Domain-specific Optimisation for the High-level Synthesis of CellML-based Simulation Accelerators

Julian Oppermann and Andreas Koch
Embedded Systems and Applications Group
Technische Universität Darmstadt, Germany
Email: {oppermann, koch}@esa.tu-darmstadt.de

Ting Yu and Oliver Sinnen
Parallel and Reconfigurable Computing Group
University of Auckland, New Zealand
Email: {ting.yu, o.sinnen}@auckland.ac.nz

***Abstract*—The simulation of biomedical models often requires the numerical integration of ordinary differential equation systems, a computationally intensive task that can be accelerated well by deeply-pipelined FPGA-based accelerators. Since the main design target is throughput, larger FPGA devices can easily be exploited by scaling-up the number of parallel datapath instances on a chip. To this end, reducing the area of each datapath becomes a key optimisation. High-level synthesis can be employed to generate custom simulation accelerators from standardised cell descriptions in CellML. In this work, we improve this process by inserting LLVM into the flow to pre-optimise the simulation models generated from CellML for hardware synthesis. This is achieved not only by the selective application of general-purpose optimisation passes, but also by adding new domain-specific optimisations, including unsafe floating-point transformations, to the optimisation flow. We investigate their effect on the quality-of-results and show that a novel strategy using our optimisations outperforms standard strategies, such as LLVM's -Oz (aggressive size reduction), when applied for hardware synthesis in 99 out of 146 example models. Our approach, which reduces area by up to 25%, leads to the smallest implementations for four models examined in detail, and allows a particularly complex cell model to fit on the target FPGA device for the first time.**

I. Introduction

Computer-based simulations of models expressed as ordinary differential equation systems help researchers understand the function of the human physiology and other biomedical problems. CellML [1] emerged as an open standard to formulate and exchange such models, and laid the foundation for an ecosystem of modeling and simulation tools. Simulations often use numerical integration to solve the underlying equation systems. This is a computationally intensive task, as it requires the evaluation of the equations at huge numbers of grid points and time steps to satisfy increasing demands for simulation accuracy.

Yu et al. [2] recently presented ODoST, a high-level synthesis tool to automatically generate FPGA-based hardware accelerators for rapid simulation of CellML models. As the problem at hand is highly data parallel, the accelerator pipelines can be assumed to be fully saturated for most of the execution time of the accelerator, making the latency negligible and elevating the throughput to the primary optimisation goal. The throughput, in turn, can be increased by mapping multiple parallel instances of the accelerator onto the target FPGA [3].

In this paper, we extend the original ODoST compilation flow by additional pre-optimisations realised in LLVM [4]. This pre-processing includes the application of general-purpose optimisations, including unsafe floating-point transformations, as well as of newly developed domain-specific passes. Using a custom recipe (pass sequence) of the existing and new optimisations, we outperform LLVM's aggressive size reduction preset recipe -Oz in up to 99 out of 146 example models, while maintaining user-selected accuracy bounds. Our extended flow delivers the smallest designs in the synthesis results for four sample accelerators generated by ODoST, and with the use of the domain-specific transformations, allows the most complex of these models to fit on the FPGA device for the first time.

II. Background

A CellML model is a *description* of a set of interconnected components. It is translated into an executable simulation model by a tool such as the CellML code generation service (CCGS) [5], [6]. The CCGS treats the mathematics embedded in the model description as an initial-value problem and determines a sequential evaluation order for the equation system, ultimately emited as C code into a function as shown in Figure 1. From a compiler's perspective, the inputs to the computation are the **`VOI`**, **`CONSTANTS`** and **`STATES`** arrays, and the **`RATES`** and **`ALGEBRAIC`** arrays are outputs. [6] explains the meaning of the different computation targets in a model simulation context. The arrays do not alias or overlap; their sizes are constant and externally known. Elements in the **`ALGEBRAIC`** array are always written before they are read in other equations.

In software, the computation is performed on double-precision floating-point numbers, using several mathematical library functions in addition to the standard C operators.

Despite the name, the elements from the **`CONSTANTS`** input are treated more like parameters than compile-time constants, as it is desirable to change the values without the need to resynthesise the hardware accelerator each time.

