Source and IR-level optimizations in the HercuLeS high-level synthesis tool

Nikolaos Kavvadias

Ajax Compilers,
Voutieridi 7 Rd, 11525 Athens, Greece
E-mail: nkavvadias@ajaxcompilers.com
*Corresponding author

Kostas Masselos

University of Peloponnese,
Department of Informatics and Telecommunications,
Terma Karaiskaki, 22100 Tripoli, Greece
E-mail: kmas@uop.gr

Abstract: HercuLeS is an extensible high-level synthesis environment for automatically mapping algorithms to hardware. It overcomes limitations of known work: insufficient representations, maintenance difficulties, necessity of code templates, lack of usage paradigms and vendor-dependence. Aspects that are highlighted include automatic IP integration and especially source- and intermediate-level optimizing transformations.

In this context, we present transformational patterns for loop and if-conversion optimizations. Further, we focus on constant multiplication and division by proposing a suitable scheme for their straightforward and decoupled utilization in user applications. It is shown that loop optimizations provide benefits of up to 32% in cycle performance, while if-conversion delivers an average improvement of 6.5%. By applying arithmetic optimizations, a 3.3-5.9$\times$ speedup over sequential implementations is achieved. It is also shown that HercuLeS is highly competitive to state-of-the-art commercial tools.

Keywords: Hardware; Integrated Circuits; FPGAs; Field-Programmable Gate Arrays; RTL; Register Transfer Level; HLS; High-level synthesis; Optimization.

Reference to this paper should be made as follows: Nikolaos Kavvadias, Kostas Masselos (xxxx) ‘Source and IR-level optimizations in the HercuLeS high-level synthesis tool’, *Int. J. Innovation and Regional Development*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Nikolaos Kavvadias received the B.Sc. degree in physics and M.Sc. degree in electronic physics from the Aristotle University of Thessaloniki, Greece in 1999 and 2002, respectively. In 2008, he received his Ph.D. degree on custom processor design methodologies from the same department. From 2008 to 2012 he was a lecturer at the Department of Computer Science and Technology of the University of Peloponnese, Greece. Since 2012 he is the co-founding CEO of Ajax Compilers. His research interests include high-level synthesis, application-specific/embedded processors and compilation techniques.

Kostas Masselos received the degree in electrical engineering from the University of Patras (UPAT), Patras, Greece, in 1994, the M.Sc. degree in VLSI systems engineering from the Institute of Science and Technology, University of Manchester, Manchester, U.K., in 1996, and the Ph.D. degree from UPAT in 2000. Since 2006, he has been with the Department of Computer Science and Technology, University of Peloponnese, Tripoli, Greece, currently at the rank of Professor, and a Visiting Lecturer with Imperial College London. His research interests include compiler optimizations, high-level synthesis, high-level power optimization, FPGAs and reconfigurable hardware, and efficient implementations of DSP algorithms.

This manuscript is an extended and significantly altered version of a 2013 conference publication.

1 Introduction

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% [11]. 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 in generating 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 ASICs (Application-Specific Integrated Circuits) or FPGAs (Field-Programmable Gate Arrays).

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); 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, the HercuLeS approach [40] taken to efficiently solve all these longstanding problems is presented. In contrast to the vast majority of competitive technologies, HercuLeS confronts all of the aforementioned problems: a) it uses the NAC IR [31] which is an exportable and extensible bit-accurate typed-assembly language for whole program descriptions; 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 backend; 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. HercuLeS-generated code does not rely on vendor-specific features such as the presence of patented architectural features; embedded memory and computational blocks that would enforce vendor tie-in. It is human-readable and allows for automatic third-party IP integration through a truly open process.

The remainder of this paper is organized as follows. Section 2 overviews previous research on the subject. Our motivations and contributions behind this work are established in Section 3. In Section 4, interesting aspects of HercuLeS are introduced. Sections 5 and 6 discuss C- and NAC-level transformation paradigms and their implementation. Section 7 provides performance metrics that illustrate their effectiveness despite being disjointed from the main compilation flow, while Section 8 compares HercuLeS against Vivado HLS [32] which is a popular flow for FPGA HLS. Finally, Section 9 summarizes the paper.

2 Related work

HLS offerings from EDA vendors include Vivado HLS [32], CatapultC [3], ImpulseC [10], Synphony HLS [19] and C-to-Silicon [1]. Vivado HLS accepts source input in C, C++ or SystemC and generates RTL hardware in VHDL [25] or Verilog HDL [24]. However, third-party IPs are not automatically integrated and 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 target specific platforms, for instance user designs are utilized as PICO/ARM coprocessors in [30]. Despite the convenience of GCC [20] is identified in [30], the actual input to the HLS engine is the low-level, machine-dependent RTL IR. LegUp [12, 13] provides a rich environment for experimentation but produces low-level, vendor-specific HDL code. DWARV [4] is a hardware compiler with a CoSy-based frontend [5] that can generate a reconfigurable processor-based implementation.

Publicly released tools producing generic HDL include ROCCC [16], SPARK [17, 18] and GAUT [6]. ROCCC [16] targets streaming 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 unfeasible.

Tools with web interface access or recent demo versions include: C-to-Verilog [2], TransC [21, 22] and HercuLeS. C-to-Verilog [2] is an LLVM [14] Verilog backend presenting limitations in accessing local or global arrays within functions. TransC [21] 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.

2.1 Summary and comparison of HLS systems

To establish a comparative summary of state-of-the-art HLS, we attempt to quantify multiple performance evaluation criteria:

- **A**bstraction: abstraction level supported for the source language.
- **T**ypes: richness and range of accepted data types.
- **E**xplore: rapid design space exploration capabilities.
- **V**erify: verification aspects such as automatic generation of self-checking testbenches, assertion insertion, code coverage.
- **R**esults: quality of results in terms of speed, area and power.

