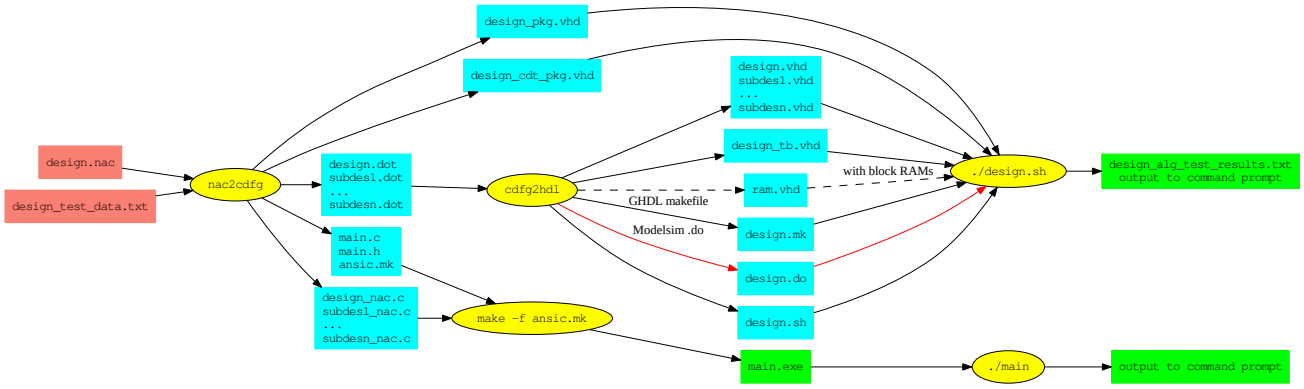


Figure 2: Innerworkings of HercuLeS.

flow analysis uses on-demand graph reachability checking. *cdfg2hdl* maps CDFGs to an extended FSMD MoC (Model of Computation) [29]. For scheduling operations to specific states, sequential, control-aware ASAP or ALAP scheduling can be used. ASAP and ALAP can be combined with fast operation chaining for better state workload balancing. Specifically, ALAP allows the user to set latency preferences and is a means for setting constraints, previously missing from HercuLeS.

An ANSI C backend allows for rapid algorithm prototyping and NAC verification. VHDL code can be simulated with GHDL [7] and Modelsim [15] and synthesized in Xilinx XST [22] using automatically generated scripts.

3.2 How it works

Fig. 2 gives a detailed view of HercuLeS operation. The user of HercuLeS must provide two input files:

- `design.nac`: A NAC program translation unit providing the entire application.
- `design_test_data.txt`: I/O reference values for use by the automatically-generated testbench.

Then, *nac2cdfg* generates several files:

- `design.dot`, `subdes1.dot`, ..., `subdesn.dot`: These files are Graphviz CDFGs for the root procedure (`design`) and all other procedures in the NAC program.
- `main.c`, `main.h`, `ansic.mk`: Files generated for running an ANSI C simulation. `ansic.mk` is an automatically-generated Makefile.
- `design_nac.c`, `subdes1_nac.c`, ..., `subdesn_nac.c`: ANSI C backend files providing C implementations of all procedures in the TU, generated directly from NAC. They are used in the C simulations.
- `design_pkg.vhd`: VHDL package for the FSMD components for all NAC procedures.
- `design_cdt_pkg.vhd`: VHDL package incorporating definitions of compound data types (arrays).

Following this, there exist two possible flows; one for the generation and simulation of synthesizable RTL VHDL for the NAC program, and one for C simulation. The C simulation flow proceeds by invoking the `ansic.mk` makefile. This produces a `main` executable specification. Then, the executable is run.

The VHDL flow involves processing all CDFG (`.dot`) files by *cdfg2hdl*, the actual backend tool of HercuLeS. *cdfg2hdl* generates several files:

Table 1: FSMD I/O interface.

Signal	Direction	Description
`clk`	*I*	signal from external clocking source
`reset`	*I*	asynchronous (or synchronous) reset
`start`	*I*	enable computation
`din`	*I*	data inputs (generally, multiple)
`dout`	*O*	data outputs (generally, multiple)
`ready`	*O*	the block is ready to accept new input
`valid`	*O*	asserted when a certain data output port is streamed-out from the block (generally it is a vector)
`done`	*O*	end of computation for the block

- `design.vhd`, `subdes1.vhd`, ..., `subdesn.vhd`: Synthesizable RTL VHDL for the root procedure and all other procedures in the NAC program.
- `ram.vhd`: VHDL model of a dual-port synchronous read RAM for block RAM inference. It is only used if block RAM mapping is enabled.
- `design_tb.vhd`: The self-checking testbench.
- `design.mk`: Makefile for running a GHDL simulation.
- `design.do`: `.do` macro file for a Modelsim simulation.
- `design.sh`: Bash shell script initiating either a GHDL or Modelsim simulation.