The CellML standard does not specify any accuracy bounds, and CCGS does not take precautions (e.g., such as the ones in [7], [8]) concerning the numerical stability of the formulas. This freedom allows the existing ODoST tools to exploit area-saving single-precision computation, and motivates our own work on exploiting unsafe floating-point transformations for further area reductions.

```
void computeRates(
double VOI, double* CONSTANTS, double* RATES, double* STATES, double* ALGEBRAIC
) {
ALGEBRAIC[1] = -0.1 * (STATES[0] + 50.0) / (exp(-(STATES[0] + 50.0) / 10.0) - 1.0);
ALGEBRAIC[5] = 4.0 * exp(- (STATES[0] + 75.0) / 18.0);
RATES[1]     = ALGEBRAIC[1] * (1.0 - STATES[1]) - ALGEBRAIC[5] * STATES[1];
ALGEBRAIC[2] = 0.07 * exp(- (STATES[0] + 75.0) / 20.0);
ALGEBRAIC[6] = 1.0 / (exp(- (STATES[0] + 45.0) / 10.0) + 1.0);
RATES[2]     = ALGEBRAIC[2] * (1.0 - STATES[2]) - ALGEBRAIC[6] * STATES[2];
ALGEBRAIC[3] = -0.01 * (STATES[0] + 65.0) / (exp(-(STATES[0] + 65.0) / 10.0) - 1.0);
ALGEBRAIC[7] = 0.125 * exp((STATES[0] + 75.0) / 80.0);
RATES[3]     = ALGEBRAIC[3] * (1.0 - STATES[3]) - ALGEBRAIC[7] * STATES[3];
ALGEBRAIC[4] = CONSTANTS[2] * pow(STATES[1], 3.0) * STATES[2]
               * (STATES[0] - CONSTANTS[5]);
ALGEBRAIC[8] = CONSTANTS[3] * pow(STATES[3], 4.0) * (STATES[0] - CONSTANTS[6]);
ALGEBRAIC[9] = CONSTANTS[4] * (STATES[0] - CONSTANTS[7]);
ALGEBRAIC[0] = (VOI >= 10.0 && VOI <= 10.5 ? 20.0 : 0.0);
RATES[0]     = - (- ALGEBRAIC[0] + ALGEBRAIC[4] + ALGEBRAIC[8] + ALGEBRAIC[9])
               / CONSTANTS[1];
}
```

Fig. 1. Equations for [9], as generated by CCGS, reformatted for readability.

ODoST [2] creates Hardware Accelerator Modules (HAM) for the numerical integration step during model simulation. It uses IEEE-754 single-precision floating-point [10] operators generated by FloPoCo [11], and currently supports the basic arithmetic operators as well as the power, exponential, natural logarithm, floor, absolute value and negation functions. Limited support exists for comparisons, logical operations and the conditional assignment operator.

The HAMs employ a fully spatial computation model, i.e., every operation in the equation system is implemented by a dedicated hardware module on the FPGA. A reduction of the *number* of operations thus directly leads to a reduction of hardware *area*. Such transformations fall into the class of architecture-independent compiler optimisations [12], [13], many of which attempt to reduce the number of operations in a given program, e.g., by the elimination of redundant computations, or replace expensive operations by cheaper ones (according to a problem-specific cost model).

In a survey on code size reduction techniques conducted by Beszédes et al. [14], most works propose compression schemes for the binary encoding of a compiled program. Only [15], [16] perform compiler transformations to extract reusable instruction sequences into procedures. The hardware equivalent of this concept would be the multiplexing of inputs for individual operators or parts of the datapath, which is infeasible in our setup, as we assume that all operators are in use for almost the entire runtime due to the pipelined architecture.

Among the high-level synthesis systems with a more general scope, Huang et al. [17] performed an extensive automated search in the space of LLVM optimisation sequences for LegUp to assess their effects on various metrics, including circuit area. However, their benchmarks did not contain floating-point operations, and they state that the different optimisation flows in the study had no significant effect on the number of required logic elements. Buyukkurt et al. [18] improved the resource requirements of ROCCC-generated accelerators by using common transformations, accompanied by a tree-height reduction and the replacement of multiplications and division by sequences of shifts and additions.

Kastner et al. [19] published a book on arithmetic optimisation for a later hardware realisation, with a focus on the efficient implementation of integer/fixed-point polynomials and linear systems. The book gives an introduction to the multiple constant multiplication problem, which is prevalent in digital finite impulse response (FIR) filter design: Given a value that has to be multiplied by a set of integer factors, one wants to eliminate the multiplications in favour of a minimal sequence of bit-shifts and additions. This concept is not directly applicable to floating-point multiplications; however, we introduce a transformation with a much simpler variant of this idea in our proposed optimisation strategy.

Trade-offs between hardware area and accuracy have also been examined in SOAP [20]. Our technique is simpler in that it applies a fixed sequence of transformations, while SOAP performs a more thorough iterative design-space exploration to determine Pareto-optimal solutions. On the other hand, SOAP supports just the four basic arithmetic operations and lacks the power and exponentiation operators crucial for the optimisation of CellML models.

Garny et al. [5] acknowledge the applicability of compiler optimisations to code generated form CellML models, and refer to the work of Cooper et al. [21] who performed partial evalation [22] on a model representation in a functional language, and converted the resulting program back to a CellML description. Additionally, during code generation, they identified a subexpression occuring frequently and with inputs known to be in a small interval and replaced the evaluation by a lookup table-backed approximation.

III. Optimising for lower resource requirements

In this work, we aim to optimise for lower FPGA resource requirements by elimination and simplification of operations in a CellML model's equation system, prior to performing the actual high-level synthesis.

To this end, we first need to define a cost-model to guide our optimisations, analyse the gains achievable by selective applications of existing optimisation passes, and then define domain-specific transformations.