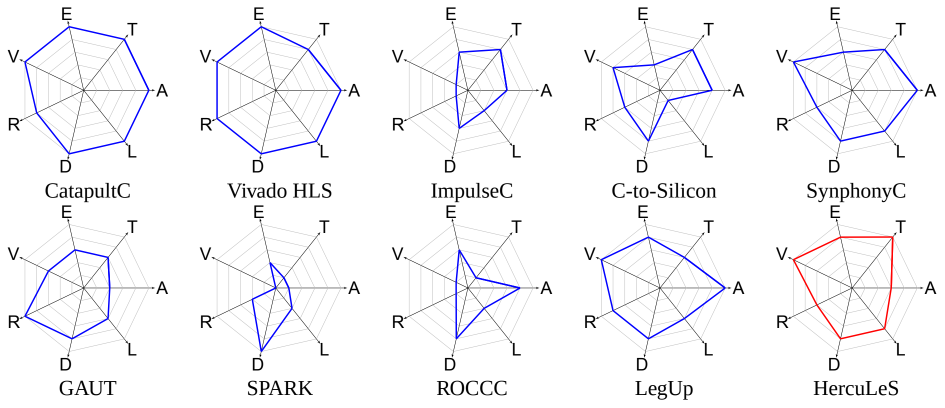


Figure 1 HLS tools comparison (**A**bstraction; **T**ypes; **E**xplore; **V**erify; **R**esults; **D**ocumentation; **L**earning).

- **D**ocumentation: available tutorial, user-oriented and developer-oriented materials.
- **L**earning: estimated learning curve.

Fig. 1 uses Kiviat graphs to visualize the comparative analysis of the tools across multiple independent dimensions. Certain tools have been omitted due to lack of sufficient information for the evaluation process, source language frontend issues or for not generating correct RTL code.

HercuLeS achieves close results (wins and losses) to Vivado HLS, which is known to compare well with the best-known open-source HLS tool, LegUp. Mentor CatapultC and SynphonyC are also considered strong offerings. ROCCC is a mature tool with version 1.0 based on the SUIF compiler infrastructure and 2.0 on LLVM but presents a steep learning curve. GAUT and SPARK fall behind, mainly due to lower ANSI C compatibility, and their runtime instability. GAUT achieves exceptional performance for DSP code using loop-level pipelining but does not perform well on control-oriented code.

3 Motivation and contribution

Commercial closed-source HLS tools as Vivado HLS [32] do not provide for user interfacing of source language frontends, custom analysis or optimization passes, and target architecture backends. At the same time, academic open-source HLS tools as LegUp [12] offer a transparent approach to user involvement, however this is only practical through the use of their established data structures and API (Application Programming Interface). No additional interaction points are established for exporting and importing to and from third-party tools, either open-source or not, such as external code analyzers, optimizers, annotators and hardware IP integrators.

In contrast to competitor tools, HercuLeS [33] is the only high-level synthesis environment that allows seamless interfacing of user tools to the source, NAC, Graphviz CDFG and HDL (VHDL) levels. This even allows for competitor tools to utilize HercuLeS as a means for pre-optimizing source code or post-optimizing the generated HDL to improve their results.

As an example, consider the segment: `int divby60(int x){return x/60;}` for performing an integer/truncating division by 60. Typical HLS tools have the following approaches in dealing with this segment:

```
int divby60(int x) {
  int q, M=-2004318071, c;
  long long int t, u, v;
  t = (long long int)M * (long long int)x;
  q = t >> 32;
  q = q + x;
  q = q >> 5;
  c = x >> 31;
  q = q + c;
  return (q);
}
```

Figure 2 C code for truncating division by 60, optimized by `kdiv`.

Table 1 Using HercuLeS' optimized constant division in Vivado HLS.

Metric	Vivado HLS (unopt.)	Vivado HLS (opt.)	% improvement
LUTs	147	78	46.9
FFs	123	81	34.1
DSP blocks	4	4	0.0
SRL shifters	9	0	100.0
Propagation time (ns)	3.169	2.927	7.6
Number of cycles	8	8	0.0

- perform the division in floating-point arithmetic which is extremely costly
- use a variable divider which again is an overkill, both area- and timing-wise (32-33 cycles are required for a typical radix-2 divider). This is what LegUp does in addition to generating non-portable, vendor-specific code (using Altera-specific memory components)
- use a suboptimal algorithm to perform the constant division (Vivado HLS)
- are unable to process the request.

On the contrary, HercuLeS allows the interfacing of `kdiv`[a], an optimizer for constant division generating either C or NAC code. This way, an optimized circuit using only integer arithmetic with a double word-size multiplication is used [38]. The optimized C code is shown in Fig. 2.

Timing (minimum propagation delay) and area metrics have been obtained using the 2013.2 Vivado Design Suite for the XC7K325T-FFG900 device (-2 speed grade). As can be seen in Table 1, Vivado HLS underperforms if using its own algorithms, while benefits significantly from using the `kdiv`-based optimizer of HercuLeS.

In this paper we argue that transformational frameworks can be used at multiple levels of the HLS process. Further, we investigate syntactical extensions for developing self-contained optimizations that can be plugged-in to any hardware compiler that exposes a textual IR. These grammatical overrides are applied to the source or IR grammar and impose source/IR transformation rules [43]. Their suitability is manifested by four distinct optimization cases: a) source-level loop optimizations, b) late if-conversion [45], c) single constant multiplication [36, 37] and d) single constant division by providing a strongly-typed variant of [38].

[a] `http://sourceforge.net/projects/kdiv/`

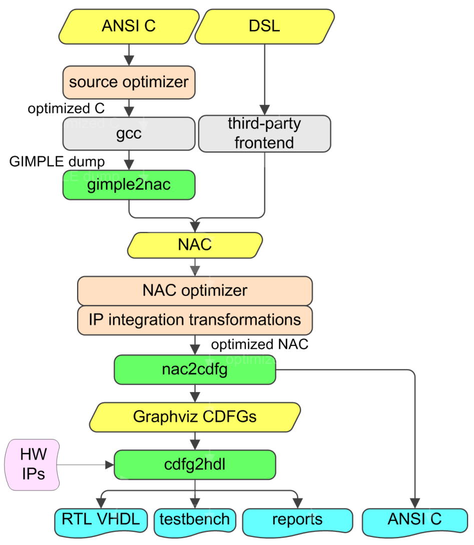


Figure 3 The HercuLeS flow.

4 HercuLeS basics

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.

4.1 Overview

The basic steps in the HercuLeS flow are shown in Fig. 3. 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 should directly supply a NAC translation unit (TU) [31] or use an owned frontend.

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].

Table 2 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

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 global or local static single assignment (SSA) form construction are supported [26, 31]. Data 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.

