Expression Synthesis in Process Networks generated by LAURA

Claudiu Zissulescu Bart Kienhuis Ed Deprettere
Leiden Institute of Advanced Computer Science (LIACS),
e-mail: {claus,kienhuis,edd}@liacs.nl
Presented at the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP2005), July 23 – 25, 2005, Samos, Greece

Abstract

The COMPAAN/LAURA *[18] tool chain maps nested loop applications written in Matlab onto reconfigurable platforms, such as FPGAs.* COMPAAN *rewrites the original Matlab application as a Process Network in which the control is parameterized and distributed. This control is given as parameterized polytopes that are expressed in terms of pseudo-linear expressions. These expressions cannot always be mapped efficiently onto hardware as they contain multiplication and integer division operations. This obstructs the data flow through the processes. Therefore, we present in this paper the Expression Compiler that efficiently maps pseudo-linear expressions onto a dedicated hardware datapath in such a way that the distributed and parameterized control* never *obstructs the data flow through processors. This compiler employs techniques like number theory axioms, method of difference, and predicated static single assignment code.*

1 Introduction

The aim of the COMPAAN compiler [8, 18] is to automatically derive a parallel description from a nested loop application written in a standard programming language like C or Matlab. The applications COMPAAN targets are compute kernels that belong to the domain of multi-media, imaging, bioinformatics, and classical signal processing. These application are computationally intensive and typically need to operate under real-time constraints. The output of COMPAAN are Kahn Process Network (KPN) specifications. A KPN is a deterministic model of computation that expresses an application in terms of distributed memory and distributed control. Once the process network is created, the individual processes can either be described in Java [4] or C++ code [5], or as synthesizable VHDL network of processors suitable for mapping onto FPGAs [20]. The

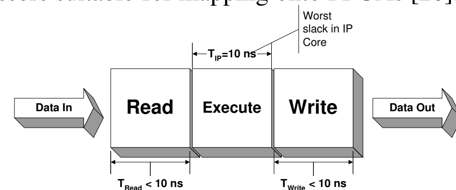

Figure 1. Running an IP core efficiently in a LAURA Processor

KPN synthesis to hardware is done by the LAURA tool [20]. During synthesis, each KPN process is first mapped to a *virtual processor* that consists of a Read, an Execute and a Write Unit. The Read and Write Units execute a particular control program that is derived by COMPAAN. This program represents the distributed control of a process network. The control program determines at each firing of the Execute Unit from where data needs to be read and to where data needs to be written. The Execute Unit embeds an IP core that implements the functionality of the assignment statement in the original Matlab or C program. Each IP Core executes a firing in a particular clock period, as given by its worst slack. This clock period becomes an important design constraints for our hardware processor, as the Read and Write Units need to determine the next read and write operation in less than this clock period. Only then do the Read and Write Unit not obstruct the data flow through the Execute Unit. In Figure 1, we show an IP Core that executes in 10 ns. As a consequence, the Read and Write Units need to calculate the next control signals in less than this 10 ns.

The control program in the Read and Write units is expressed in terms of parameterized *polytopes*. By evaluating these, we can support parameterized control in hardware [6, 13]. The parameterized polytopes are repeatedly evaluated at run-time. If a particular iteration is within the space defined by the

polytope, it means that data needs to be read or written. An example of a polytope is given in Figure 2. In it, the variables M and N are parameters, while i, j, and $stage$ are iterators from corresponding code. Each expression in the polytope may also contain pseudo-linear terms like a DIV (e.g., DIV(i+1,2)).

$$\mathcal{P} = \begin{cases} M-3 & \geq 0 \\ N-2 & \geq 0 \\ -j+3*M & \geq 0 \\ j-M & \geq 0 \\ 2*i-1 & \geq 0 \\ i+2 & \geq 0 \\ -stage+N-1 & \geq 0 \\ stage-1 & \geq 0 \\ -i+2*DIV(i+1,2)-1 & \geq 0 \\ 2*DIV(M+1,2)-M-1 & \geq 0 \end{cases}$$

Figure 2. Example of a set of expressions that defines a polytope.