A. Cost model

The HAMs constructed by ODoST currently target an Altera Stratix IV EP4SGX530 device. We define $u_{\text{ALM}}(op)$ as the demand of adaptive logic modules (ALM) and $u_{\text{DSP}}(op)$ as the demand of 18-bit DSP blocks (DSP), of individually synthesised operators relative to the target device's total resources (values are shown here in %). Table I shows the actual quantities from [2]. The usage of these resource types is orthogonal in a first approximation, leading to the notion of the combined usage as a point in a two-dimensional space. We define the cost of each operator in the equation system as the Euclidean norm, e.g. the length, of the point's position vector:

$$c(op) = \left\| \begin{pmatrix} u_{\text{ALM}}(op) \\ u_{\text{DSP}}(op) \end{pmatrix} \right\| \tag{1}$$

The cost of the whole equation system is defined as the sum of the cost of each operator as defined in Eq. (1) and serves

TABLE I
RESOURCE USAGE FOR THE USED FLOPOCO-GENERATED [11], SINGLE-PRECISION OPERATORS ON THE STRATIX IV DEVICE.

Operator	ALM	u_{ALM} [%]	DSP	u_{DSP} [%]
FPAdd	395	0.19	-	-
FPMult	132	0.06	4	0.39
FPDiv	1116	0.52	-	-
FPExp	507	0.23	2	0.20
FPLog	808	0.38	18	1.79
FPPow	2058	0.97	31	3.03
Device	212480	100	1024	100

ALM = Adaptive Logic Module · DSP = 18-bit DSP block

as an estimate of the resource requirements for the resulting HAM.

The presented cost model is tailored to the characteristics of the particular Altera FPGA used in this work, but could be easily adapted to other FPGA architectures. For ASIC designs, an estimate of the silicon area could be employed as an even simpler cost function.

The optimisations utilised in this work either eliminate an operation altogether, which is clearly always beneficial in the cost model, or replace an operation with another kind. However, no ordering relation of the different operator types exists that holds for both the ALM and DSP usage, so there are transformations of the latter kind that lower the relative ALM usage, but require a larger share of the DSP blocks, and vice versa. Defining the cost based on the vector norm allows transformations where the improvement in one dimension dominates the degradation in the other dimension. For example, replacing a multiplication having cost $c(*) = \left\|\binom{0.06}{0.39}\right\| \approx 0.39$ by an addition with $c(+) = \left\|\binom{0.19}{0}\right\| \approx 0.19$ results in a cost reduction, therefore making this a viable optimisation. Analogously, replacing a division with $c(/) \approx 0.52$ by a multiplication, another typical strength reduction transformation, is also benefical in our cost model, as indicated by the lower cost shown in Table I.

B. Existing optimisations in LLVM

The -instcombine pass facilitates local algebraic simplifications through pattern matching and replacement, constant folding, as well as a simple form of dead code elimination.

These local transformations either reduce the number of instructions, or normalise instructions into a canonical form. These normalisations are important, because they allow the other transformations to look for matches against fewer patterns, and create more constant folding opportunities.

Figure 2 shows examples for patterns that are applicable to the CellML equation systems. Eq. (2) moves constants to the right-hand side of commutative operations. Operations involving their respective neutral element are eliminated in Eq. (3). Subtractions and divisions are replaced in favour of commutative and simpler additions and multiplications with the inverse constant in Eqs (4, 5). Eq. (6) transforms a multiplication with -1 into a negation, which is encoded as $0 - x$, because the

$$c \oplus x \Rightarrow x \oplus c \tag{2}$$

$$x + 0, x \cdot 1 \Rightarrow x \tag{3}$$

$$x - c_1 \Rightarrow x + c_2 \quad \text{with } c_2 = -c_1 \tag{4}$$

$$x/c_1 \Rightarrow x \cdot c_2 \quad \text{with } c_2 = 1/c_1 \tag{5}$$

$$x \cdot (-1) \Rightarrow 0 - x \tag{6}$$

$$(0 - x) + c_1 \Rightarrow c_1 - x \tag{7}$$

$$(0 - x) \cdot c_1 \Rightarrow x \cdot c_2 \quad \text{with } c_2 = -c_1 \tag{8}$$

$$x^2 \Rightarrow x \cdot x \tag{9}$$

$$c_1 \oplus c_2 \Rightarrow c_3 \quad \text{with } c_3 = c_1 \oplus c_2 \tag{10}$$

$$(x \oplus c_1) \oplus c_2 \Rightarrow x \oplus c_3 \quad \text{with } c_3 = c_1 \oplus c_2 \tag{11}$$

$$(x \cdot c_1 + c_2) \cdot c_3 \Rightarrow x \cdot c_4 + c_5 \quad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{12}$$

$$(c_1 - x \cdot c_2) \cdot c_3 \Rightarrow c_4 - x \cdot c_5 \quad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{13}$$

Fig. 2. Sample transformations performed by -instcombine. x denotes an unknown value, the c_i are constants, $\oplus$ is either an addition or a multiplication, $\pm$ is either an addition or subtraction.

LLVM-IR does not contain a dedicated negation instruction. Eqs. (7, 8) are used to eliminate the extra subtraction where possible. The square operation is transformed from a call to the generic power function to a single multiplication in Eq. (9). Eqs. (10-13) show the application of constant folding. It is even performed on distributive expressions if the number of operations can be reduced thereby.