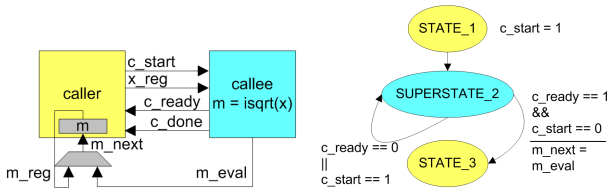
Finally, the `design.sh` script is run from the command line and this produces a text file (`design_alg_test_results.txt`) providing diagnostic output from a simulation run. `print` NAC operations can be used for tracing since they map to VHDL `assert` constructs or a C standard library `printf`. Also, a VCD or GHW [7] waveform file can be generated for viewing with GTKwave [10].

3.3 Extended FSMDs

The FSMDs of our approach use fully-synchronous conventions and register all their outputs [27]. The generated FSMDs are generalized FSMs introducing embedded actions, with: a) support of array input, output and streaming I/O ports, b) communication with embedded block and distributed LUT memories, c) latency-insensitive local interface between caller and callee FSMDs, and d) interfacing to external IP blocks. I/O port usage is summarized in Table 1.

The FSMDs use $n + 2$ states, where n is the number of required computational states. The two overhead states represent CDFG source/sink nodes. One possible optimization is merging the sink state with its immediate predecessors.

3.3.1 *Hierarchical FSMDs*



(a) Interface. (b) State transitions.

Figure 3: Caller-callee communication.

A two-state protocol describes proper communication between caller and callee FSMDs. The first state prepares the communication, while the latter is 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 `⋆_next` signals and registered values are read from `⋆_reg` signals. To avoid the problem of multiple signal drivers, callee procedure instances produce `⋆_eval` data outputs that can then be connected to register inputs by hardwiring to the `⋆_next` signal. Fig. 3 illustrates the established interface and state transitions that control a call (`(m) <= isqrt(x);`) to integer square root evaluation.

`STATE_1` sets up the callee instance. Callee operation takes place in `SUPERSTATE_2`. When the callee terminates, `ready` is raised. Since callee `start` is kept low, output data can be transferred to the m register via its `m_next` input port. Control then is handed over to `STATE_3`. The callee instance follows the established FSMD interface, reading `x_reg` and producing its result in `m_eval`.

3.4 IP integration

HercuLeS allows for automatic IP integration given that the user supplies builtin functionalities. To illustrate this approach, IP blocks for signed/unsigned addition/subtraction, multiplication, division and remainder have been designed. The HercuLeS flow user is able to import and use an owned IP by the following process:

1. Implement IP with expected interface and place in proper subdirectory
2. Add corresponding entry in a textual database
3. Use TXL transformations [21] for replacing an operator use by a black-box function call via a script
4. A list of black box functions is generated
5. HercuLeS automatically creates a hierarchical FSMD with the requested callee(s).

Fig. 4 illustrates the combined TXL/C approach. The first two steps apply preprocessing for splitting `localvar` declarations and removing those that are redundant or unused. Then, they are localized and subsequently procedure calls to black-box functions are introduced. These routines are the actual builtin functions. If the corresponding builtins are listed in the IP database, an interface-compatible VHDL implementation to HercuLeS caller FSMDs is assumed. Then, *cdfg2hdl* automatically handles interface generation and component instantiation in the HDL description for the caller FSMD description. In addition, simulation and synthesis scripts already account for the IP HDL files.

This approach is also valid for floating-point computation, currently using IPs generated by FloPoCo [5]. In addition,

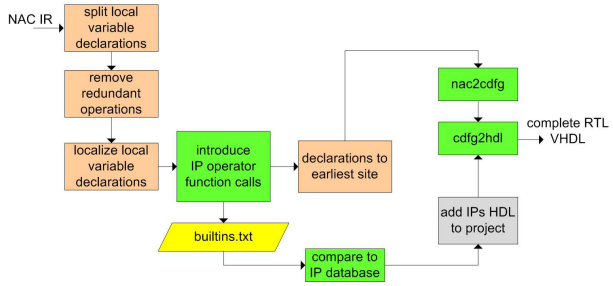


Figure 4: Automatic IP integration in HercuLeS.

both pipelined and multi-cycle third-party components are supported.