Checking whether an iteration point ($stage$, i, j) is enclosed by the polytope P or not, we have to evaluate all the condition contained by the polytope (see Figure 2). In software we implement this evaluation as a cascade of if-statements. The speed of evaluation is however low, as the evaluation of each expression happens in a sequential manner. Evaluating the polytopes on hardware can be done much faster, as each expression can be evaluated in *parallel*. Nevertheless, it is still difficult to evaluate the expressions as part of the Read or Write unit in Figure 1 in less then $10ns$. Complex operations such as multiplication and integer division may easily take more than the $10ns$ for running a stage of the IP core. In this paper, we introduce the Expression Compiler that compiles the polytope expressions in a hardware data-path using a number of techniques to reduce the evaluation time. As a consequence, we can evaluate arbitrary expressions in the Read and Write units faster than the Execute Unit.

The techniques we present as part of the Expression Compiler take advantage of the repetitive nature of the polytope evaluation, and algorithmic techniques like strength reduction of complex operations to conditional statements, addition, shifts and Look Up Tables (LUTs). Also, we propose that all expressions to be compiled in a static single assignment form suitable for future low-level optimizations, e.g. multiplexors reductions, bit-width computation, and pipelining. However, we do not address the issue of the generating the for-loops that include the polytope $\mathcal{P}$. In this paper we address only the issues of the generating hardware datapaths for the expressions within the polytope.

We start by defining the expressions in Section 2. In Section 3, we describe related work. Section 4 presents our approach to convert expressions efficiently to hardware in a two-step approach. In Section 5 and Section 6, we present the techniques to simplify expressions. The predicated single assignment form is explained in Section 7. Experimental results are given in Section 8, and we conclude this paper in Section 9.

2 Pseudo Linear Expressions

A polytope is represented by a set of terms that are linear or pseudo linear and is defined as:

$$Expression = \sum_{k=0}^{n} c_k * \begin{cases} MOD(Expression, divider); \\ DIV(Expression, divider); \\ i_k; \\ 1. \end{cases} \quad (1)$$

where c_k and c_n are constants, i_k is a for-loop iterator, MOD is the remainder of the integer division of in an expression by a constant divider, and the DIV is the integer division in an Expression by a constant divider.

3 Related Work

A KPN network generated by COMPAAN may be simulated using software KPN simulators where all expressions that define a polytope are evaluated in a sequential order. In [7], a similar approach has been tried for a hardware implementation, but for only a very limited set of expressions (no multiplications and pseudo linear operators). To allow for an efficient compilation of expressions to hardware, we investigated a more flexible approach based on two observations. Our first observation is that each expression can be evaluated in parallel as they represent a part of a geometrical figure in a n-dimensional space. A second observation is that the total evaluation time of an expression can take longer than the evaluation of an IP block embedded in our network. This is due to the presence of complex operations such as multiplication and integer division. The elimination of division and modulo operations from a sequential programs has been discussed in [16, 12]. Also the PICO project [15] employs similar techinques to avoid MOD and DIV operations. Our target is to generate a custom datapath implementation, and hence, additional issues have to be taken into account like hardware mapping, critical path delay, and the size of the implementation in hardware.

4 The Approach

For a fast and efficient implementation of the expression set given in Figure 2 we simplify them into

expressions that use only additions, LUT, and shifts. Thus, they can be executed in less then a single cycle of the IP core embedded in the Execute Unit. Moreover, the simplified expressions is represented in a data-structure that allows us further manipulation to obtain an optimal hardware implementation in terms of speed and area (e.g. pipeline, retiming and multiplexer reduction). For that reason, we have broken down the conversion from expressions to hardware in two steps as shown in Figure 3. In the first step, the input expressions are simplified using high-level optimizations that are platform independent. In the second step, the expressions are manipulated to obtain better performing hardware by taking advantage of mid and low-level optimizations that are platform dependent.

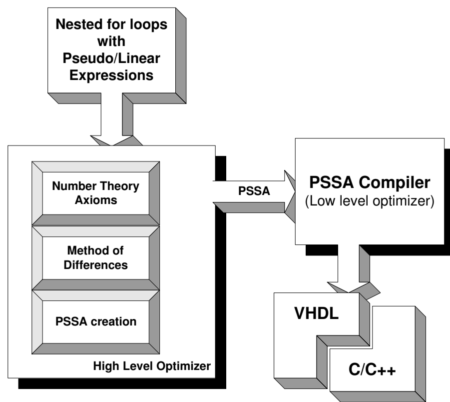

Figure 3. The Expression Compiler Flow

The High-Level Optimizer implements high-level and platform independent optimizations. It converts an expression in three steps to a simplified structure that allows further optimization.

Number Theory Axioms is exploited to reduce the strength of expressions to simpler expressions. Also, expressions that contain DIV terms are converted to MOD terms as they are simpler to realize in hardware.