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 and Vivado Design Suite [23] using automatically generated scripts.

4.2 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 2.

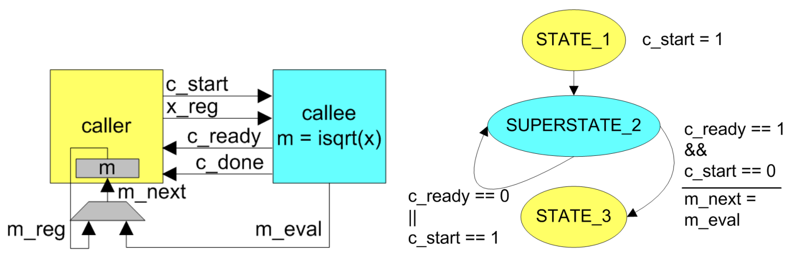
The FSMDs use $n + 2$ states, where n is the number of required computational states. The two overhead states represent CDFG source/sink nodes. One possible optimization is to merge the sink state with its immediate predecessors.

4.3 Hierarchical FSMDs

A two-state protocol describes proper communication between caller and callee FSMDs. The first state prepares the communication, while the second is an “evaluation” superstate where the entire computation applied by the callee FSMD is effectively hidden.

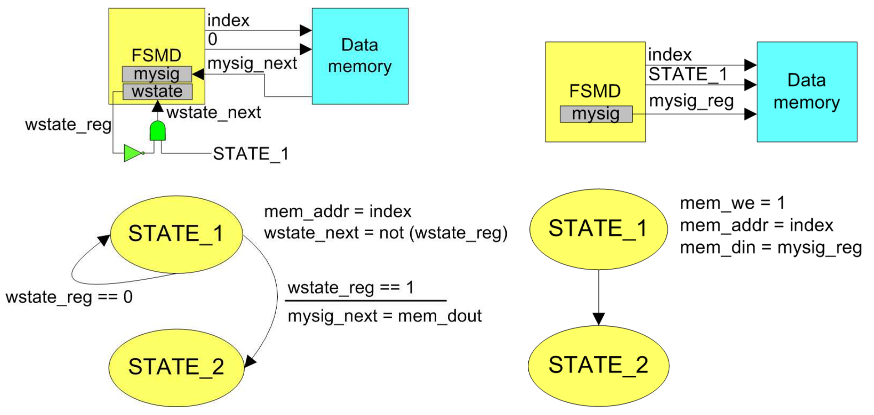
The caller FSMD performs computations where new values are assigned to `⋆_next` signals and registered values are read from `⋆_reg` signals. To avoid the problem of multiple signal drivers, callee procedure instances produce `⋆_eval` data outputs that can then be connected to register inputs by hardwiring to the `⋆_next` signal. Fig. 4 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



(a) Interface. (b) State transitions.

Figure 4 Caller-callee communication.



(a) `load` mechanism. (b) `store` mechanism.

Figure 5 Communication with on-chip memories.

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`.

4.4 *Communication with embedded memories*

In NAC, the `load` and `store` primitives are used for describing read and write memory access. We will assume a RAM model with write enable, and separate data input (`din`) and output (`dout`) sharing a common address port (`rwaddr`). To control access to such block, a set of four non-trivial signals is needed: a write enable signal (`mem_we`), and the corresponding signals for addressing, data input and output.

Fig. 5 depicts the implementation of memory access operations. Synchronous `load` requires the introduction of a `waitstate` register. This register assists in devising a dual-cycle sub-state for performing the load. During the first cycle of `STATE_1` the memory block is addressed. In the second cycle, the requested data are made available through `mem_dout` and are assigned to register `mysig`. This data can be read from `mysig_reg` during `STATE_2`. `store` raises `mem_we` in a given single-cycle state so that data are stored in memory and made available in the subsequent state/machine cycle.

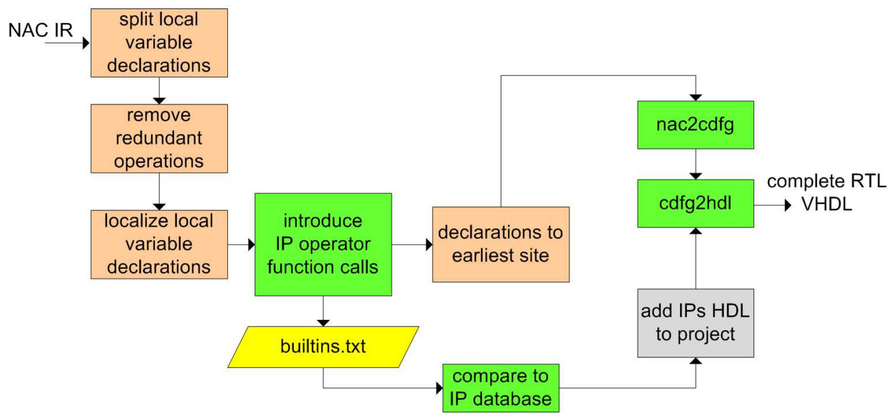

NAC IR
split local variable declarations
remove redundant operations
localize local variable declarations
introduce IP operator function calls
builtins.txt
declarations to earliest site
compare to IP database
nac2cdfg
cdfg2hdl
complete RTL VHDL
add IPs HDL to project

Figure 6 Automatic IP integration in HercuLeS.

4.5 IP integration

HercuLeS allows for automatic IP integration given that the user supplies built-in 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 [34] 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. 6 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 built-in functions. If the corresponding built-ins 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, while both pipelined and multi-cycle third-party components are supported.

5 Source level transformations

A crucial characteristic of HercuLeS against rival tools is the capability of adding external modules operating at the C source, NAC IR, and Graphviz CDFG levels. Key to this scheme is the exposure of self-contained textual representations at the appropriate levels of abstraction. This is advantageous since it allows to maintain a stable kernel while experimenting with transformations for code restructuring, optimization, and automatic IP integration.

Since NAC is a rather low-level representation for applying loop-oriented or other high-level transformations, a collection of autonomous C-to-C code transformation passes has

Table 3 Supported loop transformations.