4. SSA ALGORITHMS

The general scheme for these methods consists of a series of passes for variable numbering, ϕ-insertion, ϕ-minimization, and dead code elimination. An out-of-SSA conversion pass replaces ϕ statements at a flow-target basic block (BB) by multiple copies (`mov` operations) in its flow-predecessor BBs.

4.1 SSA construction

Appel presents a simple approach for variable renaming and ϕ-function insertion in two separate phases (algorithm S) [25]. In the first phase, every variable is split at BB boundaries, while in the second phase ϕ-functions are placed for each variable in each BB. Variable versions are actually preassigned in constant time and reflect a specific BB ordering (e.g. depth-first search). Thus, variable versioning starts from a positive integer n, equal to the number of BBs in the given CFG (Control Flow Graph).

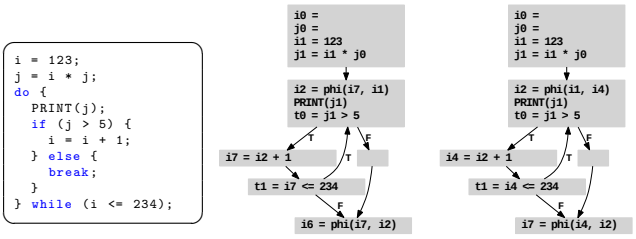
For true SSA construction, the algorithm of Aycock and Horspool (algorithm H) [26] can be used as well, which bares some similarities to Appel's approach. However, algorithm H does not predetermine variable versions at control-flow joins but accounts ϕ-functions the same way as actual computations visible in the original CFG. A detailed view of implementations for both algorithms S and H can be found in [33]. A faster algorithm for pseudo-SSA construction is also available in HercuLeS, named algorithm P.

4.2 Pseudo-SSA construction

In this algorithm, def- and use- chains (which are tracked by certain arrays or bit-vectors) are initialized prior each BB entry. Pseudo-SSA establishes that only true data dependencies are visible in a given program, however, information that would be useful in most data flow analyses cannot be maintained.

A basic difference to algorithm H, is that pseudo-SSA (algorithm P) clears read and write tracking counters for each variable per BB. With the help of a bit-vector, when a local variable has not been redefined, it simply copies it to the versioned operand, otherwise it properly constructs a new versioned operand. When the write counter must be incremented, the corresponding entry in the bit-vector is set.

In algorithm P, ϕ insertion is replaced by a process termed as "copy restoration". In copy restoration, ϕ-functions are not inserted at all. This process performs the restoration of the original variable names prior to the exit of each BB by inserting the appropriate `mov` statements (copies restoring the unversioned variables), since local versioned names are only visible within each specific BB. Unique SSA variable



(a) Input C code. (b) SSA (alg. S). (c) SSA (alg. H).

Figure 5: Example C source code and SSA form.

names can be established upon choice by concatenating the procedure identification number and BB enumeration with variable versions. Unique variable renaming can also be applied externally via a TXL transformation pass.

4.3 SSA destruction

The SSA destruction phase regards reinstating the initial non-SSA NAC form without ϕ-functions at control flow join points. In this form, the NAC representation can be processed by the `nac2c` backend for C code generation to be used for profiling via fast compiled simulation.

4.3.1 *Classic out-of-SSA conversion*

In classic out-of-SSA conversion, each ϕ-function in a control-flow target BB is replaced by a `mov` statement at each control-flow source BB. This transformation always yields a proper non-SSA NAC representation, however at the expense of larger code.

4.3.2 *Preserving PHI functions*

It is possible to preserve ϕ-functions if an out-of-SSA conversion pass is not used. For such statements to be interpretable, additional information is needed to be passed to NAC ϕ statements in the form of either additional source operands or as tags. This information regards the label or enumeration of the corresponding predecessor BB, where each ϕ source operand has been defined. Within the context of the backend C code generation, executable ϕs appear as `switch-case` statements, where the condition variable is `prevbb`, a counter that records in run-time the predecessor BB in the current execution path.

4.4 A simple example

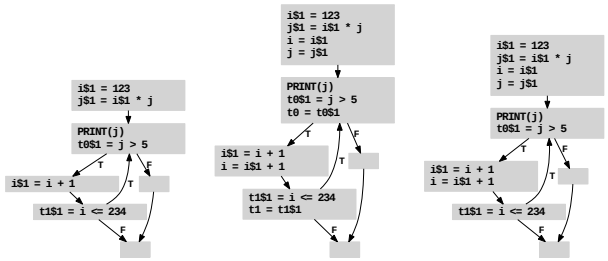
The motivating example of a simple C-like code from [26] is shown in Fig. 5(a). Valid optimized SSA for algorithms S and H is shown in Fig. 5(b) and Fig. 5(c), respectively. There, algorithm H presents only lexicographic and not semantic differences to S. Both algorithms achieve the generation of minimal SSA involving two ϕ statements.