The transformations of Eqs. (11-13) are enabled only if unsafe floating-point transformations are allowed; Eq. (5) is safe when the constant's reciprocal is exact.

According to the previous section, the *replacement* of operations using Eqs. (5, 6) is beneficial in our cost model. All other transformations actually *eliminate* operations, and therefore lower the overall resource estimation as well.

Along these transformations, a simple store-to-load forwarding is performed. This retrieves the value at a known array index of **`ALGEBRAIC`** directly from the defining expression, circumventing the effect that the underlying pairs of array stores and loads would break the def-use chain in the intermediate representation.

In addition to -instcombine, we use the following passes:

-gvn is an implementation of the global value numbering algorithm [13] and identifies reusable subexpressions in the whole equation system. This includes store-to-load forwarding at global scope.

-simplifycfg exposes computations that are used on both sides of a conditional operator, and extracts them to be shared.

-dce globally cleans up any computations that became dead during one of the other transformations

We augment the IR representing the equation system with additional facts, exploiting domain knowledge: The array parameters are marked as non-overlapping, allowing the alias

analysis framework to conclude that all accesses to these arrays with different indices are independent, and all function calls are mapped to LLVM intrinsics, characterising them as side-effect free. This more precise information allows the value numbering and store-to-load forwarding passes to build a single connected directed acyclic graph (DAG) of the computations in the equation system, which leads to more transformation opportunities for both the existing and custom optimisations.

C. Computing higher-order powers

A generic power operator uses significantly more resources than a multiplication. However, if the exponent is a constant integer, we can replace the large generic power operator by a sequence of multiplications.

We use Knuth's binary exponentation method [23] (Figure 3) to construct the required multiplication instructions. Z is used to successively compute $X^{(2^k)}$ for $1 \leq k \leq \lfloor \log_2(P) \rfloor$. For every **'1'** bit in the binary representation of P (beginning with the least significant bit), Y is multiplied with the power of X corresponding to the bit's significance. The algorithm uses $\lfloor \log_2(P) \rfloor + \nu(P) - 1$ multiplications, where $\nu(P)$ is the number of ones in the binary representation of the exponent. The term -1 corrects the error due to one multiplication by 1.0 being always constructed, which can immediately be removed. Should *different* powers of the same value occur in the equation system, already-computed intermediate powers will be reused to calculate larger powers.

Our cost model allows to use up to eight multiplications, which is sufficient to compute powers as large as 46. This range covers all relevant cases in CellML models.

The binary exponentation method does not always yield the sequence with the minimal number of multiplications. The algorithm is a specialisation of the *optimal addition chain exponentiation* method, which constructs the multiplications according to a shortest addition chain for the exponent [23]. Finding such an optimal addition chain is a NP-complete problem [24]. Considering the relatively narrow range of exponents that typically occur in CellML models, it would be feasible to optimally pre-compute optimal realisations for all exponents. However, for the CellML models we examined, Knuth's method already computed the smallest solutions.

D. Additional normalisations

In order to create more candidates for the transformations presented in the next sections, we propose to expand the set of optimised distributive expressions involving two constants according to the Eqs. (14, 15) shown in Figure 4.a, complementing LLVM's existing normalisations.

E. A closer look at the exponential function

Exponential relations are common in the biological processes modelled by CellML descriptions. A pattern that occurs frequently in these models is $e^{x+c_1} \cdot c_2$. If both $c_1, c_2 \neq 0$, according to the power laws, we can either propagate and fold c_2 into the exponent or fold c_1 into the multiplication with c_2.

Input: X : Value, P : int
1: $N \leftarrow P$, Y $\leftarrow$ "1.0", Z $\leftarrow$ X
2: **loop**
3: remember whether N is odd
4: $N \leftarrow \lfloor N/2 \rfloor$
5: **if** N was odd **then**
6: Y $\leftarrow$ **new** "`mul` Z, Y"
7: **end if**
8: **if** $N = 0$ **then**
9: **return** Y
10: **end if**
11: Z $\leftarrow$ **new** "`mul` Z, Z"
12: **end loop**

Fig. 3. Knuth's binary exponentation method [23] for lowering X^P to multiplication instructions.

a) Additional normalisations

$$(x + c_1) \cdot c_2 \Rightarrow x \cdot c_2 + c_3 \quad \text{with } c_3 = c_1 \cdot c_2 \tag{14}$$

$$(c_1 - x) \cdot c_2 \Rightarrow x \cdot c_3 + c_4 \quad \text{with } c_3 = -c_2 \text{ and } c_4 = c_1 \cdot c_2 \tag{15}$$

b) Exponential function optimisations

$$e^{x+c_1} \cdot c_2 \Rightarrow e^x \cdot c_3 \quad \text{with } c_3 = e^{c_1} \cdot c_2 \tag{16}$$

$$e^x \cdot c_1 \Rightarrow e^{x+c_2} \quad \text{with } c_2 = \ln(c_1) \text{ when } c_1 > 0, n_{\text{uses}}(e^x) = 1 \tag{17}$$

c) Assembling of multiplications with constant factors

$$x \cdot c_r \Rightarrow x \cdot c_s + x \cdot c_s \quad \text{when } c_r = 2 \cdot c_s \tag{18}$$

$$x \cdot c_t \Rightarrow x \cdot c_u + x \quad \text{when } c_t = c_u + 1 \text{ and } c_t, c_u > 0 \tag{19}$$

$$x \cdot c_v \Rightarrow x \cdot c_w - x \quad \text{when } c_v = c_w - 1 \text{ and } c_v, c_w < 0 \tag{20}$$

Fig. 4. Domain-specific optimisations.