Method of Difference is used to replace multiplications by additions, taking the advantage of a repetitive evaluation of the polytopes within KPN control.

Predicated Static Single Assignment is invoked to perform mid and low-level optimizations, like constant propagation, dead-code elimination, and retiming for gaining the best hardware performance.

The Predicated Static Single Assignment Compiler optimizes the input *Predicated Static Single Assignment* (PSSA) [3] code for a particular target architecture, taking advantage of today's research in PSSA compilation techniques [3]. The PSSA form is suitable for optimizations either for a microprocessor [14] or for a reconfigurable platform such as FPGA [17] in which we are interested. For an FPGA platform target, the desired operations are: the reduction of the number of variables used, multiplexer optimizations, bit-width, and LUT synthesis for non linear terms. Finally, the resulting PSSA is mapped onto a hardware description language like VHDL, or Verilog by associating a hardware equivalent to each operation/node of the PSSA.

5 Number Theory Axioms

A polynomial expression is composed of linear terms and pseudo linear terms like MOD and DIV. These pseudo terms require special compilation techniques to be realized efficiently and fast in hardware. An important assumption is that a MOD term can be realized more efficiently in hardware than a DIV term. A MOD term can be realized, as we will show when presenting examples, using look up tables (LUTs) and simple counters. Also, we avoid calculating the actual integer division by a constant, leading to a realization that uses less bits. COMPAAN typically generates polytopes that contain DIV terms when pseudo linear terms are involved. The $b * DIV(a, b)$ form is the most frequently occurring form of a DIV operation found in our polytopes. See, for example, the expressions in Figure 2. Since we want to work with MODs rather than DIVs, we make use of Optimization 5.1 when possible.

Optimization 5.1

If a DIV term is defined as $b * DIV(a, b)$, then it is simplified as $a - MOD(a, b)$.

An observation is that Optimization 5.1 can easily be extended to handle numbers that are a multiple of b. Conversely, the optimization can also be used to rewrite MOD terms into DIV terms. There are cases that a DIV term cannot be converted into a MOD term. In such case, we employ modern software compiler techniques to reduce the cost of integer division [1, 9, 10]. These techniques are based on so-called scaled reciprocals. In general, the techniques transform an integer division into a multiplication with a constant and a shifts expressed, see [10]. Also, we can scale the entire expression with the

constant b to arrive to an expression $b * DIV(a, b)$ in which case Optimization 5.1 holds.

The bit-width result of a DIV(a,b) term is $\lceil \log_2(a) \rceil - \lceil \log_2(b) \rceil + 1$, while the bit-width result of a MOD(a,b) term is $\lceil \log_2(b) \rceil$. In our flow, usually, we may assume that $b << a$, and we can say that the number of bits needed to represent the MOD term is less then the number of bits needed to represent a DIV term. This reinforces our goal to work with MOD rather than DIV terms.

Besides implementing operations that rewrite DIV terms in MOD terms, we implemented additional *strength reduction* operations to further optimize an expression in a systematic way to calculated the expression more efficiently using equivalent operators that are cheaper on a target platform. Next, we give some number theory axioms which we have implemented in our High-Level Optimizer to simplify modulo operations. Even when the simplification does not immediately eliminate operations, it reduces the complexity of operations and may lead later on to further optimizations.

Optimization 5.2
Simplifies modulo expressions using the following algebraic simplification rules where x and y are variables, and a, b, and d are constants.

$$\begin{aligned} MOD(a*x+b*y,d) &= MOD(MOD(a,d)*x+ \\ &\quad + MOD(b,d)*y,d) \\ MOD(x+y,d) &= MOD(MOD(x,d)+ \\ &\quad + MOD(y,d),d) \\ MOD(x*y,d) &= MOD(MOD(x,d)* \\ &\quad * MOD(y,d),d) \end{aligned}$$

This optimization is particularly interesting when the expression of the MOD term contains constant values. Using Optimization 5.2, we can remove these constants from the run-time calculation as we can compute them at compile time. Also, constants are scaled to small values, i.e., MOD(i+10,3) is converted into MOD(MOD(i,3)+1,3). A special case is when the divider is a constant of power two. In that case, we can apply Optimization 5.3.

Optimization 5.3
If the divider of the MOD operation is positive and is a power of two, then the modulo expression can be simplified to a bitwise AND operation:

$$MOD(x,\ 2^n) = x\ AND\ (2^n - 1)$$

where $(2^n - 1)$ is a string of n ones.