Fig. 6 illustrates pseudo-SSA form construction. In algorithm P, variable versions, as can be seen, are not unique when the entire CFG is accounted.

Fig. 7 illustrates out-of-SSA NAC for the motivating example. Only algorithms S and H are shown since out-of-SSA is not meaningful for P.

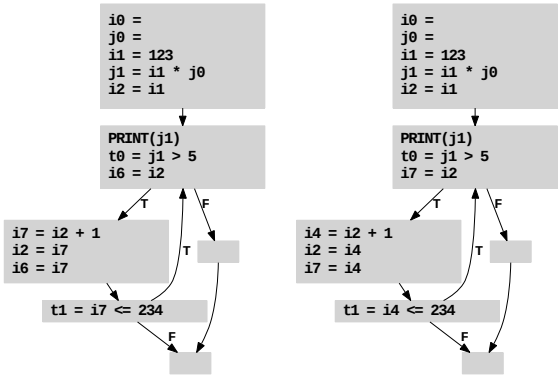
In Fig. 8, the corresponding NAC representation is shown for algorithm H, alongside the actual C code generated by `nac2c`.

5. NAC TO C CODE GENERATION



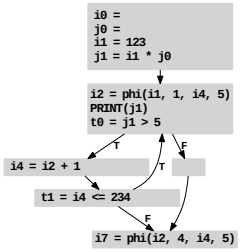
(a) Variable numbering. (b) Copy restoration. (c) Final form.

Figure 6: Step-by-step pseudo-SSA construction.



(a) Algorithm S. (b) Algorithm H.

Figure 7: Out-of-SSA representation.



(a) CFG.

```
BB1:
  i$1 = 123;
  j$1 = i$1 * j$0;
  prevbb = 1; goto BB2;
BB2:
  switch (prevbb) {
    case 1: i$2 = i$1; break;
    case 5: i$2 = i$4; break;
    default: break;}
  printf("j$1 = %08x\n", j$1);
  if (j$1 > 5) t0$3 = 1;
  else t0$3 = 0;
  prevbb = 2;
  if (t0$3 == 1) {goto BB3;}
  else {goto BB4;}
BB3:
  i$4 = i$2 + 1;
  prevbb = 3; goto BB5;
BB4:
  prevbb = 4; goto BB6;
BB5:
  if (i$4 <= 234) t1$6 = 1;
  else t1$6 = 0;
  prevbb = 5;
  if (t1$6 == 1) {goto BB2;}
  else {goto BB6;}
BB6:
  switch (prevbb) {
    case 4: i$7 = i$2; break;
    case 5: i$7 = i$4; break;
    default: break;}
```

(b) Generated C code.

Figure 8: Out of SSA conversion with PHI function interpretation using Algorithm H.

The main process involved in NAC to C code generation regards the mapping of NAC operations to C statements. Table 2 illustrates the corresponding code generation patterns, simplified where necessary. Data type mappings are not shown since they are trivial (e.g. `u16` corresponds to `unsigned short int` and `f1.11.52` to `double`).

`relop` denotes an ANSI C relational operator, and the `zz` suffix is correspondingly `eq, ne, ne, lt, gt, le` or `ge`.

Table 2: *nac2c* code generation patterns.