In both cases, no new operation is required, therefore we can either save an addition or a multiplication.

If either constant is 0, the immediate constant folding opportunities are not present, leading to the introduction of a new operation. We trade a multiplication for an addition when $c_1 = 0$, or vice versa in case $c_2 = 0$. However, the latter case can be viable, too: If the exponential function operator can be reused from elsewhere and thus be eliminated from this instance, the cost reduction is $c(*) - c(exp) - c(+) \approx -0.1$.

To this end, we propose the strategy depicted in Figure 4.b using Eqs. (16, 17). LLVM keeps track of the number of uses n_{uses} for any value in the intermediate representation, so no additional analysis is required to obtain this information.

F. Assembling constant-factor multiplications

After the presented optimisations and normalisations, we observe that the equation systems contain sets of multiplica-

Input: M_x : map(Constant $\rightarrow$ Instruction)

Factors $\leftarrow [c_1, \ldots, c_n | \forall c_i, c_j : |c_i| \leq |c_j|, c_i \in \text{keys}(\mathrm{M}_x)]$

for all $c \in$ Factors **do**
  if $c \cdot 2 \in \mathrm{M}_x$ **then**
    ensureAvailability(M_x, c, $c \cdot 2$)
    $\mathrm{M}_x[c \cdot 2] \leftarrow$ **new** "**add** $\mathrm{M}_x[c]$, $\mathrm{M}_x[c]$"
  end if
  if $c + 1 \in \mathrm{M}_x \wedge c > 0$ **then**
    ensureAvailability(M_x, c, $c + 1$)
    $\mathrm{M}_x[c + 1] \leftarrow$ **new** "**add** $\mathrm{M}_x[c]$, $\mathrm{M}_x[1]$"
  else if $c - 1 \in \mathrm{M}_x \wedge c < 0$ **then**
    ensureAvailability(M_x, c, $c - 1$)
    $\mathrm{M}_x[c - 1] \leftarrow$ **new** "**sub** $\mathrm{M}_x[c]$, $\mathrm{M}_x[1]$"
  end if
end for

Fig. 5. Algorithm to assemble constant-factor multiplication instructions.

tions $x \cdot c_i$ of a value x with pairwise distinct constants c_i.

Among these factors, there exist linear dependences that can be exploited to lower a multiplication into an addition. For example $x \cdot 0.2$ can be replaced by $x \cdot 0.1 + x \cdot 0.1$, given that $x \cdot 0.1$ is available somewhere in the equation system. Analogously, $x \cdot 3.0$ can be expressed as an addition by using $x \cdot 2.0 + x$, assuming that $x \cdot 2.0$ is already available. Eqs. (18-20) in Figure 4.c formalise this approach.

The optimisation is implemented through the algorithm in Figure 5. It operates on a map M_x for every value x that is part of a multiplication with a constant factor. M_x maps the constant factors to the multiplication operation that uses them, e.g. $c_i \rightarrow x \cdot c_i$. To find pairs of factors that are eligible for the proposed transformation, we traverse a list of all occurring factors in ascending order of the absolute values. This particular order is important, because, as an example, 0.1 is thereby visited before 0.2, and -2 is visited before -4, allowing us to handle even chains of factors (e.g. $0.1, 0.2, 0.4$) correctly.

For each factor, the applicability of Eqs. (18-20) is checked. If $\mathrm{M}_x[c]$ does not dominate the multiplication we want to replace, a copy of $\mathrm{M}_x[c]$ has to be inserted before the replacement candidate by the helper function ensureAvailability; the original $\mathrm{M}_x[c]$ will be eliminated by common subexpression elimination later. Afterwards, the respective new instruction is created and M_x updated accordingly.

For brevity, we do not show the code to ensure that every multiplication is only replaced once, as Eqs. (18) and (19/20) can both match. We assume that the factor 1 is present in M_x and initially marked as non-replaceable, due to the fact that we cannot represent x as an expression of itself.

This transformation could be extended to transform arbitrary integer factors into a series of additions (similar to the approach we used to optimise power computations). However, given the cost model of Table I, a multiplication could be replaced by at most two additions, limiting the generality of the optimisation.

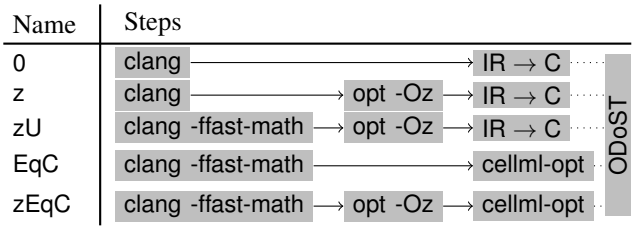

Fig. 6. Compilation flows.