Transformation	Description	Params
bump	Alter loop boundaries by an offset	offset, step
extension	Extend loop boundaries	lo, hi
reduction	Reverse the effect of extension	–
reversal	Reverse iteration direction	–
normalization	Convert arbitrary to well-behaved loops	–
fusion	Merges bodies of successive loops	–
coalescing	Nested loops into single loop	–
unswitching	Move invariant control code	–
strip mining	Single loop tiling	tilesize
partial unrolling	Partially unroll a loop by a factor	uf
full unrolling	Fully unroll a loop	–

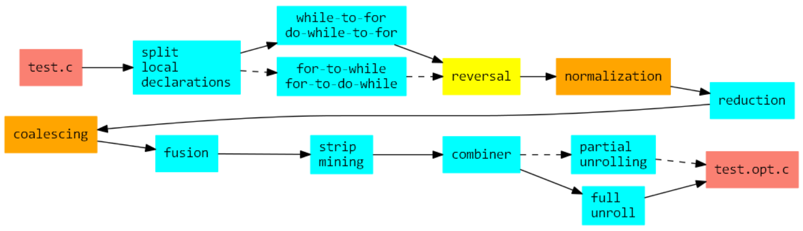


Figure 7 Flow of TXL transformations for C code optimization.

been developed in TXL [34]. TXL provides the means for developing syntactical extensions and rule-based transformations without the need to expose internally built, matched and substituted ASTs to the user. Further, TXL allows for agile parsing, meaning that internal variants of a grammar can be exploited to simplify analyses and optimizations.

This high-level optimizer supports generic restructuring transformations (GRTs) and loop-specific optimizations (LSOs). GRTs include code canonicalization for removing programming idioms, arithmetic optimizations, syntactic conversions among iteration schemes, and statement local vectorization. Loop-specific optimizations [35] are summarized in Table 3.

Fig. 7 illustrates a possible optimization flow using TXL passes. Certain decisions in this flow regard the ordering of transformations, since e.g. loop coalescing prerequisites loop normalization. It should be noted that strip mining eliminates chances for loop unrolling, and should not be applied unconditionally to static loops. Further, it is not meaningful to use both partial and full loop unrolling. User decisions also involve the proper selection of tile size, vector size and unroll factor values.

Fig. 8 illustrates the TXL transformation for strip mining [41]. This rule is provably safe and can be applied in all occasions. Strip mining is applicable when the length of vector-like processing (VL: Vector Length) is unknown at compile time, and is possibly larger than the hardware parallelism expressed as TS (Tile Size). TS is usually defined as the maximum vector length (MVL) of the underlying microarchitecture. It allows generating code such that each vector operation is performed for a size that is less than or equal to MVL. The TXL transformation creates a single strip-mined loop that is parameterized to handle first a portion smaller than TS and for all remaining iterations, portions of the loop equal to TS, until VL is consumed.

```
rule replaceStripMiningLT TS [number]
  replace $ [repeat decl_or_stmt]
    'for (Expr1 [expr]; Expr2 [expr]; Expr3 [expr])
      { Stmts [decl_or_stmt*] }
    MoreStmts [decl_or_stmt]
    deconstruct * [expr] Expr1
        Ix [id] = Expr4 [primary]
    deconstruct * [expr] Expr2
        Ix2 [id] < Expr5 [primary]
    deconstruct * [expr] Expr3
        Ix3 [id] = Ix4 [id] + Expr6 [primary]
    construct NewIx [id] Ix [_ 'sm] [!]
    construct VL [id] Ix [_ 'vl] [!]
    construct Low [id] Ix [_ 'low] [!]
    % Allow only for the innermost loop to be converted
    deconstruct not Stmts _ [for_stmt]
    where Ix [= Ix2] where Ix [= Ix3] where Ix3 [= Ix4]
    by
      'int Low '; Low = Expr4; 'int VL;
      % Find the odd-sized piece.
      VL = (Expr5-Expr4) % TS;
      'int NewIx;
      % Outer loop.
      'for (NewIx=-1; NewIx<(Expr5-Expr4)/TS; NewIx=NewIx+1)
        % Runs for length VL.
        { 'for (Ix=Low; Ix<Low+VL; Ix=Ix+Expr6)
          % Main operation(s).
          { Stmts }
        % Start of next vector.
        Low = Low + VL;
        % Reset the length to max/tilesize.
        VL = TS; }
      MoreStmts
end rule
```

Figure 8 TXL transformation rule for strip mining.

5.1 Example

As a running example to applying syntactical loop transformations according to the proposed flow in Fig. 7, the general matrix multiplication kernel is used (`gemm.c`), which is part of PolyBench [42]. The effect of alternative configurations regarding the sequence and selection of loop transformations is shown in Fig. 9.

6 NAC level transformations

In this section, we present self-contained optimizations for the NAC grammar. The TXL grammar for NAC uses different tokens for target (`id`) and source operands (`ident`). Further, TXL assists in developing syntactic extensions of NAC. The corresponding NAC dialects are then used in intermediate stages of these transformations since TXL rules that convert patterns to replacements starting from the original pattern are unfeasible. TXL coding enforces certain transformation paradigms such as declaration localization, globalization and statement vectorization [44] and scalarization as stepping stones for developing custom transformations.

```
for (i = 0; i < NI; i++) {
  for (j = 0; j < NJ; j++) {
    C[i][j] *= beta;
    for (k = 0; k < NK; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
    }}}
```

(a) Initial code.

```
#define MIN(x, y) ((x) <  (y) ? (x) : (y))
int i_coal4sm1;
for (i_coal4sm1=0; i_coal4sm1<4096; i_coal4sm1=i_coal4sm1+8) {
  for (i_coal4=i_coal4sm1; i_coal4<MIN(4096,i_coal4sm1+8); i_coal4=i_coal4+1) {
    i = i_coal4 / 64, j = i_coal4 % 64;
    C[i][j] *= beta;
#define MIN(x, y) ((x) <  (y) ? (x) : (y))
    int k_sm1;
    for (k_sm1=0; k_sm1<64; k_sm1=k_sm1+8) {
      for (k=k_sm1; k<MIN(64,k_sm1+8); k = k + 1) {
        C[i][j] += alpha * A[i][k] * B[k][j];
      }}}}
```