Operation	NAC statement	C code template
No-operation	`nop;`	`/* NOP */;`
Move, load constant/address	`t <= op u;`	`t = u;`
Logical not	`t <= not u;`	`t = ~(u);`
Logical and	`t <= and u, v;`	`t = u & v;`
Logical or	`t <= ior u, v;`	`t = u | v;`
Logical xor	`t <= xor u, v;`	`t = u ^ v;`
Addition	`t <= add u, v;`	`t = u + v;`
Doubling	`t <= dbl u;`	`t = u + u;`
Subtraction	`t <= sub u, v;`	`t = u - v;`
Negation	`t <= neg u;`	float, double: `t = -(u);` integer: `t = ~(u) + 1;`
Multiply	`t <= mul u, v;`	`type=datatype(t); t=(type)u * (type)v;`
Squaring	`t <= sqr u;`	`type=datatype(t); t=(type)u * (type)u;`
Division	`t <= div u, v;`	`t = u / v;`
Remainder	`t <= rem u, v;`	`t = u % v;`
Modulo	`t <= mod u, v;`	`t = (v<0)?-(u%v):(u%v);`
Shift left	`t <= shl u, v;`	`t = u << v;`
Shift right	`t <= shr u, v;`	`t = u >> v;`
Right rotate	`t <= rotr u, v;`	`w = bitwidth(u); t = (u>>v)|(u<<(w-v));`
Left rotate	`t <= rotl u, v;`	`w = bitwidth(u); t = (u<<v)|(u>>(w-v));`
Multiplexing	`t <= muxzz u, v, w, x;`	`t = (u relop v) ? w : x;`
Set on condition	`t <= szz u, v;`	`t = (u relop v) ? 1 : 0;`
Absolute value	`t <= abs u;`	`t = (u < 0) ? -(u): u;`
Minimum	`t <= min u, v;`	`t = (u < v) ? u : v;`
Maximum	`t <= max u, v;`	`t = (u > v) ? u : v;`
Load	`t <= load u, v;`	`t = u[v];`
Store	`t <= store u, v;`	`t[v] = u;`
Self copy	`t <= self u;`	`t[u] = u;`
Sign-extension	`t <= sxt u;`	`type=datatype(t); t = (type)(u);`
Zero-extension	`t <= zxt u;`	`type=datatype(t); t = (type)(u);`
Truncation	`t <= trunc u;`	floating-point: `type=datatype(t); t = (type)(u);` integer: `w=bitwidth(u); bitmask = 2**u-1; t = u & bitmask;`
Integer to float	`t <= i2s u;`	`t = (float)(u);`
Integer to double	`t <= i2d u;`	`t = (double)(u);`
Floating-point to integer	`t <= [s|d]2i u;`	`type = datatype(t); t = (type)(u);`
Unconditional jump	`tlab <= jmpun;`	`if (keep_ssa) then prevbb = current BB; endif goto tlab;`
Conditional jump	`tlabt, tlabf <= jmpzz u, v;`	`if (keep_ssa) then prevbb = current BB; endif if (u relop v) {goto tlabt;} else {goto tlabf;}`
Phi statement	`t <= phi u1, <num1>,... un, <numn>;`	`switch (prevbb) { iterate the input opnd list [u1,un]; get input opnd ui; case defix: t = ui; break; end iterate default: break;}`
Print variable	`print u;`	`w=bitwidth(u); n=W/4; type=datatype(u); PRINT(u,type,w);`

`defix` is the definition BB for `ui`. Statements `dbl` (doubling), `sqr` (squaring) and `self` (copying data to a homonymous address) are used when an `add`, `mul` or `store`, respectively, has two identical source variables. This is required since C code generation uses the same data structures that are used for CDFG generation. It has been chosen that CDFGs are simple graphs and not hypergraphs, thus there can be no multiple edges between a node pair. All possible cases of identical source operands can be eliminated via appropriate transformations using a peephole transformation pass.

6. DSE CASE STUDY: FIBONACCI SERIES

HLS tools should enable the efficient architectural design space exploration of possible implementations. As a case study, we examine the well-known Fibonacci series, defined as:

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

using HercuLeS high-level synthesis[3].

[3] `http://www.nkavvadias.com/hercules/`

```
uint32 fibo(uint32 x) {
  uint32 f0=0, f1=1;
  k = 2;
  do {
    k = k + 1;
    f1 = f1 + f0;
    f0 = f1 - f0;
  } while (k <= x);
  return (f1);
}
```

(a) Algorithm A.

```
uint32 fibo(uint32 x) {
  uint32 f0=0, f1=1, k, f;
  k = 2;
  do {
    k = k + 1;
    f = f1 + f0;
    f0 = f1;
    f1 = f;
  } while (k <= x);
  return (f);
}
```

(b) Algorithm B.

```
uint32 fibo(uint32 n) {
  uint32 f0=1, f1=1;
  k = 2;
  while (k <= n) {
    k = k + 1;
    f0 = f1 + f0;
    SWAP(f0, f1);
  }
  return (f0);
}
```

(c) Algorithm C.

Figure 9: Iterative algorithms for Fibonacci series.

Table 3: Machine cycles for Fibonacci series hardware.