IV. Evaluation

We implemented the optimisation strategy from the previous section in an LLVM-based source-to-source optimiser called cellml-opt. While LLVM generally does not generate C code for arbitrary programs, we can easily do so for the limited forms of IR in the CCGS-created simulation models, once again exploiting domain knowledge.

We evaluate the LLVM-extended ODoST hardware synthesis in five flows (shown in Figure 6). These range from a baseline flow 0 that just uses the LLVM front-end clang and performs no optimisations of its own, to a flow z that uses LLVMs generic optimiser opt for aggressive size reduction. The flow EqC adds the new domain specific-optimisations of our cellml-opt pass, which are combined with the generic opt in zEqC. All of the cellml-opt flows employ unsafe floating-point transformations. We currently use a simple driver that iterates the existing and new optimisations in a fixed order. For reference, we also show the impact of these transformations with just the generic optimiser in the flow zU.

We use tools from and link cellml-opt against the LLVM compiler framework, version 3.5.1. The low-level synthesis of the accelerator modules generated by ODoST is performed by Altera Quartus II, version 14.0, targeting an Altera Stratix IV EP4SGX530 FPGA device.

The CellML model repository [25] contains a vast number of models from biology as well as other disciplines. We chose to filter out models for which *any* of the following criteria are true: a) they are too simple (have fewer than 20 equations), b) they have low quality (a curation rating below 2 stars), c) do not converge in the fixed input interval used for testing (e.g., due to singularities in the interval), d) contain arithmetic operators of functions not yet supported by the current cellml-opt implementation (e.g., integer modulo, $\arccos$).

Testing and evaluation is performed by a simple software driver calling the CCGS-generated code (changed to single-precision arithmetic) and performing 1000 integration steps of 10^{-6} each, starting with an initial value of the integration variable of 1.0. We use the results obtained in this manner to evaluate the accuracy of our optimised models: The absolute error is the difference between the final integration values computed by these models and the unoptimised reference, the relative error normalises this to the reference value.

Applying the selection criteria a)…d), we thus obtain 146 CellML models for our evaluation. We will consider the results

of the evaluation first in an aggregated fashion, and then in detail for four sample models.

A. CellML repository optimisation results

For one particular model, Cooper et al. [21] state that an error below 0.01 % is smaller than the precision obtainable in "wet biology" lab experiments. However it is difficult to define a generic threshold, as the acceptable inaccuracy will depend on the context of the specific simulation goals. We will thus examine the performance of our optimised accelerators across a number of error thresholds.

Figure 7.a shows the sum of the estimated area cost across all 146 models when compiled with the different flows, normalised to the summed costs of using the unoptimised flow 0 for the models. At a loose error threshold of 2.0%, our EqC optimisations can lead to area savings of up to 29.2% across the entire model suite. With increasingly tighter error thresholds, however, fewer models can be compiled with the more aggressive area-reducing but floating-point unsafe transformations. Instead, their larger implementations generated by omitting the unsafe transformations have to be used. Applied in isolation, the safe general-purpose optimisations (flow z) lead to a 14.6% reduction of area over the original ODoST synthesis flow.

Figure 7.b gives a different perspective of the data and shows for how many of the 146 models each flow resulted in the least estimated area cost at the given error threshold. For loose error bounds, our unsafe domain-specific flow EqC gives the best results for most models. For tighter bounds, more and more models must refrain from using unsafe optimisations to maintain accuracy and have to use the general-purpose z flow. Only very few models profit from combining the generic and domain-specific optimisation (flow zEqC).

B. Detailed analysis and hardware synthesis

We pick four models of increasing complexity (the same as were used in [2]) for a detailed analysis of our approach (Table II), including a look at the actual Quartus-fitted FPGA implementations of the HAMs generated by ODoST. As the EqC and zEqC flows produce identical results, we omit the latter from the discussion.

The data shows that the custom transformations are successful in reducing the number of multiplications, exponential functions, and power operators. The "Other" row covers the logarithm, floor, absolute value, comparison and negation operators, for which no special transformations are currently implemented. The domain-specific transformations are applied after the replacement of all divisions by a constant with multiplications by the constant's reciprocal, similar to that performed by the generic zU flow. In combination, the domain-specific optimisations thus lead to the smallest accelerator sizes (allowing maximum parallelism using multiple instances per chip). Despite the unsafe floating-point transformations, only one model (Hilgemann & Noble [26]) has an error threshold significantly exceeding the precision of "wet biology" experiments (0.32% vs 0.01%). Using the less aggressive (but still floating-point unsafe!) transformations of the zU flow, even its accuracy can be sufficiently increased to lie below the experimental threshold, and still save around 12% of DSP blocks over the unoptimised version. Note that these HAMs perform the integration step in addition to the evaluation of the equation system, which requires a small number of extra additions and multiplications.