After applying Optimization 5.2, we obtain a MOD operation with a known range of $r = [0,(d-1)], r \in \mathbb{N}$. For small dividers, this MOD operation can be implemented using a Look Up Table (LUT) as in Optimization 6.1 below.

6 Method of Difference (MoD)

Within each processor, a sequence schedule is executed. This schedule is captured by for loops. The for-loops define iteration points for which the polytope needs to be evaluated, e.g., for iteration point ($stage$,i,j). Because of the for-loops, a polytope, hence the expressions of the polytope, are evaluated repeatedly. This repetitive behavior can be exploited to simplify the evaluation of our expressions by removing all multiplications and convert them into additions. This is an important step, as a multiplication takes more FPGA resources and time compared to an addition.

The technique that exploits the repetitive behavior is called the *Method of Differences* (MoD) [2], as it is based on using differences of the terms of an expression to calculate the next value. Although the method of differences can be applied to a polynomial of any degree, we are dealing only with pseudo linear expressions and thus polynomials of degree one. Pseudo-linear terms such as MOD we can use the following optimizations.

Optimization 6.1
The modulo expressions that creates discontinuities within the iteration space can be simplified using a conditional statement.

Let there be given a modulo expression in a loop of the following form:

```
for j = L to U,
  exp = MOD(f(j),divider);
end
```

Where $f(j)$ is a linear function defined as $f(j) = f_0 + c * j$, with f_0, c and $divider$ constant. Then the loop can be transformed to the following:

```
for j = L to U,
    if j = L then
        exp = MOD(f0 + c*L, divider);
    else
        exp = MOD(exp + c, divider);
    end if
end
```

If L is a constant, then value exp on the TRUE branch of the if statement is a constant and can be computed at compile time. Otherwise, Optimization 6.1 is applied recursively to further reduce the multiplication operation to additions. Optimization 6.1 compiles a MOD term into operations that makes use of LUTs. These LUTs have always the maximum table length equal with $2 * divider - 1$ after applying Optimization 5.2. The bit-width is equal with $\lceil log_2(divider - 1) \rceil$. Hence, a LUT implementation for small values of $divider$ is very well achievable.

Optimization 6.2

Let $exp_{k+1} = MOD(exp_k + c, d)$ be a MOD operation in a nested for loop, then the operation can be written as:

```
for j = L to U,
   a = exp + MOD(c,d);
   if a >= d then
       a = a - d;
   elsif a < 0 then
       a = a + d;
   end if
   exp = a;
end
```

7 Predicated Static Single Assignment

The *Predicated Static Single Assignment* (PSSA) [3] form is suitable for mid and low-level optimizations either for micro-processors [14] or for reconfigurable platforms such as FPGAs [17]. We are primarily interested in optimization for a FPGA platform. The *Static Single Assignment* form requires that every variable within a computation is assigned a value only once, thereby explicitly expressing the data-dependency between operations. The Static Single Assignment form is almost equivalent to a Dependency Graph (DG), which is a very suitable form for hardware implementations. For example, variables for intermediate results correspond to nothing more than wires that are required anyway to perform the computation. By extending SSA with predication, every statement in the original computation is tagged with a *guard* that controls whether or not a statement is actually executed. Advanced techniques [17, 19] can be applied on a PSSA to optimize its output for the FPGA platform. Examples of mid-level optimizations are dead code elimination, constant propagation, and retiming. Examples of low-level optimizations are minimization of the uses of multiplexors and bit-width minimization of the PSSA variables.

In the PSSA code, each assignment statement is predicated with a condition that may be always true. Each variable is given an unique name to make sure a value is assigned only once to a variable in an evaluation, as required by the Static Single Assignment form.

The PSSA form can be efficiently used for low-level optimizations such as bit-width optimization. This analysis is very useful as it drastically improves both the area usage and the performance. A data-path operating on 5 bit integers is smaller and faster than an operation on 16 bit integer. Since the variables in an expression depend only on the loop indices, the bit-width of all operators can be derived from the original loop indices bit-width. The loop indices depend on the upper and lower loop bounds. Suppose that U_i is the upper bound for loop index i, then its bit-width is given by $w_i = \lceil \log_2(U_i) \rceil$. Using this information and the fact that all the operation in a PSSA tree are additions, we can propagate the bit-width constraint along the DG structure of the PSSA. Using Equation 2, in which variables in_1 and in_2 have a particular bit width, we calculate the required bit-width for the result of the addition.

$$w(in_1, in_2) = max(w(in_1), w(in_2)) + 1 \quad (2)$$

The result of the PSSA compiler is efficiently mapped in hardware as a dedicated data-path. We do not address the problem of generating the hardware for the for-loops (which are implemented as counters), but we are interested only in a efficient derivation of the hardware data-path for expressions in a polytope.