Design	Cycles	Design	Cycles	Design	Cycles
A0	n	*B0*	n	*C0*	$2n-1$
A1	$4n+3$	*B1*	$5n+2$	*C1*	$7n+1$
A2	$4n+2$	*B2*	$4n+2$	*C2*	$7n$
A3-A5	$n+2$	*B3-B5*	$n+2$	*C3-C5*	$2(n+1)$

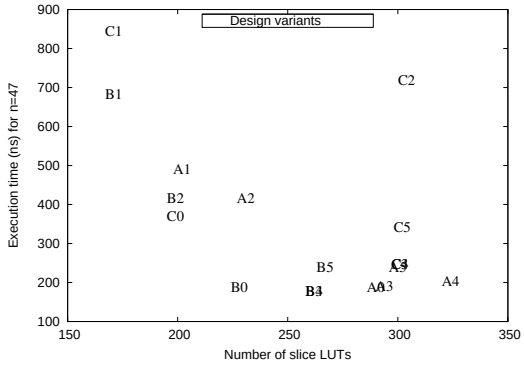
An implementation closely following the recursive definition would require passing arguments through a stack [35]. A non-recursive approach can result in faster and more efficient hardware. Fig. 9 illustrates three iterative algorithms for computing $F(n)$. Algorithm A uses addition and subtraction in the main loop, B uses only addition but needs an additional temporary (f), and C again uses a single addition, does not make use of f, and involves a register swap. The swap can be implemented either in-situ using repeated XORing or by register moves requiring temporary storage. A closed-form solution exists but uses real arithmetic and may only competitive for a large number of terms.

HercuLeS can be used for exploring different choices of frontend translation to the NAC IR as well as hardware optimization. For this purpose, we exercised five different option sets for each algorithm variant: sequential scheduling (O1), ASAP scheduling using SSA (O2), ASAP with operation chaining (O3), O3 with pseudo-SSA construction (O4) and O3 with preserving ϕ functions (O5).

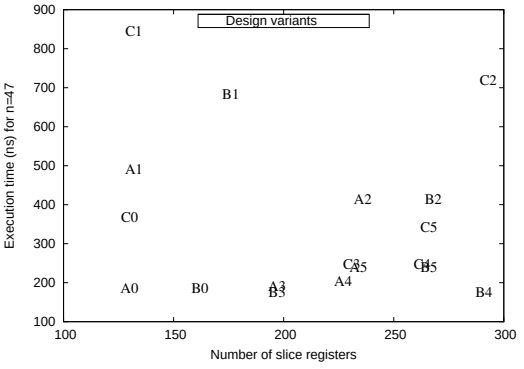
For comparison purposes, carefully hand-coded designs were also developed for each variant. A highly-optimized circuit as such computes a new term of the series per clock cycle, thus the required computation time for $F(n)$ expressed in machine cycles is equal to n. Table 3 depicts analytical expressions for calculating the number of machine cycles for each design point. Machine cycles have been deduced by simulation of the respective HDL designs automatically generated by HercuLeS. Human expert designs are denoted as A0, B0, and C0, respectively. Overall, a total of 18 design points, each one corresponding to different design trade-offs have been defined.

Interestingly, the benefit of using ASAP (O2) over sequential (O1) scheduling is not significant, which is easily explained by the data dependencies of the algorithm. Further, HercuLeS can closely match the result of a human expert; optimization schemes O3-O5 produce hardware that only needs $n+2$ computation cycles, compared to n for the reference circuits. For high values of n, the difference is negligible. The two-cycle difference is due to design choices of the human expert: a) initializing $f0$, $f1$ and k in the FSMD entry state, and b) passing the output data argument without use of an intermediate register. Both techniques can be supported by HercuLeS with minor additions to *cdfg2hdl*.

For quantitative assessment of the design points, we ob-



(a) Execution time vs. LUTs.



(b) Execution time vs. Regs.

Figure 10: The architectural design space of Fibonacci series hardware.

tained propagation delay (t_{PD}), maximum operating frequency (MOF) and area metrics. All designs were synthesized on the XC6VLX75T Xilinx Virtex-6 device with Xilinx Webpack ISE 12.3i. Total execution time is computed by $t_{PD} \times cycles$.

Fig. 10 illustrates time-versus-area scatter plots for number of slice LUTs and registers. Each design point is represented by a label. In both cases, better results are placed near the bottom-left corner of the graph (small execution time and area). In this sense, B0 and C0 are Pareto-optimal points in Fig. 10(a). B2, B3, and B4 are potential candidates for the optimal realization produced by HercuLeS. In Fig. 10(b) A0 is optimal. Designs A1 and B3 are the Pareto-optimal designs generated by HercuLeS.

Another implied, yet important, trade-off here is the design concept-to-implementation time. To provide a consistent measure, a human developer with very high expertise required 2.5h for the design, testing and synthesis of A0, B0 and C0. On the contrary, using HercuLeS, HDL generation, GHDL simulation [7] and XST logic synthesis (using scripts) required 629 sec on a low-resource Intel Centrino Duo laptop. Automatically generating the 15 different architectures in VHDL took about 35 sec.