(b) Following loop normalization, coalescing and strip mining.

```
int i_coal4;
for (i_coal4=0; i_coal4<4096; i_coal4=i_coal4 + 1) {
  i = i_coal4 / 64, j = i_coal4 % 64;
  C[i][j] *= beta;
  {
    C[i][j] += alpha * A[i][0] * B[0][j];
    C[i][j] += alpha * A[i][1] * B[1][j];
    ...
    C[i][j] += alpha * A[i][63] * B[63][j];
  }}
```

(c) Following loop coalescing and full unrolling.

```
int i_coal4;
for (i_coal4=0; i_coal4<4096; i_coal4=i_coal4+1) {
  i = i_coal4 / 64, j = i_coal4 % 64;
  C[i][j] *= beta;
  for (k = 0; k < 63;) {
    C[i][j] += alpha * A[i][k] * B[k][j];
    k = k + 1;
    C[i][j] += alpha * A[i][k] * B[k][j];
    k = k + 1;}
  for (; k < 64; k = k + 1) {
    C[i][j] += alpha * A[i][k] * B[k][j];}}
```

(d) Following loop coalescing and partial unrolling.

Figure 9 `gemm.c` exposed to different configurations of the loop transformation flow.

6.1 If conversion

If-conversion reduces control- to data-dependence [45]. NAC provides the `muxzz` quaternary multiplexing operation that implements a conditional move resolving both conditions of the predicate.

Fig. 10 illustrates the flow of the corresponding TXL transformation applied to SSA NAC. First, control-flow regions amenable to dependence conversion are identified. Two new variable declarations are needed for keeping the transfer operands of the new `muxzz` operation. At this point, these declarations use dummy data types. Then, intermediate variable declarations are moved to their declaration sites. Since SSA form is used, these declarations always immediately precede their definition and can be easily matched. The

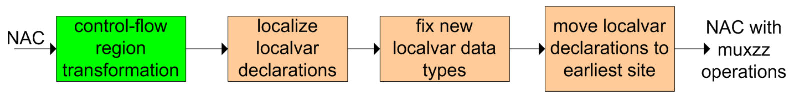

NAC
control-flow region transformation
localize localvar declarations
fix new localvar data types
move localvar declarations to earliest site
NAC with muxzz operations

Figure 10 Flow of TXL transformations for if-conversion.

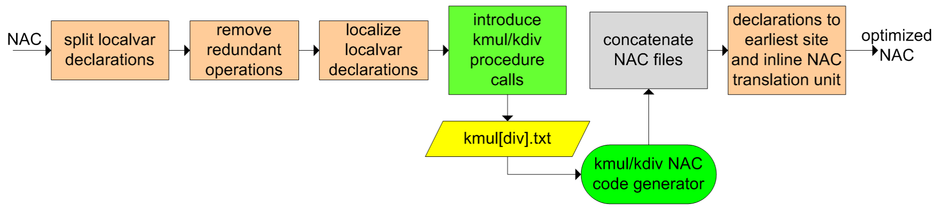

NAC
split localvar declarations
remove redundant operations
localize localvar declarations
introduce kmul/kdiv procedure calls
concatenate NAC files
declarations to earliest site and inline NAC translation unit
optimized NAC
kmul[div].txt
kmul/kdiv NAC code generator

Figure 11 Flow of TXL transformations for constant multiplication/division.

following step fixes the transfer operand declarations to the type of their target operand. Then, local variable declarations are finally moved to the earliest site in the current scope in order to be compatible with the NAC grammar.

6.2 *Constant multiplication*

Multiplication by an integer constant (*kmul*) allows for significant speed improvement and area reduction since a variable multiplier needs not be used. Contemporary FPGAs offer high-speed embedded multipliers, however it is always relevant to microprocessors without full-range hardware multipliers.

TXL is unsuitable for extensive numerical computation, thus the actual NAC code generator for the constant multiplication is written in C. Fig. 11 illustrates the combined TXL and ANSI 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 specialized constant multiplication routines are introduced. These routines are then generated by the C tool and concatenated to the NAC IR of the application. The optimized routines can be inlined at their call site to eliminate argument passing overheads. Constant division optimization is applied by using the exact same process.

6.3 *Constant division*

A workaround for constant division (*kdiv*) in current processors uses a multiplication with the multiplicative inverse (or ‘magic’ number) of the constant followed by a number of adjustment steps [38, 39].

Integer division instructions are either omitted from modern microarchitectures or tend to be a speed limiting factor. While the Intel E6xx series employs a radix-16 divider, there is no easy way to incorporate a high-performance divider in FPGA implementations of soft-core processors. Nios-II and Microblaze incorporate multi-cycle radix-2 dividers; LEON3 [46] uses a division step primitive to achieve similar performance. For a generic division routine, typically $c_l N + offset$ cycles are spend for the calculation, with c_l being the cycle count for the inner loop, N the number of calculation stages $(W/log_2(k)$ for k-

```
CONST-DIV-SIGNED
begin
  declare localvar s$W$ q, M, c;
  declare localvar s2$W$ t, u, v;
  LAB_1:
  // S1. Load magic constant.
  M <= ldc $m$;
  // S2. Perform signed high multiplication.
  t <= mul M, $n$;
  u <= shr t, $W$;
  q <= trunc u;
  // S3. Add or subtract n from q and store in q.
  if $d > 0$ and $m < 0$ then
    q <= add q, n;
  else if $d < 0$ and $m > 0$ then
    q <= sub q, n;
  endif
  // S4. Quotient correction steps using dividend and divisor sign.
  if $s > 0$ then
    q <= shr q, $s$;
  endif
  c <= shr n, $W - 1$;
  q <= add q, c;
  if $d < 0$ then
    c <= setne n, 0;
    q <= add q, c;
  endif
  y <= mov q;
end
```

Figure 12 Pseudocode for the signed constant division generator.

radix division and W-bit word-lengths) and $offset$ denotes the sum of pre- and post-loop overhead cycles.

We have modified the aforementioned *kdiv* algorithm for use with strongly-typed low-level languages as shown in Fig. 12. Past implementations gratuitously allow for language- and machine-specific behavior when interpreting certain cases for type casting and promotion. In our case, specialized operations such as the `shrxi` extended shift immediate operation in [38] are not needed. Our algorithm produces a quotient for the equivalent of a high multiplication (`umulhi` in [39]) implemented by a 64-bit `mul` followed by `shr`. Alternatively, NAC can be extended with a dedicated high multiplication.

The code generator accounts for the generic case of a constant divisor d. The magic number is denoted by m, s is a correctional shift amount and W is the bitwidth of a machine word. Input dividend n and quotient y both have a width of W. Emitted NAC code is shown in monospace font. First, the algorithm generates declarations for `localvar` variables. Following this, it generates the `ldc` and `mul` operations, the `add` if $d > 0$ and $m < 0$, or the `sub` if $d < 0$ and $m > 0$. Then, the extended immediate shift concatenating the carry/borrow bit by the previous operation is generated if $s > 0$. Additional quotient correctional steps are generated by using a regular shift immediate and an increment.

7 Performance evaluation of the transformational schemes

To assess the performance of the proposed transformational schemes, we demonstrate the following:

1. Effect of loop-oriented transformations.
2. Effect of if-conversion on benchmark cycles.
3. Machine cycles for optimized *kmul/kdiv*.

Table 4 PolyBench applications with absolute cycle counts on VEX.

Benchmark	Description	Reference	Full loop unrolling
2mm	two matrix-matrix multiplications	329566	65635
3mm	three matrix-matrix multiplications	491558	92991
atax	matrix transpose and vector multiplication	72729	10519
bicg	subkernel of BiCGStab linear solver	51802	40122
doitgen	reduction sum of a 3D$\times$2D matrix	373942	158586
dynprog	dynamic programming problem solver	65635	230644
gemm	general matrix-matrix multiplication	1478041	155712
gesummv	scalar, vector and matrix multiplication	43669	32477
mvt	matrix-vector product and transpose	57048	97042
symm	symmetric matrix-matrix multiplication	186558	209593
syr2k	symmetric rank-2k update	1735512	4008928
syrk	symmetric rank-k update	1452630	3992287
trmm	triangular matrix-matrix multiplication	1368907	1392347

4. Estimated timing and area demands for hardware implementations of the latter.

For evaluating scenario 1, a cycle-accurate simulator for an experimental VLIW microarchitecture was used. For scenarios 2–3, *gimple2nac* was used and machine cycles are measured on a cycle-accurate compiled C model of the NAC virtual machine.

The timing (minimum propagation delay) and area requirements are estimated for the 40nm Virtex-6 6-input LUT FPGA process. The logic synthesis tool used is Xilinx Webpack ISE 12.3i. Throughout the evaluations, the XC6VLX75T device has been selected, ranking among the smallest devices in the respective family. It should be noted that *kdiv* hardware uses a 32-bit multiply producing a 64-bit result, which nominally requires 4 DSP48E datapath blocks (3 if further optimized).

7.1 *Effect of C-level transformations*

In the first set of experiments, the loop transformation engine is evaluated over a parameterized VLIW architecture named VEX [47, 48]. The VEX toolchain provides the means to target a wide class of embedded VLIW processors, by using a complete ANSI C compilation toolset and a cycle-accurate simulator. VEX was configured as a single-cluster VLIW machine featuring eight execution slots. The VEX scheduler attempts to schedule the maximum available number of independent operations in parallel, performing a number of loop-oriented optimizations.

Table 4 summarizes the PolyBench benchmarks [42] that have been used in this evaluation. It also provides the exact absolute cycles for VEX for the three distinct cases of using reference and then optimized code by full loop unrolling. In order to obtain these measurements, the same transformation script is applied to all applications.

Many applications have significant gains, even by following the strategy of unconditionally applying the loop transformation sequence. A closer look would unveil that some applications are hampered by the generic form of strip mining, which increases the looping overhead. Overall, a 32% improvement on geometric means is achieved.

7.2 *Effect of if-conversion*

To analyze the effect of the if-conversion transformation, a set of small integer/fixed-point kernels has been selected: *bitunzip/bitzip* (data interleaving/(de)compression algorithm [49]), *cordic* (multi-function fixed-point CORDIC), *divider* (sequential division algo-

Table 5 Summary of NAC abstract machine cycles for the given benchmarks.

Bench.	Cycles for each case					
	Seq	Seq+ ifconv	%diff	ASAP	ASAP+ ifconv	%diff
bitunzip	193536	191488	1.1	80896	79872	1.3
bitzip	197632	197632	0.0	96256	96256	0.0
cordic	667526	599892	10.1	421161	291561	30.8
divider	1375402	1375402	0.0	970651	970651	0.0
easter	4318	4200	2.7	4018	3600	10.4
eda	768	736	4.2	512	480	6.3
float2half	30208	28339	6.2	21208	19339	8.8
half2float	983042	917506	6.7	655362	589826	10.0
perfect	528268	527248	0.2	527758	526228	0.3
sieve	463058	463058	0.0	463047	463047	0.0

rithm), *easter* (Easter date calculations), *eda* (Euclidean distance approximation), *float2half* and *half2float* (conversion to/from 16-bit floating-point format), *perfect* (perfect number detection) and *sieve* (prime sieve of Eratosthenes).

Reported machine cycles have been collected in Table 5 assuming either a sequential or a vectorized abstract machine model based on NAC. The latter case is equivalent to ASAP scheduling of scalar operations. For each benchmark, measured cycles with/without the use of if-conversion are given in columns 2-3 and 5-6, respectively for the two models. Columns 4 and 7 report the percentage difference between the values in the two preceding columns. From these results, it can be deduced that if-conversion moderately improves cycle performance: by 3.1% for sequential and 6.8% for ASAP scheduling. Some benchmarks do not benefit at all since they do not incorporate a pattern that can be matched by the respective TXL rules. *cordic* is the application with the most profound impact (10.1% and 30.8% improvement, respectively).

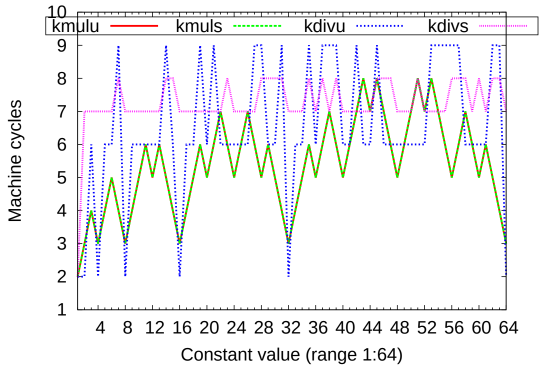
7.3 Timing measurements for implementing kmul/kdiv

Fig. 13(a) illustrates the number of machine cycles for signed and unsigned *kmul/kdiv* among 32-bit quantities for all divisors up to 64. It can be seen that multiplication (`kmulu, kmuls`) requires 2–8 cycles to complete, while division takes 2–9 cycles. It is assumed that the multiplier and divider are separable IPs using registered outputs. In average, `kmul` operators require 16.8% and 25.4% less cycles than the respective `kdiv` ones. Still, the benefit from avoiding division is significant, depending on the specific profile of a given application.