Across all of these models, the area savings of the actual FPGA implementations can reach up to 25% for large models such as ten Tusscher et al. [27], allowing an accelerator for this model to fit the target device for the very first time. The clock frequency, and thus the throughput of the saturated accelerator pipelines, is almost unaffected in the optimised versions, and never falls below the 100 MHz typically used as the reference clock for the FPGA board.

V. Conclusion and Future Work

We presented prefixing an existing domain-specific high-level synthesis flow with an LLVM-based optimisation pipeline, with the aim to reduce the hardware area requirements of CellML-based simulation accelerators.

In addition to selectively applying general-purpose techniques, we also introduced domain-specific transformations tailored for hardware synthesis. We also investigated the applicability of unsafe floating-point transformations for the cell simulation domain. We discovered that they can lead to area reductions of up to 25%, if computation inaccuracies of 0.01% … 2% are acceptable.

The cellml-opt tool was initially intended as a front- and middle-end to ODoST, but we expect that its implementation as a source-to-source optimiser will allow it to be used in conjunction with other tools from the CellML ecosystem as well. We also plan to investigate whether it is possible to make the use of unsafe transformations context-aware, such that their application can be restricted to areas where they cause only limited accuracy loss.

cellml-opt is available as open-source from the *Downloads* section of http://www.esa.cs.tu-darmstadt.de.

References

[1] A. A. Cuellar, C. M. Lloyd, P. F. Nielsen, D. P. Bullivant, D. P. Nickerson, and P. J. Hunter, "An Overview of CellML 1.1, a Biological Model Description Language," 2003.

[2] T. Yu, C. Bradley, and O. Sinnen, "Automatic Hardware Acceleration for Biomedical Model Integration using ODoST," *ACM Trans. on Reconfigurable Technology and Systems*, submitted 2015.

[3] T. Yu, J. Oppermann, C. Bradley, and O. Sinnen, "Performance Optimisation Strategies for Automatically Generated FPGA Accelerators for Biomedical Models," *Concurrency and Computation: Practice and Experience*, submitted 2015.

[4] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Intl. Symp. on Code Generation and Optimization, CGO*, 2004.

[5] A. Garny, D. P. Nickerson, J. Cooper, R. Weber dos Santos, A. K. Miller, S. McKeever, P. M. F. Nielsen, and P. J. Hunter, "CellML and associated tools and techniques." *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 366, 2008.

[6] A. K. Miller, J. Marsh, A. Reeve, A. Garny, R. Britten, M. Halstead, J. Cooper, D. P. Nickerson, and P. F. Nielsen, "An overview of the CellML API and its implementation." *BMC bioinformatics*, vol. 11, 2010.

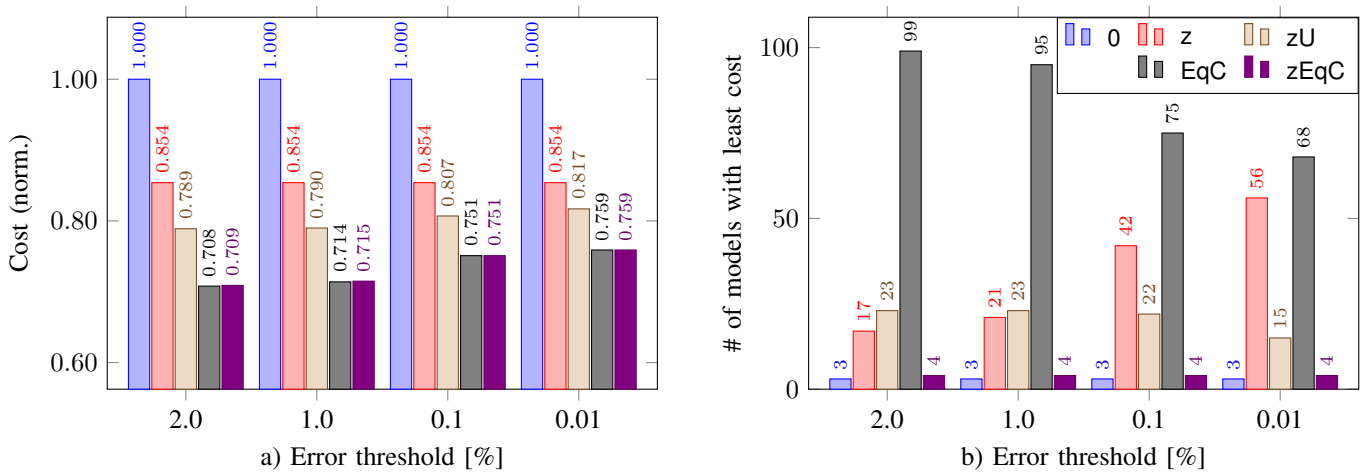

Fig. 7. a) Summed estimated area cost of 146 models per flow, normalised to the summed cost of the results of the unoptimised 0 flow, and b) number of models with the least estimated cost per flow.

TABLE II
OPTIMISATION AND SYNTHESIS RESULTS FOR THE FOUR SAMPLE MODELS.

	Hodgkin & Huxley [9]				Beeler & Reuter [28]				Hilgemann & Noble [26]				ten Tusscher et al. [27]			
	0	z	zU	EqC	0	z	zU	EqC	0	z	zU	EqC	0	z	zU	EqC
Add/Sub	23	**21**	**21**	23	67	64	**56**	68	117	111	**106**	114	171	169	**166**	170
Mul	**17**	**17**	22	21	52	51	63	**47**	132	127	131	**113**	**139**	141	171	155
Div	10	10	**4**	**4**	28	27	**15**	**15**	52	49	41	**40**	123	120	80	**79**
Exp	6	6	6	6	25	25	25	**24**	21	21	18	**17**	52	52	50	**48**
Pow	2	2	2	**0**	1	1	1	**0**	7	5	3	**2**	26	11	10	**2**
Other	10	9	**4**	**4**	15	16	**8**	**8**	20	23	20	**18**	26	29	25	**21**
Total	68	65	59	**58**	188	184	168	**162**	349	336	319	**304**	537	522	502	**475**
ALM (est.) [%]	15.6	15.1	11.3	**9.7**	39.8	38.8	**30.3**	**30.3**	72.2	67.9	60.0	**58.2**	147.3	131.5	109.6	**99.9**
DSP (est.) [%]	13.9	13.9	15.8	**9.4**	30.0	29.6	34.3	**24.8**	83.9	75.9	70.8	**60.5**	150.2	105.6	113.9	**83.0**
Cost (norm.)	1.00	0.98	0.93	**0.64**	1.00	0.98	0.92	**0.79**	1.00	0.92	0.84	**0.76**	1.00	0.80	0.75	**0.62**
Rel. error [%]	-	-	$\frac{17}{10^4}$	$\frac{19}{10^4}$	-	-	$\frac{25}{10^3}$	$\frac{40}{10^3}$	-	-	$\frac{13}{10^6}$	$\frac{32}{10^2}$	-	-	$\frac{54}{10^5}$	$\frac{12}{10^4}$
ALM (syn.) [%]	18.7	18.3	16.8	**15.7**	43.1	43.5	40.2	**40.0**	83.8	**83.3**	84.7	83.8	121.9	104.9	101.2	**96.8**
DSP (syn.) [%]	13.5	13.5	15.4	**8.6**	30.0	29.6	33.9	**23.6**	81.2	74.3	68.9	**59.8**	100.0	99.0	99.2	**79.9**
f_{max} [MHz]	135	**141**	130	133	132	129	**134**	**134**	129	131	**133**	116	-	-	-	**111**

[7] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, 1991.

[8] C. Farnum, "Compiler support for floating-point computation," *Software: Practice and Experience*, vol. 18, no. 7, 1988.

[9] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its applications to conduction and excitation in nerve," *J. of Physiology*, vol. 117, 1952.

[10] IEEE, "754-2008 IEEE Standard for Floating-Point Arithmetic," Tech. Rep., 2008.

[11] F. De Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test of Computers*, vol. 28, 2011.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2006.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Computing Surveys*, vol. 35, no. 3, 2003.

[15] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. on Programming Languages and Systems*, vol. 22, no. 2, 2000.

[16] K. D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," in *Proc. of the ACM SIGPLAN 1999 Conf. on Programming Language Design and Implementation*, 1999.

[17] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, "The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 8, no. 3, 2015.

[18] B. Buyukkurt, J. Cortes, J. Villarreal, and W. A. Najjar, "Impact of High-level Transformations Within the ROCCC Framework," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 4, 2010.

[19] R. Kastner, A. Hosangadi, and F. Fallah, *Arithmetic optimization techniques for hardware and software design*. Cambridge University Press, 2010.

[20] X. Gao and G. A. Constantinides, "Numerical Program Optimization for High-Level Synthesis," in *Proc. of the 2015 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays - FPGA '15*, 2015.

[21] J. Cooper, S. McKeever, and A. Garny, "On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations," in *Proc. of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006.

[22] N. D. Jones, "An introduction to partial evaluation," *ACM Computing Surveys*, vol. 28, 1996.

[23] D. E. Knuth, *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*. Pearson Education, 2014.

[24] P. Downey, B. Leong, and R. Sethi, "Computing Sequences with Addition Chains," *SIAM J. on Computing*, vol. 10, no. 3, 1981.

[25] C. M. Lloyd, J. R. Lawson, P. J. Hunter, and P. F. Nielsen, "The CellML Model Repository," *Bioinformatics*, vol. 24, no. 18, 2008.

[26] D. W. Hilgemann and D. Noble, "Excitation-Contraction Coupling and Extracellular Calcium Transients in Rabbit Atrium: Reconstruction of Basic Cellular Mechanisms," *Proc. of the Royal Society of London B: Biological Sciences*, vol. 230, no. 1259, 1987.

[27] K. H. W. J. ten Tusscher, D. Noble, P. J. Noble, and A. V. Panfilov, "A model for human ventricular tissue." *American journal of physiology. Heart and circulatory physiology*, vol. 286, 2004.

[28] G. W. Beeler and H. Reuter, "Reconstruction of the action potential of ventricular myocardial fibres," *J. of Physiology*, vol. 268, no. 1, 1977.