As can be noticed, for the execution time metric, calculating $F(47)$ was used. This is the largest Fibonacci series term that can be represented in 32 bits. The HercuLeS backend is able to generate hardware architectures that use arbitrary-precision arithmetic. The definition of ANSI/ISO C does not support multi-precision integers as primitive data types. Fig. 11 shows reference source code for a multi-precision version of variant B using the fgmp library [30] compared to the equivalent NAC IR. In this case, the multi-precision C version can only be used a guideline; the HercuLeS frontend does not yet support translation to NAC. However, arbitrary-sized integer data types are supported by the HercuLeS backend. For instance, a 1024-bit circuit can compute up to the 1476-th term of the series.

```
MP_INT fibofast2(uint32 n) {
  MP_INT f0, f1, f;
  unsigned int k;
  mpz_init_set_ui(&f0,0);
  mpz_init_set_ui(&f1,1);
  mpz_init(&f);
  k = 2;
  do {
    k = k + 1;
    mpz_add(&f,&f1,&f0);
    mpz_set(&f0, &f1);
    mpz_set(&f1, &f);
  } while (k <= n);
  mpz_clear(&f0);
  mpz_clear(&f1);
  return (f);
}
```

(a) ANSI C code.

```
procedure fibo(in u32 n,
  out u1024 outp)
{
  localvar u32 k;
  localvar u1024 f0, f1, f;
LL0:
  f0 <= ldc 0;
  f1 <= ldc 1;
  k <= ldc 2;
LL1:
  k <= add k, 1;
  f <= add f1, f0;
  f0 <= mov f1;
  f1 <= mov f;
  LL1, LL2 <= jmple k, n;
LL2:
  outp <= mov f;
}
```

(b) NAC IR (1024-bit).

Figure 11: Multi-precision versions of variant B.

7. EVALUATION

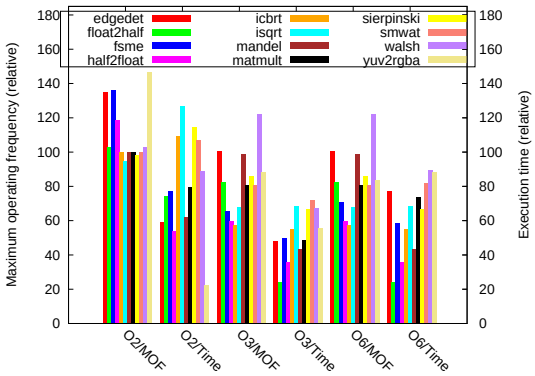
Quantitative comparisons and exploration scenarios were also obtained for selected generic applications. These include edge detection (*edgedet*), conversion among 16-bit floats and integers (*float2half*, *half2float*), full-search motion estimation (*fsme*), fixed-point, integer and cubic square root approximations (*fixsqrt*, *isqrt*, *icbrt*), Mandelbrot fractal (*mandel*), Sierpinski gasket (*sierpinski*), matrix multiplication (*matmult*), the Smith-Waterman kernel (*smwat*), 2D Walsh transform (*walsh*) and color space conversion (*yuv2rgba*). For each case, optimization scenarios O1, O2 and O3 were investigated, along with O6 defined as O3 with mapping all arrays to embedded, synchronous read, block RAMs. With the exception of O6, all other scenarios map storage resources to asynchronous read LUT RAM.

7.1 Speed and area metrics

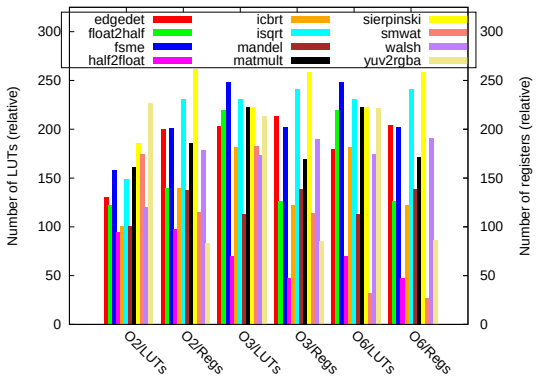
First, MOF and total execution time are evaluated. Time-related metrics are shown in Fig. 12(a) while number of LUTs and registers are shown in Fig. 12(b). Both figures use relative values. As a reference for 100%, O1 is used.

Average computation time is reduced by 44.3% when comparing O1 to O3. BRAMs impose a fixed cycle latency for synchronous readout as expected, which limits this gain to about 11.7% for O6. Computer arithmetic problems such as *float2half* and *half2float* achieve up to 4× reduction in execution time. Memory-intensive benchmarks (*matmult, smwat, walsh*) present lesser opportunities due to the memory accesses activity interfering with chaining. All benchmarks achieve MOFs in the range of 119-450MHz.