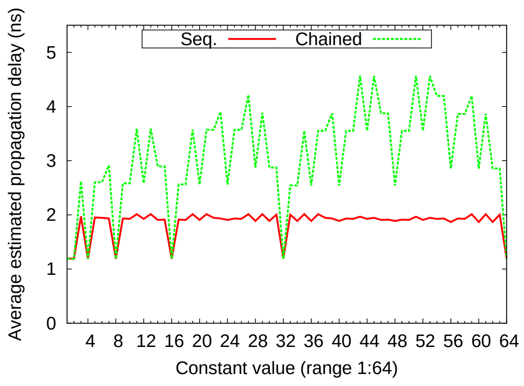
It should be noted that the *kdiv* cost is amortized to the maximum of 9 cycles irrespective to the specific constant. It compares favorably to the 32-33 cycles usually needed for wordwise integer division. On the other hand, *kmul* requires an increasing number of operations for certain larger constants.

The discussed *kmul/kdiv* operators have been implemented in RTL VHDL using HercuLeS HLS. For spacing reasons, detailed results are only shown for signed arithmetic. Fig. 13(b) and Fig. 13(c) depict the estimated propagation delay for different *kmul/kdiv* operators for the same set of factors: {1:64}, respectively. We have investigated two possible intra-unit schedules: a sequential with one NAC operation occurring per machine cycle and chained ASAP schedule where the entire computation is collapsed into a single machine cycle.

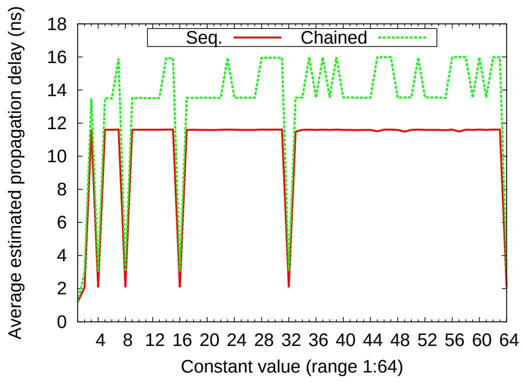
As expected, the estimated propagation delay is higher for the case of the chained



(a) Machine cycles for constant mul/div.



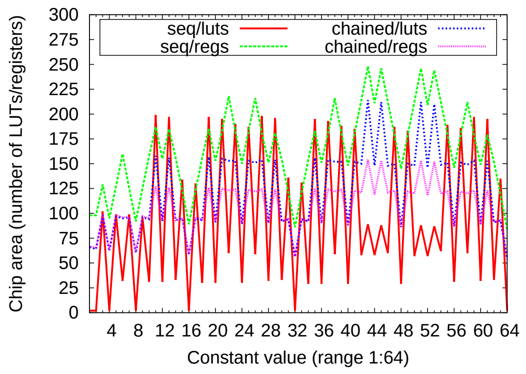
(b) Estimated propagation delay (ns) for the *kmul* units.



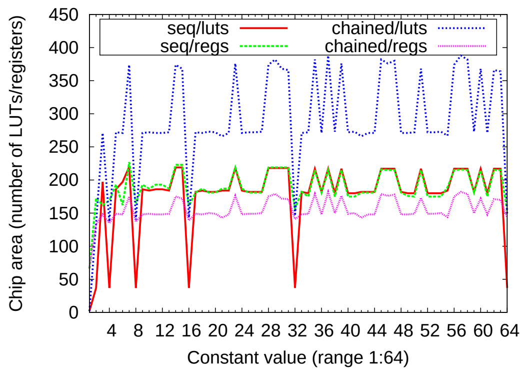
(c) Estimated propagation delay (ns) for the *kdiv* units.

Figure 13 Timing measurements for *kmul/kdiv* units on the XC6VLX75T Virtex-6 FPGA.

schedule by 40.1% and 19.9%, respectively for multiplication and division. Due to the use of a combinational wide multiplier using 4 DSP48E blocks, division is slower by 5.7× and 4.2× compared to multiplication. In the context of a custom architecture and not a microprocessor, the chained version of the units may be preferred over the sequential one, since it reduces the total computation time by a factor of 3.3× and 5.9×, respectively for



(a) Area for `kmuls`.



(b) Area for `kdivs`.

Figure 14 Chip area in number of LUTs and registers for the constant multiplication/division units on the XC6VLX75T Virtex-6 FPGA.

kmul/kdiv.

7.4 Area measurements for implementing kmul/kdiv

The chip area requirements for signed operators are shown in Fig. 14. For the case of `kmuls` there appears a trade-off between the number of LUT and register resources since the combinational version requires a 28.1% increase in LUTs and a 33.2% decrease in registers (average). The corresponding trade-off for `kdivs` is 41.2% and 16.6%, respectively. In general, the extreme cases of resource consumption appear for constants 57 (*kdiv*) and 43 (*kmul*).

8 Performance evaluation against Vivado HLS

Table 6 provides information on the performance of HercuLeS against Vivado HLS 2013.2 on Virtex-6 and Kintex-7 (XC7K70TFBG676-2 FPGA device). Better results (lower execution time; smaller area) have been typeset in bold. HercuLeS outperforms Vivado HLS in key benchmarks such as filtering and numerical processing. As expected

Table 6 Out-of-the-box comparison of HercuLeS against Vivado HLS 2013.2.

Benchmark	Vivado HLS			HercuLeS			Family
	LUTs	Regs	Time (ns)	LUTs	Regs	Time (ns)	
Bit reversal	67	**39**	72.0	**42**	40	**11.6**	Virtex-6
Radix-2 division	**218**	**226**	63.6	318	332	**30.6**	Kintex-7
Edge detection	**246**	**130**	1636.3	680	361	**1606.4**	Virtex-6
Fibonacci series	138	**131**	**60.2**	**137**	197	102.7	Virtex-6
FIR filter	**89**	**114**	1027.1	606	540	**393.8**	Kintex-7
Greatest common divisor	210	98	**35.2**	**128**	**93**	75.9	Virtex-6
Cubic root approx.	**239**	207	**260.6**	365	**201**	400.5	Virtex-6
Prime sieve	525	595	6108.4	**565**	**523**	**3869.5**	Virtex-6

in many occasions, better speed/performance can be traded-off for lower area.

9 Conclusion

HercuLeS delivers a contemporary HLS environment that can be comfortably used for algorithm acceleration by software-oriented engineers. In this manuscript, automatic IP integration and the transformational framework used in HercuLeS have been presented in detail. Transformations are envisioned as standalone passes that do not affect the inner workings of other aspects of the compiler. Both control- and data-oriented optimizations were investigated focusing on loop optimization, if-conversion and arithmetic transformations. Loop optimizations provide 71% improvement in cycle performance, while in another scenario, if-conversion offers an average improvement of 6.5%. Further, operations-by-constant can be implemented with an amortized cost of a few cycles.

References and Notes

1 C-to-Silicon. http://www.cadence.com/products/sd/silicon_compiler/. Last accessed: Sep. 24, 2014.
2 C-to-Verilog. http://www.c-to-verilog.com. Last accessed: Sep. 24, 2014.
3 CatapultC. http://calypto.com/en/products/catapult/overview/. Last accessed: Sep. 24, 2014.
4 R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 619–622, Oslo, Norway, Aug. 2012.
5 CoSy. http://www.ace.nl/compiler/cosy.html. Last accessed: Sep. 29, 2014.
6 GAUT. http://www-labsticc.univ-ubs.fr/www-gaut/. Last accessed: Sep. 24, 2014.
7 GHDL. http://ghdl.free.fr. Last accessed: Sep. 24, 2014.
8 GIMPLE. http://gcc.gnu.org/wiki/GIMPLE. Last accessed: Sep. 24, 2014.
9 Graphviz. http://www.graphviz.org. Last accessed: Sep. 29, 2014.
10 ImpulseC. http://www.impulseaccelerated.com. Last accessed: Sep. 24, 2014.
11 ITRS. http://www.itrs.net/reports.html. Last accessed: Sep. 24, 2014.
12 LegUp. http://legup.eecg.utoronto.ca/. Last accessed: Sep. 24, 2014.
13 A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, J. H. Anderson. LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. on Embedded Computing Systems*, 13(2):1–27, Sep. 2013.
14 LLVM. http://llvm.org. Last accessed: Sep. 24, 2014.
15 Modelsim. http://www.mentor.com/products/fpga/model/. Last accessed: Sep. 24, 2014.
16 ROCCC. http://www.jacquardcomputing.com/roccc/. Last accessed: Sep. 24, 2014.
17 SPARK. http://mesl.ucsd.edu/spark/. Last accessed: Sep. 24, 2014.
18 S. Gupta, R. Gupta, N. D. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2004.

19 Synphony HLS. http://www.synopsys.com/Systems/BlockDesign/HLS/. Last accessed: Sep. 24, 2014.
20 The GNU Compiler Collection homepage. http://gcc.gnu.org. Last accessed: Sep. 24, 2014.
21 TransC. http://cgi.tu-harburg.de/˜ti6hm/. Last accessed: Sep. 24, 2014.
22 H. Manteuffel, C. S. Başsoy, and F. Mayer-Lindenberg. The TransC process model and inter-process communication. In *Proc. Int. Conf. on Field-Programmable Technology (FPT 2010)*, pp. 87–93, Tsinghua University, Beijing, China, USA, Dec. 2010.
23 Xilinx. http://www.xilinx.com. Last accessed: Sep. 24, 2014.
24 IEEE. *1364-2005 Standard for Verilog Hardware Description Language*, Apr. 2006.
25 IEEE. *1076-2008 Standard VHDL Language Reference Manual*, Jan. 2009.
26 A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr. 1998.
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 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.
31 N. Kavvadias and K. Masselos. NAC: A lightweight intermediate representation for ASIP compilers. In *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*, pp. 351–354, Las Vegas, Nevada, USA, Jul. 2011.
32 Xilinx. Vivado ESL Design. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm. Last accessed: Sep. 24, 2014.
33 Ajax Compilers. HercuLeS HLS. http://www.ajaxcompilers.com/technology/hercules-high-level-synthesis. Last accessed: Sep. 24, 2014.
34 J. R. Cordy. TXL programming language. http://www.txl.ca. Last accessed: Sep. 24, 2014.
35 D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.
36 R. Bernstein. Multiplication by integer constants. *Software: Practice and Experience*, vol. 16, no. 7, pp. 641–652, Jul. 1986.
37 P. Briggs and T. Harvey. Multiplication by integer constants. Rice Univ., Tech. report, Jul. 1994.
38 H. S. Warren. *Hacker's Delight*, 2nd edition. Addison-Wesley, Oct. 2012.
39 N. Möller and T. Granlund. Improved division by invariant integers. *IEEE Trans. on Computers*, 60(2):165–175, Feb. 2011.
40 N. Kavvadias and K. Masselos Hardware design space exploration using HercuLeS HLS. In *17th Panhellenic Conference on Informatics with international participation*, pp. 195–202, Thessaloniki, Greece, Sep. 2013.
41 J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.
42 L.-N. Pouchet. PolyBench. http://sourceforge.net/projects/polybench/.
43 J. R. Cordy. The TXL Source Transformation Language. In *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, Aug. 2006.
44 J. R. Cordy. Excerpts from the TXL cookbook. In *Generative and Transformational Techniques in Software Engineering III*, pp. 27–91, Braga, Portugal, Jul. 2009.
45 J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th ACM SIGPLAN Symp. on Principles of programming languages*, 1983, pp. 177–189.
46 Aeroflex Gaisler. http://www.gaisler.com.
47 VEX homepage. http://www.hpl.hp.com/downloads/vex/.
48 J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, Dec. 2004. http://www.vliw.org/book/.
49 J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Online version: http://www.jjj.de/fxt/fxtbook.pdf, Aug. 2011.