8 Example

In this section we show typical results obtained with our Expression Compiler, based on one complex examples. We used the Symplify 7.2 tool for synthesis and the Xilinx ISE 6.2 tool for hardware mapping on a Xilinx xc2v40 platform.

Consider the example given in Figure 4. It shows the linear expression $81 * i + 15 * DIV(j, 5) + MOD(5 * k, 3) - 78$ that is nested within three for-loops. We want to map this expression to hardware using the Expression Compiler as the expression contains multiplications and pseudo-linear terms. Applying the number theory axioms will help in this case to strength reduce the DIV and the MOD operators. In the first phase the DIV term is converted

```
for (i = 2; i<=1091; i++) {
 for (j = 1; j<=i-1; j++) {
  for (k = i+1; k<=523; k++) {
   exp = 81*i + 15*DIV(j,5) + MOD(5*k,3) - 78;
  }// end k
 }// end j
}// end i
```

Figure 4. A pseudo-linear expression

to a MOD term using Optimization 5.2, as follows: $81 * i + (3 * j - 3 * MOD(j, 5)) + MOD(5 * k, 3) - 78 \Rightarrow 81 * i + 3 * j - MOD(3 * j, 15) + MOD(5 * k, 3) - 78$. The term $MOD(5 * k, 3)$ is further strength reduced to $MOD(2 * k, 3)$ using Optimization 5.3. Next, the expression is compiled the MoD code, as it is described in Section 6. The code we obtain after applying the MoD technique is given in Figure 5. This code is equivalent to the original code, but all multiplications and pseudo-terms are replaced by additions and strength reduced

```
 1. for (i = 2; i<=1091; i++) {
 2.     if (i == 2) {
 3.        k1 = 1;
 4.     } else {
 5.        k1 = MOD(k1 + 2,3);
 6.     }
 7.     if (i == 2) {
 8.        i0 = 162;
 9.     } else {
10.        i0 = i0 + 81;
11.     }
12.     for (j = 1; j<=i-1; j++) {
13         k2 = MOD(k1 + 2,3);
14.        if (j == 1) {
15.           j0 = 3;
16.        } else {
17.           j0 = j0 +  3;
18.        }
19.        if (j == 1) {
20.           modj1 = 3;
21.        } else {
22.           modj1 = MOD(modj1 + 3,15);
23.        }
24.        for (k = i+1; k<=523; k++) {
25.           if (k == i+1) {
26.              k3= k2;
27.           } else  {
28.              k3 = MOD(k3 + 2,3);
29.           }
30.           exp = k3 + j0 + i0 - modj1 - 78;
31.        }// end k
32.     }// end j
33.}// end i
```

Figure 5. The MoD code

MOD operations. We also change the names of the variables to make the conversion to the PSSA code in the last step of the Expression Compiler easier.

As we mentioned in the introduction of this paper, we are only interested to generate the hardware data-path for each expression of a polytope $\mathcal{P}$. The scanning of the polytope (i.e. the implementation of the for-loops) is not the issue of this paper. Thus, we separate the nested-loops control from the control associated with the compilation of an expression. To do so, we transform the MoD code, as given in Figure 5, in a perfect nested-loop code. This is done by adding extra conditional statements for the code that is not in the most inner-loop. Next, the *body* of this *perfect nested-loop* is compiled to a PSSA code. After rewriting the MoD code to its PSSA form and generating VHDL, we obtained the hardware data-path for the input expression. In our current implementation of the PSSA compiler, we do not yet take full advantage of the fact that we can break the critical path using pipeline techniques. Also, we do not yet employ multiplexer reduction techniques to reduce the number of multiplexors used. Figure 6 depicts the data-path architecture that evaluates our example. In the *loop stage*, we implement the nested for-loops. It signalize to the next stage when low-bounds conditions are true. Each term of the expression is evaluated in parallel in the *expression terms*

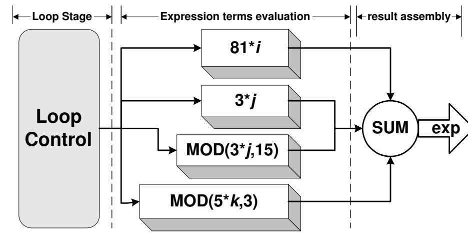

Figure 6. Expression evaluator architecture

evaluation stage. The results are summed in the *result assembly* stage.

The FPGA mapping of the data-path for the expression depicted in Figure 4 takes 68 *Slices* and runs at $250\ Mhz$ (minimum clock period: $3.997ns$). In our evaluation we did not include the area taken by the *Loop Control* implementation. We should be able to integrate IP cores that run at $200\ Mhz$ on a Virtex-II. However, signals routing in an FPGA negatively affects this number. In practice, we have found that LAURA has no problem embedding arbitrary IP cores on a Virtex-II at $100\ Mhz$.

9 Future work and Conclusions

Expressions that contain linear and pseudo-linear terms can be converted efficiently to hardware using Expression Compiler, as presented in this paper. The Expression Compiler is needed in LAURA to make sure that the evaluation of parameterized polytopes in the Read and Write Unit of a hardware process happens faster than the evaluation of an IP Core embedded in the Execute Unit. Only then the data flow in a KPN network is not obstructed by control needed to distribute the original application. Because the control programs in the Read and Write units are expressed in terms of parameterized polytopes, the data flow should even not be obstructed while supporting parameterized control in hardware.

The Expression Compiler first performs high-level optimizations based on number theory axioms and Method of Difference technique. This step is followed by platform dependent optimizations using the Predicated Single Assignment Statement (PSSA). The PSSA form uses only additions, LUTs and conditional statements, and makes possible an efficient mapping to FPGAs platforms. Furthermore, the research community has shown that the PSSA form is well fitted to be mapped in reconfigurable hardware [17]. Expression Compiler is a part of LAURA, helping in improving the quality of the synthesized network of processors. The Expression Compiler uses techniques that are so efficient that we can *always* run the distributed and parameterized

control faster then any IP core we may try to integrate in an Execute Unit. Therefore, the process networks derived by LAURA run efficiently in hardware for stream based applications. These applications typically require highly tuned IP cores and LAURA is able to take full advantages of these IP cores in a distributed manner. Currently, we have not added all the low-level optimization in our tool, but the PSSA form can be further optimized for FPGA platforms using the synthesis techniques as described, for example, in [11].

References

[1] E. Artzy, J. A. Hinds, and H. J. Saal. A Fast Division Technique for Constant Divisors. *Commun. ACM*, 19(2):98–101, 1976.

[2] C. Babbage. *Passages from the life of a Philosopher*. London, 1964.

[3] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *IEEE PACT*, pages 245–255, 1999.

[4] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous concurrent modeling and design in java. Technical Report Memorandum UCB/ERL M01/12, University of California, Dept EECS, Berkeley, CA USA 94720, Mar. 2001.

[5] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.

[6] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis, and E. Deprettere. Deriving efficient control in process networks with compaan/laura. *International Journal of Embedded Systems*, 2005. inderscience publishers.

[7] T. Harriss, R. Walke, B. Kienhuis, and E. F. Depettere. Compilation from matlab to process networks realized in fpga. In *Proceedings of the 35th Asilomar conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, November 4 – 7 2001.

[8] B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.

[9] S.-Y. R. Li. Fast Constan Division Routines. *IEEE Trans. Computers*, 34(9):866–869, 1985.

[10] D. J. Magenheimer, L. Peters, K. Pettis, and D. Zuras. Integer multiplication and division on the hp precision architecture. *IEEE Trans. Computers*, 37(8):980–990, 1988.

[11] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[12] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[13] H. Nikolov, T. Stefanov, and D. Ed. Modeling and fpga implementation of applications using parameterized process networks with non-static parameters. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[14] J. Park and M. Schlansker. On Predicated Execution. In *Technical Report HPL-91-58*. HP Labs, 1991.

[15] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. In *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 113, Washington, DC, USA, 2000. IEEE Computer Society.

[16] J. W. Sheldon, W. Lee, B. Greenwald, and S. Amarasinghe. Strength reduction of integer divison and modulo operations. In *Languages and Compilers for Parallel Computing*, Cumberland Falls, Kentucky, Aug. 2001.

[17] G. Snider, B. Shackleford, and R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 115–124. ACM Press, 2001.

[18] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004.

[19] A. Stoutchinin and F. de Ferriere. Efficient static single assignment form for predication. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 172–181. IEEE Computer Society, 2001.

[20] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 1-3 2003.