Regarding FPGA area, O1 generates smaller hardware in terms of LUTs and registers. With O3, LUT and register demand are increased by 90.1% and 59.1%, compared to O1, a price paid for much higher speed. However, it is more reasonable to compare O2, O3 and O6 which all prerequisite SSA form. Then, O3 introduces the highest LUT requirements, while O2 the highest register demand. Registers are reduced by 17.5% among O2 and O6. Also, O6 reduces the LUT demand in O3 by 14%.



(a) MOF and total execution time.



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

Figure 12: Speed and area metrics.

8. CONCLUSION

HercuLeS delivers a contemporary HLS environment that can be comfortably used for algorithm acceleration by predominantly software-oriented engineers. Even by using a primarily push-button, black-box approach, efficient exploration of the hardware architecture design space is possible so that the developer can select an appropriate design point that effectively complies with user-defined criteria. Apart from this, the innerworkings of HercuLeS, automatic IP integration, SSA forms for hardware compilation, and elements of the backend C code generator were presented in detail.

Future work involves additional optimizations such as loop pipelining, recursion removal as well as supporting recursive structures in hardware, resource sharing and on the frontend side compiling numerical analysis languages to hardware.

9. REFERENCES

[1] C-to-Silicon. http://www.cadence.com/products/sd/-silicon_compiler/pages/default.aspx.

[2] C-to-Verilog. http://www.c-to-verilog.com.

[3] C2H. http://www.altera.com/products/ip/-processors/nios2/tools/c2h/ni2-c2h.html.

[4] CatapultC. http://calypto.com/en/products/catapult/overview/.

[5] FloPoCo. http://flopoco.gforge.inria.fr/.

[6] GAUT. http://www-labsticc.univ-ubs.fr/www-gaut/.

[7] GHDL. http://ghdl.free.fr.

[8] GIMPLE. http://gcc.gnu.org/wiki/GIMPLE.

[9] Graphviz. http://www.graphviz.org.

[10] GTKwave. http://sourceforge.net/projects/gtkwave.

[11] ImpulseC. http://www.acceleratedtechnologies.com.

[12] ITRS. http://www.itrs.net/reports.html.

[13] LegUp. http://www.legup.org.

[14] LLVM.

[15] Modelsim. http://www.model.com.

[16] ROCCC. http://www.jacquardcomputing.com/roccc/.

[17] SPARK. http://mesl.ucsd.edu/spark/.

[18] Synphony HLS. http://www.synopsys.com/Tools/-SLD/HLS/Pages/default.aspx.

[19] The GNU Compiler Collection homepage. http://gcc.gnu.org.

[20] TransC. http://cgi.tu-harburg.de/~ti6hm/.

[21] Txl programming language homepage. http://www.txl.ca.

[22] Xilinx. http://www.xilinx.com.

[23] *IEEE 1364-2005 Standard for Verilog Hardware Description Language*, Apr. 2006.

[24] *IEEE 1076-2008 Standard VHDL Language Reference Manual*, Jan. 2009.

[25] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr. 1998.

[26] J. Aycock and N. Horspool. Simple generation of static single assignment form. In *Proc. 9th Int. Conf. in Compiler Construction*, pages 110–125, 2000.

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

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

[29] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(1):44–54, Jan.-Mar. 1994.

[30] M. Henderson. fgmp: Free/public-domain MP library. http://ftp.ee.netbsd.org/pub/pkgsrc/packages/-NetBSD/sparc/5.1/math/.

[31] G. N. T. Huong and S. W. Kim. GCC2Verilog: Compiler toolset for complete translation of C programming language into Verilog HDL. *ETRI Journal*, 33(5):731–740, Oct. 2011.

[32] N. Kavvadias, V. Giannakopoulou, and K. Masselos. FSMD-based hardware accelerators for FPGAs. In D. K. Tanaka, editor, *Embedded Systems - Theory and Design Methodology*, pages 157–160. InTech, Mar. 2012.

[33] 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)*, pages 351–354, Las Vegas, Nevada, USA, Jul. 2011.

[34] N. Kavvadias and K. Masselos. Automated synthesis of FSMD-based accelerators for hardware compilation. In *Proc. of the 2012 IEEE 23rd Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 157–160, Delft, The Netherlands, Jul. 2012.

[35] V. Sklyarov. FPGA-based implementation of recursive algorithms. *Microprocessors and Microsystems*, 28(5-6):197–211, 2004.

[36] Xilinx. Vivado ESL Design. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm.