Data Reuse Buffer Synthesis Using the Polyhedral Model

Wim Meeus and Dirk Stroobandt, *Member, IEEE*

***Abstract*— Current high-level synthesis (HLS) tools for the automatic design of computing hardware perform excellently for the synthesis of computation kernels, but they often do not optimize memory bandwidth. As accessing memory is a bottleneck in many algorithms, the performance of the generated circuit could benefit substantially from memory access optimization. In this paper, we present a method and a tool to automate the optimization of memory accesses to array data in HLS by introducing local memory tailored perfectly to store only the data that are used repeatedly. Our method detects data reuse in the source code of the algorithm to be implemented in hardware, selects and parameterizes data reuse buffers, and generates a register transfer level design of the data buffers and a matching loop controller that coordinates reuse buffers and datapath operations. Throughout this paper, the polyhedral representation is used extensively as it proves to be well suited for calculations on loop nests and data accesses. As a consequence, this paper is limited to affine programs which can be represented in this model. Experiments show that our method outperforms state-of-the-art academic and commercial HLS tools.**

***Index Terms*— Design methodology, high-level synthesis (HLS), memory architecture, polyhedral model, stencil computing.**

I. Introduction

HIGH-LEVEL synthesis (HLS) is a recent step in the design flow of a digital electronic circuit, in which a design is entered at the algorithmic abstraction level instead of register transfer level (RTL). The HLS tool translates the algorithm into an RTL design, after which the usual RTL design flow follows with synthesis, place, and route. Several benefits arise from starting the design flow at a higher abstraction level using the HLS, including improved designer productivity, faster design verification, and the potential of extensive design space exploration and improved circuit performance with respect to a hand-crafted RTL design. Many HLS tools are available and these tools are being introduced in application-specific integrated circuit and field-programmable gate array (FPGA) design flows. A more extensive discussion of HLS can be found in [1] and [2].

In [3], we have shown that some of these tools generate excellent RTL designs for computation kernels, i.e., accelerators for calculation-dominated dataflow type algorithms. Most of these HLS tools, however, do not optimize memory accesses across loop iterations. This is unfortunate, because the communication between processing elements (PEs) and memory is a well-known bottleneck, which limits circuit performance. Implementing an algorithm in hardware offers massive parallelism in the PE, but this only improves circuit performance if the memory system can keep up. Parallelism and memory access speed do not scale well. This makes memory operations expensive in terms of circuit performance.

Instruction set processors (ISPs) have caches and scratchpad memories to alleviate the memory bottleneck. These generic memory hierarchies perform well for a broad range of applications, and compilers optimize the application for the available memory hierarchy. Automatic memory access optimization, which exists in ISP compilers, was introduced only recently into HLS tools. In the case of HLS, the application is known upfront, so in addition to optimizing the application (e.g., for data locality), the memory architecture can be tailored to the application for improved performance.

This paper improves the throughput and latency of circuits generated by HLS by removing unnecessary data transfers between the circuit and external data memories. To this end, we propose automated methods to generate efficient local storage buffers for data that are used multiple times. We first define classes of data access patterns and then introduce matching local storage buffers to exploit data reuse. We envision an architecture as in Fig. 1, consisting of a datapath for calculations, a memory hierarchy consisting of one or more data reuse buffers, and a loop controller that coordinates buffer and datapath operations. The main goal of this paper is to automate the design flow with the requirement that the generated design should perform as well as handcrafted or hand-optimized designs. Compared with earlier work, our methodology can efficiently handle a broader range of algorithms.

The contributions of this paper are as follows:

1) an automated method, based on the polyhedral model, to analyze data reuse in a loop nest and to select an appropriate reuse buffer template;
2) an automated method to generate instances of data reuse buffers from the selected templates that exploit data reuse between array accesses;
3) an automated method to generate loop controllers that coordinate memory reads and writes and to reuse buffer operations and loop body execution;
4) experiments showing that our presented method performs well without the need for manual optimization.

Manuscript received March 14, 2017; revised October 31, 2017 and February 5, 2018; accepted March 11, 2018. Date of publication April 9, 2018; date of current version June 26, 2018. This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA Project under Grant 671653. *(Corresponding author: Wim Meeus.)*

The authors are with the Department of Electronics and Information Systems, Ghent University, 9000 Ghent, Belgium (e-mail: wim.meeus@ugent.be).

Digital Object Identifier 10.1109/TVLSI.2018.2817159

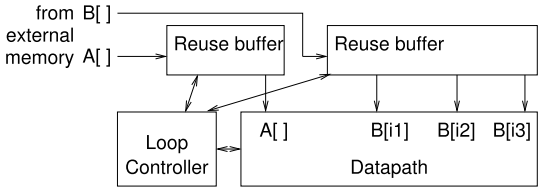

Fig. 1. Proposed architecture.

II. Data Reuse: An Example

To sketch the problem and to introduce our approach, we discuss a hardware design of a Sobel edge detector. The Sobel filter is an image processing algorithm shown in Fig. 2. For the calculation of each $Q[r, c]$, eight input pixels $P[][]$ are needed. A naive hardware implementation would fetch eight pixels to produce each output $Q[][]$. The same pixels are fetched up to eight times, which is a waste of memory bandwidth. Surprisingly, this is what some high-end HLS tools generate.

A more efficient design would fetch each pixel only once from external memory and store its value locally for reuse in later calculations. A data reuse buffer as shown in Fig. 3, consisting of a tapped shift register, can store the pixels for reuse. Each *tap* of the shift register corresponds to an array reference in the source code. Data are shifted into the reuse buffer in the order in which they are used. With this design, no memory bandwidth gets wasted. However, implementing the reuse buffer in RTL and hooking it up to a (potentially generated) datapath is not trivial. For that reason, we want to *automate* the design of such data reuse buffers as well as matching loop controllers.

III. Related Work

As shown in [4], HLS tools (even high-end commercial ones) may not optimize memory accesses, and hand-crafted optimization can be difficult with them. An exception is riverside optimizing configurable computing compiler (ROCCC) [5] which introduces *smart* data reuse buffers [6]. ROCCC is built on the SUIF2 and MachineSUIF platforms, to which it adds memory access analysis and optimization passes as well as a VHDL code generator. ROCCC has some major drawbacks: it only works for sliding window algorithms, it has severe limitations on the input C code, and the generated design does not scale well with the reuse distance [3].

Some authors introduce optimized memory architectures at the algorithmic level, as an optimization of the source code. The generation of the memory hierarchy is left to the HLS tool. In [7], methods are presented to optimize both data transfers from/to the main memory and local storage requirements. Loop nests get optimized for the available memory resources and fitting cyclic reuse buffers are generated using HLS. In Section IX-E, we will demonstrate that this can limit the final circuit performance significantly.

More recently, polyhedral techniques have been used in HLS to optimize the source algorithm for data locality, after which a further optimization step was applied to make the algorithm more suitable for implementation in custom hardware [8]. Data reuse buffers, either FIFOs or scratchpadlike memories, are used between data PEs. These buffers could

```
for (r : 1..rows-2)
  for (c : 1..cols-2)
    gradX = P[r-1,c-1] + 2*P[r-1,c] + P[r-1,c+1]
            - P[r+1,c-1] - 2*P[r+1,c] - P[r+1,c+1]
    gradY = P[r-1,c-1] + 2*P[r,c-1] + P[r+1,c-1]
            - P[r-1,c+1] - 2*P[r,c+1] - P[r+1,c+1]
    grad = abs(gradX) + abs(gradY)
    if (grad>255) grad = 255
    Q[r,c] = 255 - grad
```

Fig. 2. Sobel filter.

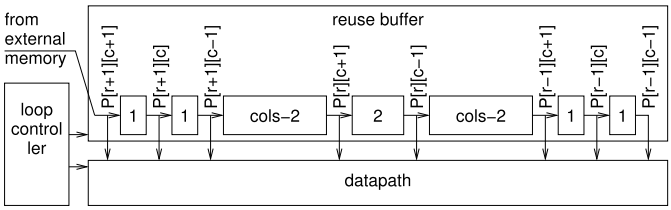

Fig. 3. Rectangular Sobel edge detector design. Inside the reuse buffer, rectangles are shift registers with depth as indicated

be adapted for use between a memory and a PE instead of between PEs. Details on how their buffers are dimensioned and how they perform are missing.

A thorough analysis of data accesses in algorithms is presented in [9]. The dataflow analysis covers all possible cases of data reuse. The data channels that are found can be implemented in hardware directly. However, the resulting hardware can be more complex than what we obtain using our methodology. This paper uses the Sobel edge detector as a running example. The *Sobel* part in Fig. 7 of [9] corresponds with our design of Fig. 3. While our Sobel design has one input channel and seven internal channels, theirs has nine input channels (of which three are shown in the figure) and about 16 internal channels. A design based on the analysis in [9] would thus contain three times as many FIFOs and also additional multiplexing between them.

This paper builds on [4], in which methods are described to find data reuse in a loop nest and to generate the hardware design of a data reuse buffer and loop controller that coordinates reuse buffer and data processing. We extend this paper by adding: 1) reuse buffer templates; 2) support for a single access function; and 3) support for variable reuse distances and writes in the sliding window case. Similar work is presented in [10]. Their microarchitecture also contains a chain of tapped FIFOs to store data for reuse, but they have *filters* at each tap to filter the required data. This filtering happens at runtime. Data are processed when all filters have the required data. In contrast, we determine at design time when data will be available at the FIFO's taps, so we do not need the filters. Instead, we make use of a loop controller which coordinates data fetches, FIFO control, and data processing. Cong *et al.* [10] prove that their (and our) microarchitecture has a minimal memory footprint. As we will show, their design has the same throughput as ours but is considerably larger in area. In [11], another approach for sliding window-type data accesses is presented. Algorithms are mapped onto a pipeline of *stencil time-steps*, PEs equipped with local data buffering. The construction of data buffers is hardly discussed, and only one data access pattern is considered.

In some papers, data reuse is considered at a more coarse level. Loop nests are optimized using *loop tiling*, and data reuse is exploited between successive tiles. Reference [12] discusses the optimization and synthesis of memory accesses using local data buffers. Their transformations are entirely at the source level. As mentioned, we will show that this approach leads to suboptimal results. The methods presented in [13] are based on loop tiling as well. Local data buffers are generated so that that they can store all required data for processing a tile. At the start, loop transformations are applied to improve data locality and expose parallelism. The tile size is determined such that the local data buffers fit in the available on-chip (e.g., FPGA) memory, reducing the memory requirements if necessary. Data are kept in local memory while a whole tile is processed, while our method only keeps data in local memory from the first to the last access. As a consequence, the memory requirements of the methods from [13] may be larger than ours. Also in [13], data reuse is limited to consecutive tiles, which is not a limitation in this paper.

Systolic arrays [14] represent a different approach for array-type calculations. They have one PE per result (e.g., N^2 PEs for $N \times N$ matrix multiplication), such that data can be consumed as soon as they are available. While systolic arrays do not need to store source data and intermediate results are stored inside the PEs, they suffer from bad scalability: the number of PEs has to scale with the size of the problem and the data bandwidth requirements are huge (e.g., 2 N data words per cycle in our matrix multiplication example). Our approach makes use of a single PE, which avoids scalability problems. An extension of this paper to a limited number of parallel PEs is future work.

IV. Polyhedral Model

In this paper, the polyhedral model is used to represent the execution of an algorithm in a geometric way. Its computational simplicity makes it particularly suitable for analysis and automatic optimization. For the polyhedral model to be applicable, loop bounds and array indices must be affine functions of loop variables and constants. This is the case for the majority of loops and array indices found in algorithms, e.g., between 83% and 100% of the array indices of the benchmarks studied in [15] were affine. An extensive discussion on the polyhedral model can be found in [16] and [17]. We briefly discuss the aspects that are key to this paper.

Central to the polyhedral model is the concept of statements. Each statement of a computer program is characterized by a quadruple $(\mathcal{D}, \mathcal{L}, \mathcal{R}, \theta)$, denoting its iteration domain ($\mathcal{D}$), the access functions of written [$\mathcal{L}$ (left-hand side)] and read [$\mathcal{R}$ (right-hand side)] data, and its schedule (θ). The polyhedral model does not define what a statement is, giving the user the flexibility to define the boundaries of a statement. For example, statements in the polyhedral model could correspond with C statements, or with operations in a C program, or with a sequence of C statements. We make use of this flexibility to simplify calculations where possible.

Table I shows how the code fragment in Fig. 4 is translated to the polyhedral model. The iteration domain $\mathcal{D}$ of each

```
for (r : 1..4)
  for (c : 1..r)
    Q = A[r+1][c+1] + B[c][5-r]   // data statement 1
    if (r>1) QQ[r+c] += Q         // data statement 2
```

Fig. 4. Pseudocode.

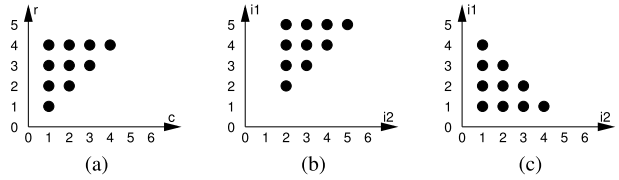

Fig. 5. Access functions project iteration domain of data statement 1 on the data space. Loop variables r and c and array indices i_1 and i_2. (a) Iteration domain. (b) A[r+1][c+1]. (c) B[c][5-r].

TABLE I
Polyhedral Model of Code in Fig. 4

Statement 1		
Iteration domain	$\mathcal{D}_1$	$1 \leq r \leq 4$ and $1 \leq c \leq r$
Schedule	θ_1	(1 r 1 c 1)
Access function of A	$\mathcal{R}_{1,A}$	$(r+1, c+1)$
Access function of B	$\mathcal{R}_{1,B}$	$(c, 5-r)$
Statement 2		
Iteration domain	$\mathcal{D}_2$	$2 \leq r \leq 4$ and $1 \leq c \leq r$
Schedule	θ_2	(1 r 1 c 2)
Access function of QQ	$\mathcal{L}_{2,QQ}$	$(r+c)$

data statement is the set of integer points contained inside a polyhedron defined by the loop bounds. The schedule θ of each statement maps the iterations on a timestamp in a multidimensional time space. Statements are executed according to the lexicographic order of their schedule. The schedule is constructed from the order of the statement in each loop level and the loop variables, e.g., for data statement 2: the r-loop is the first statement of the code fragment (1), r loop (r); the c-loop is the first statement inside the r loop (1), c loop (c); and data statement 2 is the second statement inside the c loop (2). The array references define access functions $\mathcal{L}$ and $\mathcal{R}$. The image of an access function is its data domain, i.e., the set of array indices that are accessed during execution of the loop nest. Fig. 5(a) shows the iteration domain of data statement 1, and Fig. 5(b) and (c) shows the data domains of arrays A and B. The inverse access function determines in which iteration array elements are accessed. The concatenation of an inverse access function and the schedule of a statement are a schedule for data, as it maps data elements to timestamps when the data are accessed. In other words, array data are ordered into a stream.

In this paper, we use Jolylib [19], a Java version of Polylib [20], and iscc from the Barvinok library [18] for polyhedral calculations. In Table II, we highlight some iscc syntax.

V. Detailed Problem Definition

We now present a more detailed definition of our research question. The starting point of this paper is an algorithm, expressed as a perfect loop nest in which array data are accessed. As many general loop nests can be converted into perfect ones [21], [22], the requirement of a perfect loop nest

TABLE II
ESSENTIAL ISCC SYNTAX (SEE [18])

$\{ S[indices] : boundary \}$	definition of domain
$\{ S[indices] \rightarrow A[indices] \}$	definition of mapping
$:=$	assignment
.	concatenation of mappings
$*$	intersection of either 2 domains or a mapping and a domain
$+$	union
$-$	set difference
m^{-1}	inverse mapping
$<<$	a mapping to lexicographically smaller elements
$<<=$	a mapping to lexicographically smaller or equal elements
lexmin	lexicographical minimal element of a set or of the image of a mapping
lexmax	lexicographical maximal element of a set or of the image of a mapping
ub	upper bound
ran	range (image) of a mapping
card	number of elements in set or in image of mapping

is seldom a limitation. The outcome is an RTL design as in Fig. 3. The data reuse buffers and the loop controller are generated by our tool, and the datapath is generated using a commercial HLS tool. The challenges of this paper are as follows:

1) designing a top level architecture that integrates the datapath, local data storage, and the loop controller (Section VI);
2) designing reuse buffer templates that are suited for different types of data access patterns (Section VI);
3) analyzing data reuse and data access patterns in the algorithm to be implemented in hardware (Section VII-A);
4) selecting a suitable reuse buffer design template for the data access patterns in the algorithm (Section VII-B);
5) parameterizing these templates for the algorithm to be implemented (Sections VIII and IX);
6) building a loop controller that coordinates data reuse buffer operations and datapath execution (Sections VIII and IX).

Rather than designing a solution that works for all the cases, using a set of templates for generating data reuse buffers reduces the complexity of the problem. Handling particular data access patterns separately may be less general, but this approach has a few advantages. Templates can be fine-tuned to the specific data access pattern, reducing the latency and hardware overhead. Also, the design effort can be focused at those data access patterns that occur frequently. Designing the templates needs to be done only once. These templates serve as the input for a software tool that automates the subsequent tasks. The more templates that are available, the more access patterns that can be optimized. In this paper, we present two such templates. A first template matches array references that have the same access function, which is found in, among others, matrix multiplication and FIR filters. The second template matches the *sliding window* access pattern with bijective access functions, which is found in stencil computing. In this case, an array is accessed with different access functions that have the same dependence on the loop variables. Developing additional reuse buffer templates, e.g., to support a sliding window access pattern with a nonbijective access function, and access functions that access data in opposite directions (e.g., $A[i]$ and $A[k-i]$), is future work.

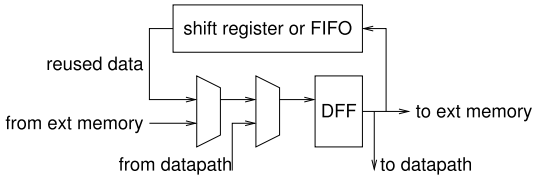

Fig. 6. Buffer architecture

In this paper, we do not apply loop transformations to make the algorithm more suitable for implementation in hardware. We assume that the source algorithm is already optimized for hardware implementation. These optimizations, such as data locality improvement, are discussed extensively in [7], [9], [13], [16], and [17]. Our flow will generate an architecture that is tailored to the algorithm as it is given.

It is paramount to clearly distinguish an *array reference*, a source code construct indicating an access to array data, from an *array access*, which is an actual array read or write operation that is the result of executing the array reference. In a loop nest, each array reference causes a series of array accesses. The *access function*, defined by the index expression(s) of an array reference, expresses the relation between the loop variables and parameters, and the referenced data element.

VI. PROPOSED ARCHITECTURE

The architecture of our solution, shown in Fig. 1, extends the architecture from [23]. It has a datapath that executes the data statements and a loop controller to generate the values of loop variables and control the datapath operations. We propose two changes: 1) we consider the whole loop body as one statement and 2) we add data reuse buffers for array data as additional components. When the loop body is compiled as one statement, the loop controller will be simpler as it only needs to control one data statement, while an HLS tool gets a larger piece of code for the datapath which enables more optimization. With the addition of the data reuse buffers, external memory accesses are handled by the loop controller and the data reuse buffers. In this paper, we make use of Catapult C from Mentor Graphics to compile the loop body to RTL.

For a circuit that has low latency and high throughput, the design of the data reuse buffers needs to match the data access pattern. Our tool uses a number of data reuse buffer design templates from which it selects and parameterizes the appropriate one at design time. The templates can be hand-crafted, because they only need to be designed once. In Sections VI-A and VI-B, we present templates for two different data access patterns.

A. Single Access Function

In this data access pattern, all references to an array have the same access function. This pattern occurs in, for example, matrix multiplication and correlation. In order to produce the same data multiple times, the data buffer needs to contain a loop as in Fig. 6. Data are fetched from external memory for their first use and kept in the local buffer for subsequent uses. After their last use, the data are written back to the

1: $ED := \{\ S[i,j] : 0 \leq j \leq 12 \text{ and } 0 \leq i < 1000\ \};$ ▷ iteration domain (iteration space)
2: $A_1 := \{\ S[i,j] \rightarrow A[i+j]\ \} * ED;$ ▷ access relation
3: $DD := ran(A_1);$ ▷ data domain: *range* of the access relation in the data space
4: $REUSE := card(ED) - card(DD);$ ▷ number of cells reused (0 = no reuse, ≥ 1 = reuse)

Fig. 7. Data reuse detection: single access function (iscc syntax: Table II or [18]).

1: $ED := \{\ S[i,j] : 0 \leq j \leq 12 \text{ and } 0 \leq i < 1000\ \};$ ▷ iteration domain (iteration space)
2: $A_1 := \{\ S[i,j] \rightarrow A[i][j]\ \} * ED;$ ▷ access relation
3: $A_2 := \{\ S[i,j] \rightarrow A[i+1][j]\ \} * ED;$
4: $DD_1 := ran(A_1);$ ▷ data domain
5: $DD_2 := ran(A_2);$
6: $REUSE := card(DD_1 * DD_2);$ ▷ number of cells reused (0 = no reuse, ≥ 1 = reuse)

Fig. 8. Data reuse detection: two access functions (iscc syntax: Table II or [18]). Note the different meanings of $*$ in this code: in line 6, it denotes an intersection between two domains, while in lines 2 and 3, it adds a domain to a mapping.

main memory or discarded from the reuse buffer. This data access pattern and potential reuse buffer simplifications will be discussed in Section VIII.

B. Sliding Window Access Pattern

The sliding window access pattern is found in stencil computing. Array references have different access functions, but these access functions all depend on the loop variables in the same way. In addition, the access functions must be bijective over the loop nest. In this pattern, different array references may access the same data in different loop iterations. All array references will access the shared data in the same order, but the accesses will be offset in time. For example, $A[i]$ and $A[i+1]$ access the same data in the same order, but (for incrementing i) $A[i+1]$ will do that one iteration earlier than $A[i]$. In this case, a suitable architecture consists of a sequence of shift registers and FIFOs, as shown in Fig. 3. The number and depth of FIFOs and shift registers and the presence of read and write ports depend on the source algorithm, and will be discussed in Section IX.

VII. Automated Design Flow

The design flow is shown in Fig. 9. First, the algorithm to be implemented in hardware is searched for data reuse. If data reuse occurs, and if a matching reuse buffer template is found, our tool generates RTL code of a fitting reuse buffer instance from the template. In addition, a loop controller is generated to coordinate all reuse buffer and datapath operations.

A. Detecting and Organizing Data Reuse

The first part of our automated flow deals with finding data reuse and organizing array references that share data. Reuse analysis is done per array, and we assume that no aliasing between arrays occurs. Two types of data reuse can occur. A single array reference can access the same data in different loop iterations, or two array references may access the same data in either the same or different loop iterations. Array references to the same array with the same access function can be considered as one array reference with the iteration domain as the union of their iteration domains. Array references that share data can be grouped into *reuse sets*. For the first type (one array reference), the reuse detection

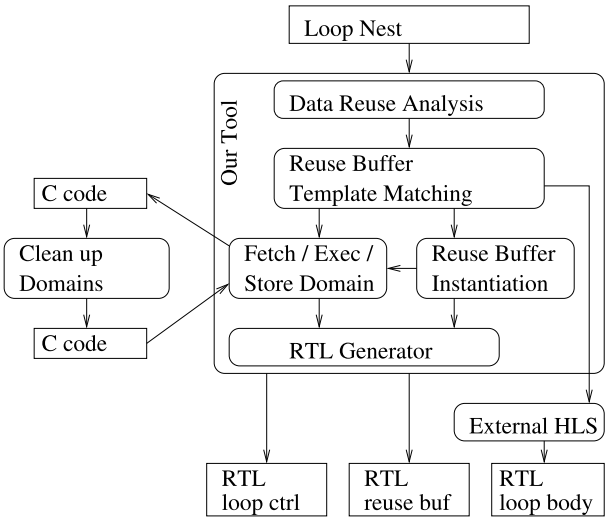

Fig. 9. Polyhedral reuse buffer design flow.

algorithm is given in Fig. 7. The example is taken from the correlation use case, which is discussed in Section VIII-E. ED is the iteration domain, i.e., (i, j) pairs of iterations of the loop nest. A_1 represents the array index of array $A[]$. Data domain DD is the set of elements from $A[]$ that are used in the loop and is found as the projection of ED on the data space. In each iteration, one element of $A[]$ is used. If the number of iterations or the number of pairs in ED [$card(ED)$] equals the number of cells of $A[]$ that are accessed, a different cell of $A[]$ is used in each iteration and no reuse occurs. In that case, $REUSE$ is zero. If the number of accessed cells of $A[]$ is smaller than the number of iterations (and $REUSE > 0$), some elements of $A[]$ are used in multiple iterations, so data reuse occurs. Note that, if the loop bounds or access functions contain parameters whose values are unknown at design time, evaluating and comparing the cardinality of sets may not be possible.

The reuse detection algorithm for the second type is shown in Fig. 8. The intersection of the data domains of two different array references contains data which are used by both array references. If this intersection is not empty, data reuse occurs.

B. Reuse Buffer Template Selection

The next step is to find the best fitting reuse buffer template for each reuse set. The applicability of the reuse buffer

1: $ED := \{\ S[i,j] : 0 \le j \le 12 \text{ and } 0 \le i < 1000\ \}$; ▷ iteration domain (iteration space)
2: $ACC := \{\ S[i,j] \to A[i+j]\ \} * ED$; ▷ access relation
3: $DD := ran(ACC)$; ▷ data domain
4: $LT := ED << ED$; ▷ map to all later iterations
5: $LE := ED <<= ED$; ▷ map to all later and current iterations
6: $T := (ACC.(ACC^{-1})) * LT$; ▷ map to all later iterations that access same data
7: $M := lexmin\ T$; ▷ map to the the next iteration that accesses same data
8: $AFTER\_PREV := (M^{-1}).LE$; ▷ map to all iterations after the previous access to the same data
9: $BEFORE := (LE^{-1})$; ▷ map to all previous and current iterations
10: $RDIST := card((AFTER\_PREV * BEFORE).ACC)$; ▷ reuse distance
11: $ND := ran(lexmin(ACC^{-1} * DD))$; ▷ iterations with fetch from external memory
12: $LD := ED - ND$; ▷ iterations with data reuse from the loop buffer
13: $FRD := ED - ran(lexmax(A^{-1}))$; ▷ iterations using data that will be reused in a later iteration
14: $MAXDEPTH := ub(RDIST) - 1$; ▷ maximum FIFO depth (in case of variable reuse distance)

Fig. 10. Calculating reuse distance (similar to [24]) and fetch/reuse domains for array A of the correlation example (iscc syntax: Table II or [18]).

templates from Section VI can be tested using the polyhedral representation. To match the single access function template, the only thing to check is that the access functions of all array references in the reuse set are the same. To match the sliding window template, one must check that all access functions have the same dependence on loop variables and that all access functions are bijective over their domain.

C. Generating Data Reuse Buffers From the Selected Templates

From each of the selected reuse buffer templates, the RTL design is generated, taking the exact data access pattern into account. This customization is template-specific. In addition to generating the RTL design of the reuse buffer, this step also calculates the timing of the buffer's control signals for later implementation in the loop controller. Customizing the single access function template and the sliding window template is discussed in Sections VIII and IX, respectively.

D. Generating the Loop Controller

From the domains that were calculated during reuse buffer generation, a loop controller is generated as in [23]. One loop controller controls all reuse buffers and the datapath. The loop controller consists of a number of coupled finite state machines which generate the values of loop variables and the control signals for reuse buffers and datapath. Details will be presented in Sections VIII and IX. For the loop controller RTL code to be efficient, the polyhedral domains sometimes need to be optimized for implementation in hardware. Optimizations in, e.g., iscc [18] are instruction processor-oriented and often are not well-suited for a direct hardware implementation. We perform a manual cleanup of polyhedral domains, but leave automation of this step to future work.

VIII. Single Access Function Template

In this section, we discuss the single access function data reuse buffer template. We assume that the algorithm contains one or more array references that fit this template. We discuss the additional polyhedral analysis and a number of use cases, and we present experimental results.

```
for (i : 0..99)
  for (j : 0..99)
    for (k : 0..99)
      if (k==0) Q[i][j]  = A[i][k]*B[k][j];
      else      Q[i][j] += A[i][k]*B[k][j];
```

Fig. 11. Matrix multiplication benchmark.

A. Additional Data Access Pattern Analysis

First, the reuse distance needs to be calculated, as shown in lines 1–10 of Fig. 10. For the polyhedral representation, the whole inner loop body is considered as one statement. This simplifies the polyhedral calculations without loss of accuracy. In lines 1 and 2, the iteration domain of the loop body and the access relation are given. The data domain (line 3) is the projection of the iteration domain onto the data space using the access function.

The reuse distance calculation, similar to the method in [24], is presented in lines 4–10. It counts the number of array cells accessed between two accesses to the same cell. In lines 4 and 5, LT maps each iteration to all successive iterations and LE maps each iteration to all successive iterations and to itself. In line 6, $ACC.ACC^{-1}$ maps iterations to all iterations that access the same data, and T restricts this to all later iterations that access the same data. In line 7, M maps iterations to the next iteration that accesses the same data. In line 8, M^{-1} maps iterations to the iteration in which the same data were accessed previously, and $AFTER\_PREV$ maps iterations to all iterations after the previous iteration in which the same array cell was accessed. $BEFORE$ in line 9 is the inverse of LE. Finally, in line 10, $AFTER\_PREV * BEFORE$ maps iterations to the set of iterations between two successive accesses to the same array cell, $(AFTER\_PREV * BEFORE) * ACC$ maps to the set of array cells accessed in these iterations, and the number ($card$) of cells in this latter set is the reuse distance.

The calculated reuse distance may be variable or constant during loop execution. In the latter case, a design as in Section VIII-B can be applied, while in the variable reuse distance case, the more complex design of Section VIII-C is necessary. Read-only reuse buffers are considered first. In Section VIII-D, extensions to support a mix of reads and writes, regardless of constant or variable reuse distance, are

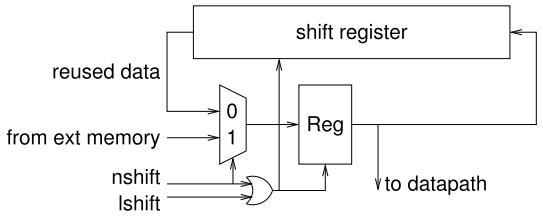

Fig. 12. Fixed reuse distance buffer design.

```
for (i : 0..99)
  for (j : 0..99)
    for (k : i..99)
      if (k==i) Q[i][j]  = A[i][k]*B[k][j];
      else      Q[i][j] += A[i][k]*B[k][j];
```

Fig. 13. Triangular matrix multiplication benchmark.

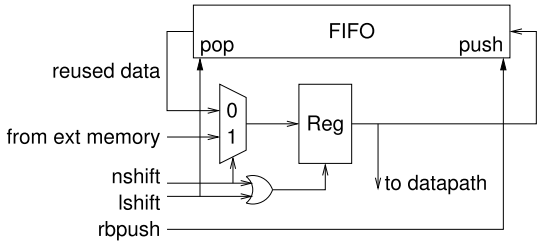

Fig. 14. Reuse buffer design with variable reuse distance.

presented.

B. Reuse Buffer With Constant Reuse Distance

In matrix multiplication (Fig. 11), the reuse distance is constant for all arrays (100 for A, 10 000 for B, and 1 for Q in the example). The design of the reuse buffer is shown in Fig. 12. On their first use, data are read from an external memory and kept in the *active data* register for use in the datapath. After use, all data are shifted into the shift register that loops back to the input multiplexer. The depth of the shift register equals the reuse distance minus one. Two signals control the operation of the reuse buffer: $nshift$ is asserted when new data from the external memory are shifted into the buffer, and $lshift$ is asserted when data from the buffer are reused. The activity of these control signals is calculated in lines 11 and 12 of Fig. 10. $nshift$ (domain ND) is active when data are accessed for the first time, and $lshift$ (domain LD) is active in all remaining iterations. If the reuse distance is equal to one, the length of the shift register becomes zero, so the shift register, mux, OR gate, and lshift input can all be omitted and the reuse buffer can be simplified to a single register.

C. Reuse Buffer With Variable Reuse Distance

When the reuse distance is not constant (e.g., triangular matrix multiplication as in Fig. 13), a design as in Fig. 14 is required that uses an FIFO to store data between subsequent uses. Three control signals are required. $nshift$ and $lshift$ are the same as in the constant reuse distance case, with $lshift$ popping data from the reuse FIFO. In addition, $rbpush$ controls which data are pushed into the FIFO, namely data that

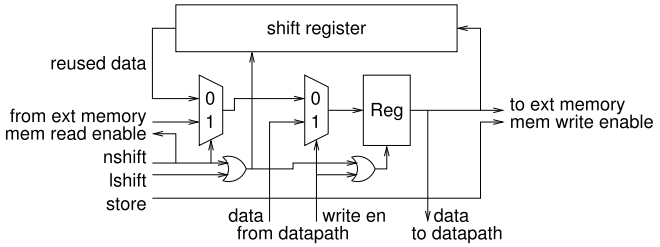

Fig. 15. Reuse buffer design with read/write.

```
for (i : 0..99)
  for (j : 0..99)

    Q[i] = foo(Q[i],
           A[i][j]...);
```

(a)

```
for (i : 0..99)
  for (j : 0..99)
    if (j==0) Q[i]=0;
    Q[i] = foo(Q[i],
           A[i][j]...);
```

(b)

Fig. 16. Write before or after read. (a) Read first: Q must be initialized. (b) Write first: no initialization of Q required.

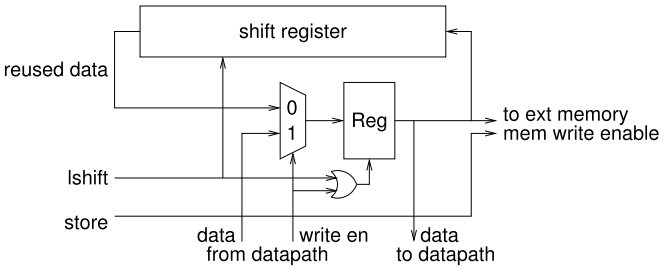

Fig. 17. Reuse buffer design with read/write, when no data need to be read from external memory

will be reused in a later iteration. The calculation of $rbpush$'s iteration domain is shown in line 13 of Fig. 10. The maximum depth of the FIFO is one less than the upper bound of the reuse distance, as shown in l.14 of Fig. 10.

D. Read/Write Reuse Buffer

The solutions that we have presented so far apply to array data that are only read during the execution of the algorithm. Read and write access can be supported with a design as in Fig. 15 (assuming a constant reuse distance). The interface to the datapath gets both read and write ports and a write enable signal. An additional signal is needed to control the operation to store written data in external memory after the last write. In the case of a variable reuse distance, the same modifications as in Section VIII-C can be applied.

For optimal performance, unnecessary reads and writes from and to external memory must be avoided. If the initial (and potentially uninitiated) values of array data are not used, as is the case with Q, these values should not be fetched from external memory. Whether initial values are used depends on the first operation: if it is a read, as in Fig. 16(a), the initial value is used, while in the case of a write [Fig. 16(b)], the initial value is not used. If all data of an array are written first, the memory read port can be omitted and the design gets simplified as in Fig. 17.

The calculation of the control signals' activity needs to be modified with respect to the read-only design. Considering

1: $DW := \{q0[i,j,k] : 0 \le i < 100 \text{ and } 0 \le j < 100 \text{ and } 1 \le k < 100;$ ▷ iteration domain of writes
$q2[i,j,k] : 0 \le i < 100 \text{ and } 0 \le j < 100 \text{ and } k = 0\};$
2: $DR := \{q1[i,j,k] : 0 \le i < 100 \text{ and } 0 \le j < 100 \text{ and } 1 \le k < 100\};$ ▷ iteration domain of reads
3: $AW := \{q0[i,j,k] \to A[i][j]; q2[i,j,k] \to A[i][j]\} * DW;$ ▷ write access relations
4: $AR := \{q1[i,j,k] \to A[i][j]\} * DR;$ ▷ read access relations
5: $S := \{q0[i,j,k] \to [0,i,0,j,0,k,1];$ ▷ schedules
$q1[i,j,k] \to [0,i,0,j,0,k,1]; q2[i,j,k] \to [0,i,0,j,0,k,0]\};$
6: $WDATA := ran(AW);$ ▷ written data domain
7: $RDATA := ran(AR);$ ▷ read data domain
8: $DD := ran(AW) + ran(AR);$ ▷ data domain
9: $FIRSTREAD := (lexmin(AR^{-1}.S)).(S^{-1});$ ▷ map data to iteration of first read
10: $BEFOREW :=$ any AW before $FIRSTREAD^{-1}$ under $S;$ ▷ writes before the first read
11: $WBDATA := ran(AW * dom(BEFOREW));$ ▷ data that do not need to be fetched
12: $FETCH := ran(lexmin(AR^{-1} * (RDATA - WBDATA).S).S^{-1}) * DR;$ ▷ iterations in which data must be fetched
13: $LSHIFT := (DR + DW) - FETCH;$ ▷ iterations in which data are reused
14: $STORE := ran(lexmax(AW^{-1}.S).S^{-1}) * DW;$ ▷ iterations in which data are written for the last time

Fig. 18. Calculating domains for array Q (matrix multiplication example) (iscc syntax [18]).

```
for (i : 0...)
  for (j : 0..12)
    if (j==0) corr = 0;
    corr = corr + ( A[i+j] * B[j] )
    if (j==12)
      corr = abs(corr);
      if (corr > maxcorr)
        maxindex = i
        maxcorr  = corr
```

Fig. 19. Correlation pseudocode.

the loop body as a single statement in the polyhedral representation would hide whether the first operation on data is a read or a write, so individual statements have to be considered. In Fig. 18, the calculations are shown for array Q[] of the matrix multiplication case, which we will discuss in Section VIII-E. Lines 1–5 of Fig. 18 contain the information from the loop nest: iteration domains, access functions, and schedules of the array accesses to Q[]. Lines 6–8 calculate the data domains of written, read, and accessed (read or write) data. In line 9, the iterations are found in which an element of Q[] is accessed for the first time. In line 10, we determine whether there was a write access before the first read access. In line 11, we determine which data are written before they are read. These data do not need to be initialized or read from external memory before use. In line 12, the iterations are calculated in which data that do need to be fetched from external memory are accessed for the first time. The iterations in which data are reused from the buffer are determined in line 13. Finally, in line 14, we find the iterations in which elements of Q[] are written for the last time, after which they can be stored in external memory.

E. Experimental Results

We have studied three common algorithms that each access multiple arrays using a single access function, representing a number of different use cases, which can all be optimized using our method.

1) Correlation (see Fig. 19): The correlation between a signal A[] and a short pulse B[] to find the index of A[] where the correlation is maximal. Both arrays are 1-D and have constant reuse distances. After one iteration of the outer loop, array A reuses 12 data elements and gets 1 new element per iteration of the outer loop, while array B only reuses data.
2) Matrix Multiplication: $Q[] = A[] \cdot B[]$ (see Fig. 11). The arrays are 2-D. Arrays A and B are read-only and have constant reuse distances. Array Q is read–write and its access pattern has a constant reuse distance of one.
3) Matrix Multiplication With an Upper Triangular Matrix: $Q[] = A[] \cdot B[]$ with $A[i][j] = 0 \; \forall \; j < i$, which reduces the number of multiplications as shown in Fig. 13. Only a little more than half of array A is read. Its access pattern has a variable reuse distance.

TABLE III
EXPERIMENTAL RESULTS: SINGLE ACCESS FUNCTION. FPGA RESOURCES: SLICES/DFFS/BLOCKRAMS/DSPS

	II	Latency (cycles)	FPGA resources	memory accesses	mem acc / result
Rectangular matrix multiplication					
HLS, unoptimized code	2	2,000,004	35/104/0/1	3,990,000	399
HLS, optimized code 1	1	1,000,003	36/46/0/1	2,010,000	201
HLS, optimized code 2	1	1,000,006	46/189/3/2	30,000	3
Our tool, unopt. code	**1**	**1,000,002**	**64/99/3/1**	**30,000**	**3**
Triangular matrix multiplication					
HLS, unoptimized code	2	1,030,004	29/124/0/3	2,010,000	201
HLS, optimized code 1	1	515,003	27/108/0/4	1,020,000	102
HLS, optimized code 2	1	515,004	39/201/3/4	25,050	2.5
Our tool, unopt. code	**1**	**505,002**	**77/219/3/1**	**25,050**	**2.5**
Correlation					
HLS, unoptimized code	1	13,003	34/99/0/1	26,000	26
HLS, optimized code, FF	1	13,003	94/335/0/1	1,013	1.01
HLS, optimized code, ram	1	17,000	46/112/4/1	1,013	1.01
Our tool, unopt. code	**1**	**13,002**	**40/125/0/1**	**1,013**	**1.01**

```
for (i : 0..10)
  B[i] = A[i] + A[i+1]
```

Fig. 20. Code fragment.

We have compared designs generated by our tool from unoptimized source code with designs generated using a commercial HLS tool—Catapult C—from either unoptimized or manually optimized code. Table III summarizes the results. Starting from unoptimized source code, the designs generated by commercial HLS tools access the memory considerably more than a design generated by our tool, which can be seen from the right two columns (total number of

1: $ED := [rows, cols] \rightarrow \{\ S[i,j] : j \geq 1$ and $j \leq cols - 2$ and $i \geq 1$ and $i \leq rows - 2\ \}$; ▷ iteration domain (iteration space)
2: $A_1 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, 1+j]\ \} * D$; ▷ access functions
3: $A_2 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, j]\ \} * D$;
4: $M := \{\ S[i,j] \rightarrow [i,j]\ \}$; ▷ schedule
5: $M_0 := lexmin(A_0^{-1}.M)$; ▷ map data to time of first use
6: $M_1 := lexmin(A_1^{-1}.M)$;
7: $TIME := ran\, M$; $LT := TIME << TIME$; ▷ check order of accesses
8: $MM := M_0^{-1}.M_1$;
9: $R := card(MM * LT)$; ▷ $R > 0 : A_0$ before A_1; $R = 0 : A_0$ after A_1

Fig. 21. Ordering array references $A[i+1][j+1]$ and $A[i+1][j]$ (iscc syntax [18]).

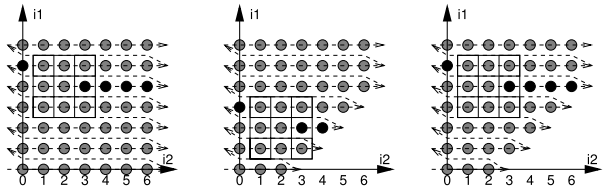

Fig. 22. Reuse distance between $P[r+1][c-1]$ and $P[r][c+1]$. (a) Rectangular and (b) Nonrectangular data domain. Dotted arrows indicate access order of P[i1][i2]. Gray and black dots indicate which elements of $A[]$ are read during algorithm execution, and black dots are the ones between $A[i+1][j-1]$ and $A[i][j+1]$ at some point. The number of black dots equals the reuse distance between $A[i+1][j-1]$ and $A[i][j+1]$.

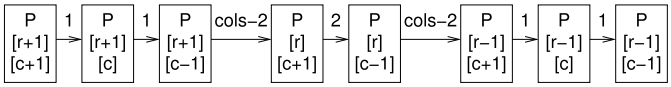

Fig. 23. Reuse chain: array references in the boxes and reuse distances above the arrows.

```
for (I ∈ iteration_domain)
  FETCH_DATA_INTO_REUSE_BUFFER(I);
  EXECUTE_LOOP_BODY_WITH_BUFFERED_DATA();
```

Fig. 24. Incomplete loop control.

```
for (I ∈ extended_iteration_domain)
  if (I ∈ fetch_domain)
    FETCH_DATA_INTO_REUSE_BUFFER(I);
  if (I ∈ execute_domain)
    EXECUTE_LOOP_BODY_WITH_BUFFERED_DATA();
```

Fig. 25. Correct loop control.

memory accesses and accesses per calculation). Designs with the commercial HLS tool improve if the source code is optimized, but for optimal performance, these optimizations require far from trivial source code changes. Matrix multiplication was optimized by keeping $Q[i][j]$ in a local register during the k loop (optimization 1) and by explicitly adding local storage (optimization 2). For correlation, HLS constraints were used to force Catapult C to map local storage either as block RAM or in the FPGA fabric. An optimal solution could not be achieved using blockram. In contrast, our toolflow produces equally good designs from source code that was not optimized by hand. The resulting designs meet the same clock speed target (200 MHz) and use comparable numbers of FPGA resources.

IX. Sliding Window Template

The sliding window data reuse buffer template applies if reuse sets were found with access functions that have the same dependence on loop variables. We discuss the polyhedral analysis for this template and experimental results from a number of use cases, such as the Sobel edge detector.

A. Additional Data Access Pattern Analysis

Array references with the same dependence on loop variables can be ordered according to the time when they access the shared data. For example, in the loop of Fig. 20, $A[i]$ accesses each element of $A[\,]$ one iteration earlier than $A[i-1]$. We call the ordered reuse set a *reuse chain*. Using the polyhedral representation, ordering pairs of array references from the Sobel algorithm (Fig. 2) happen as in Fig. 21. Line 1 defines the data domain, lines 2 and 3 define the access relations, and line 4 defines the schedule. In lines 5 and 6, the data domain is projected onto the schedule domain using the inverse of each of the access functions concatenated with the schedule function. In line 7, LT is a mapping in the schedule space of a point in time to all later points in time. In line 8, a mapping MM is defined in the schedule space of the time when A_0 accesses data to the time when A_1 accesses the same data. In line 9, we check whether the A_1 access was later than the A_0 access (intersection not empty), which defines the order of these accesses in the reuse chain. The relative ordering of all pairs of access functions enables our tool to order all of them as a reuse chain. The first array reference in the reuse chain is called the *head* of the reuse chain. We also calculate the spatial reuse distance of each pair of successive array references in the chain. The polyhedral calculation of the reuse distance is graphically represented in Fig. 22 and is similar to the one in Section VIII-A.

In the Sobel filter example, each pair of references to $P[\,][\,]$ has overlapping data domains, so they form a single reuse set. Ordering the array references gives a reuse chain as in Fig. 23; P[r+1][c+1] is the head of the reuse chain. The reuse distance between each pair of successive array references is given above the arrows.

If the reuse distance between each pair of successive array references is constant during loop execution as in Fig. 22(a), the method described in Section IX-B applies. Otherwise, the more complex approach of Section IX-C is required.

B. Read-Only Reuse Buffer With Constant Reuse Distances

In this section, we create the hardware design of the data reuse buffer and a matching loop controller. These are building

1: $ED := [rows, cols] \rightarrow \{\ S[i,j] : j >= 1$ and $j <= cols - 2$ and $i >= 1$ and $i <= rows - 2\ \}$; ▷ iteration domain (iteration space)
2: $A_1 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, 1+j]\ \}$; ▷ access functions
3: $A_2 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, j]\ \}$;
4: $A_3 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, -1+j]\ \}$;
5: $A_4 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[i, 1+j]\ \}$;
6: $A_5 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[i, -1+j]\ \}$;
7: $A_6 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[-1+i, 1+j]\ \}$;
8: $A_7 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[-1+i, j]\ \}$;
9: $A_8 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[-1+i, -1+j]\ \}$;
10: $DD := ran((A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8) * ED)$; ▷ data domain of reuse set (data space)
11: $FD := ran(A_1^{-1} * DD)$; ▷ data fetch domain (iteration space)

Fig. 26. Calculating data and fetch domains (iscc syntax [18]). A_1 is the head of the reuse chain.

blocks that fit into the design in Fig. 1. The reuse chain can be mapped directly onto a chain of shift registers as in Fig. 3. For each pair of successive array references in the reuse chain, a shift register is generated. The array references in the reuse chain map to the taps before, between, and after the shift registers, where data are tapped from the reuse buffer to feed the datapath. The head of the chain maps onto the leftmost tap where data are fed in from the main memory. The length of each shift register equals the reuse distance as defined earlier. Starting from the calculated reuse chain and reuse distances, our tool generates a synthesizable RTL design of the reuse buffers. A configurable reuse distance threshold is used to choose between a register or RAM-based shift registers.

The design of the loop controller is next, which coordinates the reuse buffer and datapath operations. The pseudocode in Fig. 24 describes how the loop should execute. In each iteration, one new data element per reuse chain is fetched from the main memory, after which the loop body executes. While the code in Fig. 24 is valid for many loop iterations, a number of data fetches need to be added for correct execution. In order to include all necessary data fetches and coordinate them with loop body execution, the iteration domain is extended as in Fig. 25. At the start of the loop nest, a number of iterations are added to load the first two lines of the image into the buffer (when $I \in$ fetch_domain but $I \notin$ execute_domain). At the start of the inner loop, two more pixels at the start of each line are loaded. These additional fetches prefill the reuse buffer. Once the reuse buffer is filled, the loop body can be executed to produce a first result. With each subsequent pixel read, a new pixel can be produced by the datapath ($I \in$ fetch_domain and $I \in$ execute_domain).

The extended iteration domain is the union of the *execute* and *fetch* domains. The execute domain is the same as the domain of the original loop nest. The presence of the loop buffer does not change the number of calculations or the associated iterator values. The fetch domain is the iteration domain of the data fetch operation. We want only the required data to be fetched from the main memory, data to be fetched only once, and fed in the right order to the reuse buffer, and before they are needed by the datapath. Using the access function of the head of the reuse chain to project the data domain of the reuse set onto the iteration domain, the desired fetch domain is found. For the Sobel filter example with rows = cols = 4, the polyhedral calculations are given in Fig. 26. First, the iteration domain of the original loop nest (or execution

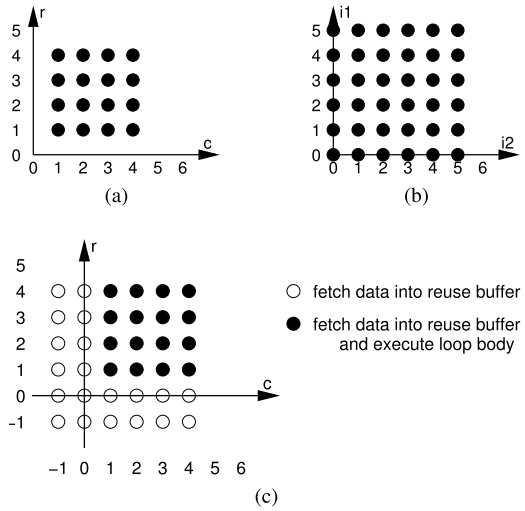

Fig. 27. Rectangular case: iteration and data domain. (a) Iteration domain. (b) Data domain. (c) Extended iteration domain.

```
for (r : 1..rows-1)
  for (c : 1..r) // bound of c depends on value of r now!
    gradX = P[c-1,r-1] + 2*P[c-1,r] + P[c-1,r+1]
            - P[c+1,r-1] - 2*P[c+1,r] - P[c+1,r+1]
    gradY = P[c-1,r-1] + 2*P[c,r-1] + P[c+1,r-1]
            - P[c-1,r+1] - 2*P[c,r+1] - P[c+1,r+1]
    // ...
```

Fig. 28. Triangular Sobel filter.

domain, line 1) and the access functions (lines 2–9) are given. The data domain is the projection of the execution domain using the union of access functions (line 10). The fetch domain is the projection of the data domain onto the iteration space using the inverse of the first access function (line 11). Fig. 27 shows a graphical representation of the domains. Fig. 27(a) represents the iteration domain and Fig. 27(b) represents the corresponding data domain. The head of the reuse chain is P[r+1][c+1]. Inverting the access function gives $r = i_1 - 1$ and $c = i_2 - 1$. Applying the inverse access function to the data domain gives the fetch domain in Fig. 27(c). The white dots belong to the fetch domain only, and the black dots belong to both the fetch and execute domains.

The RTL design of the loop controller is based on [23]. The controller calculates the values of the loop variables and starts reuse buffer and datapath operations. The generated designs are evaluated in Section IX-E.

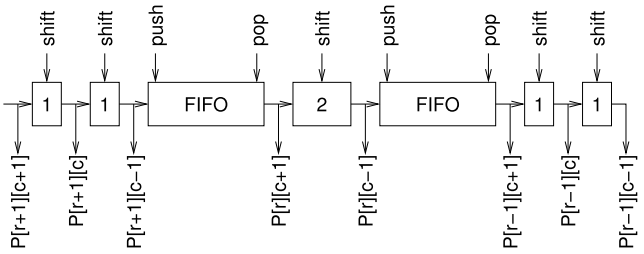

Fig. 29. Reuse buffer for triangular Sobel edge detector. Rectangles with a number are shift registers with depth as indicated, and FIFOs accommodate variable reuse distances.

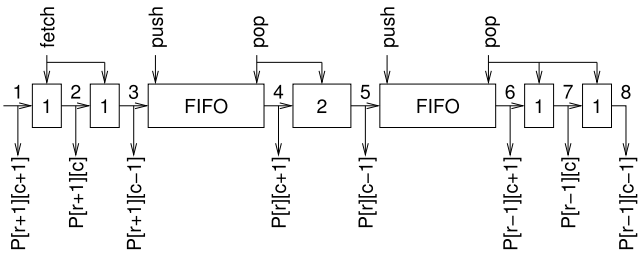

Fig. 30. Reuse buffer for triangular Sobel edge detector. A number of control signals are shared.

C. Variable Reuse Distance

In some algorithms, the reuse distances are not constant, e.g., when a triangular 2-D data domain is accessed, or when the data window is not rectangular. An example is the *triangular* Sobel filter of Fig. 28. The methods from Section VII-A to find data reuse and to build the reuse chain are still valid. The method to calculate the reuse distance will return either an expression that depends on the iterators or multiple expressions. The architecture of the reuse buffer still resembles the one of Section IX-B, but FIFOs are required as in Fig. 29 to accommodate variable reuse distances.

Controlling a reuse buffer with FIFOs is more complex than without them. The timing of the different `shift`, `push`, and `pop` signals must be calculated, as well as the maximum depth of the FIFOs. `shift` signals of successive shift registers must be activated at the same time, otherwise data either get duplicated or lost. The same applies to the `pop` signal of an upstream FIFO. At the upstream end of the buffer, the shift signal must be activated when new data are fetched. This simplifies the circuit of Fig. 29 to the one shown in Fig. 30. The `push` signal is slightly different from its upstream `pop` or `fetch`: it should be activated at the same time as the upstream `pop` or `fetch` (i.e., when data arrive from the upstream FIFO or shift register), except when no valid data are available at the input of the FIFO. This only happens during circuit startup, when shift registers between FIFOs contain uninitialized data. This means that the `push` signal of an FIFO should activate every time the upstream `pop` or `fetch` signal is active except for the first n times, with n as the total length of the shift registers between the successive FIFOs.

The overall design is shown in Fig. 31. The controller generates control signals for fetch, pop, and datapath operations, of which we need to determine the iteration domains. As in

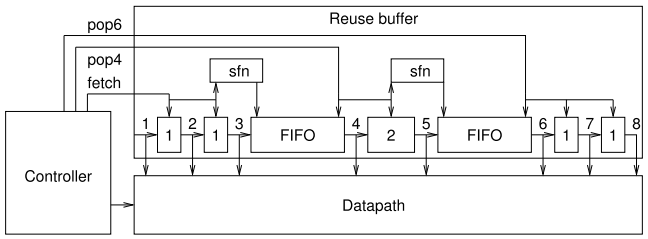

Fig. 31. Top-level design of the triangular Sobel edge detector, with the details of the reuse buffer control signals. The `sfn` blocks suspend the first n pulses.

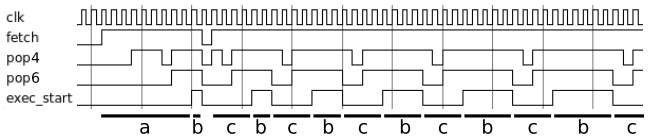

Fig. 32. Control signals of the triangular Sobel edge design. (a) Prefill buffer, (b) fetch and do calculations (one additional fetch and calculation per line), and (c) fetch additional pixels at the start and end of every line.

the rectangular case, the execute domain is the domain of the original loop nest, and the fetch domain (i.e., when the `fetch` signal must be activated) is found as the inverse projection of the data domain of the reuse set onto the iteration domain using the access function of the head of the reuse chain. Similarly, the iteration domains of each of the `pop` signals are found by calculating the inverse projection of the data domain of the reuse set onto the iteration domain using the access function associated with the reuse buffer tap right after the FIFO. Fig. 32 shows the waveforms of the control signals. Three different phases can be seen: 1) filling the buffer before the first calculation; 2) fetching a pixel and doing calculations; and 3) fetching additional pixels at the start and the end of every line, i.e., at the start and the end of the inner loop. In order to build the hardware design, it is also important to know the maximum number of data elements in the FIFO. At any time, the number of data elements in the FIFO equals the number of pushes minus the number of pops. The maximum number of data elements is the upper bound over the loop nest.

The triangular Sobel filter of our example has two variable length sections. The reuse buffer design is shown in Fig. 30. The algorithm to calculate in which loop iterations data are fetched from main memory and popped from the FIFOs is given in Fig. 33. The execution domain (line 1) is now triangular. The data and fetch domains are found the same as with a constant reuse distance (lines 2–6). In lines 7 and 8, the pop and push operations of the FIFO between taps 3 and 4 are found by projecting the data domain onto the iteration space using the inverse of the fourth and third access functions, respectively. From these, the maximum number of data elements in the FIFO can be calculated as in line 9. For the FIFO between taps 5 and 6, the procedure of lines 7–9 must be repeated with the proper access functions. In Section IX-E, experimental results are discussed for a number of designs with variable reuse distances.

D. Adding Writes

Supporting arrays with both read and write accesses require changes to the reuse buffer itself as well as to the controller.

1: $ED := [rows, cols] \rightarrow \{\ S[i,j] : j \geq 1$ and $j \leq i-1$ and $i \geq 1$ and $ilerows - 2\ \}$; ▷ iteration domain (iteration space)
2: $ACC1 := [rows, cols] \rightarrow \{\ S[i,j] \rightarrow A[1+i, 1+j]\ \}$; ▷ access functions as in Figure 24
3: [...] ▷ ACC1: head of reuse chain
4: $ACCS := ACC1 + ACC2 + ACC3 + ACC4 + ACC5 + ACC6 + ACC7 + ACC8$; ▷ union of access functions
5: $DD := ran(ACCS * ED)$; ▷ data domain of reuse set (data space)
6: $FD := ran(ACC1^{-1} * DD)$; ▷ data fetch domain (iteration space)
7: $POP4DOM := ran(ACC4^{-1} * DD)$; ▷ iterations with pop at tap 4
8: $PUSH3DOM := ran(ACC3^{-1} * DD)$; ▷ iterations with push at tap 3
9: $MAXFIFO34 := ub(card(PUSH3DOM >>= PUSH3DOM) - card(PUSH3DOM >>= POP4DOM))$; ▷ max. length of FIFO

Fig. 33. Calculating fetch and pop domains (iscc syntax [18]).

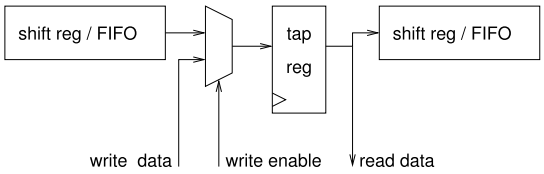

Fig. 34. Read/write register in reuse buffer.

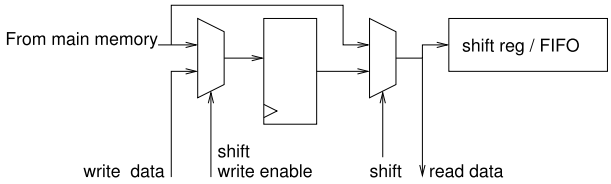

Fig. 35. Read/write register at start of the reuse buffer.

The reuse buffer has to accept data at intermediate points and modified data need to be written back to the main memory. As an (artificial) example, we assume a modified Sobel filter that modifies input pixels P[r+1][c-1] and P[r][c-1].

To enable the reuse buffer to accept writes, the depth of the shift register or FIFO directly upstream from the tap that has write accesses is decreased by one, and a register is added as in Fig. 34. A multiplexer at the data port of the register selects between data from the reuse buffer or the datapath. If writes occur at the first tap of the reuse chain, a modified design is required as shown in Fig. 35. A second modification is required to write modified data from the reuse buffer into the main memory. Data could be stored as they leave the reuse buffer after the last tap, as in Fig. 36. This solution is correct but introduces unnecessary latency if the last write tap is not the last tap of the reuse buffer. Writing data directly after the last write tap as in Fig. 37 improves the latency.

The third modification is related to the loop controller, which now coordinates two additional operations: writing data to the main memory and potentially shifting the reuse buffer to skip data that do not need to be written, as in Fig. 38. The data domain of the written array elements is called the *write data domain*. Projecting the write data domain onto the iteration space using the inverse access function of the last write tap gives the *write iteration domain*. If the reuse buffer contains only one write tap, the write iteration domain is a subset of the execution domain. If there are multiple write taps, additional write cycles after the last loop body execution may be required to store all the data. The additional shift operations for writing back data are found by projecting the entire data domain onto the iteration space using the inverse access function of the last

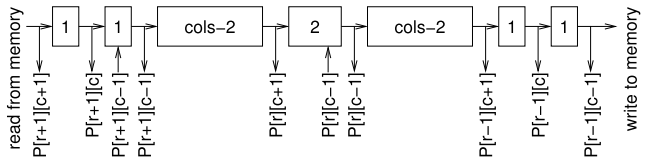

Fig. 36. Writing data from the end of the buffer (inefficient).

write tap and by limiting this domain to the iterations after the last loop body execution and up to the last write. This way, only data that are known to be written by the datapath will be written to main memory. For this reason, a clean/dirty bit like in cache memories is not needed.

E. Experimental Results

We compare our design methodology with a few other approaches. For the evaluation of our methods and our tool, we have always used unoptimized C code as input. The reuse buffer and loop controller were generated with our tool, while the datapath was generated using Catapult C. Note that the complexity of the reuse buffer design depends on the complexity of the data access pattern (i.e., the shape and size of the data window) and not on the complexity of the algorithm being implemented.

A first set of experiments compares our methodology with designs that are generated entirely by Catapult C from either unoptimized or hand-optimized source code of the Sobel filter. Pixel intensity is represented by 8-bit values. With an unoptimized source code, no data reuse happens between successive iterations. Optimizations to force Catapult C to introduce a reuse buffer were not trivial. Designs were generated both with and without pipelining. With pipelining, the Initiation Interval (II) was optimized to the best that Catapult C could achieve. The designs were targeted to a Xilinx Virtex 5 FPGA with a target clock rate of 100 MHz. RTL synthesis was done using Mentor Precision or Xilinx XPS. For benchmarking, a 100×100 pixel input image was used. One pixel per clock cycle can be read from or written to the external memories.

For a rectangular data domain, the results of the comparison are shown in Table IV. We compare the designs for performance (lowest II and latency) and area (FPGA slices, flipflops, and RAM). All 10 000 pixels from the input image get accessed, which means that the latency is at least 10 000 cycles. We first compare results without pipelining. With 39 602 cycles of latency, the circuit generated by our tool largely outperforms the circuit generated using the HLS

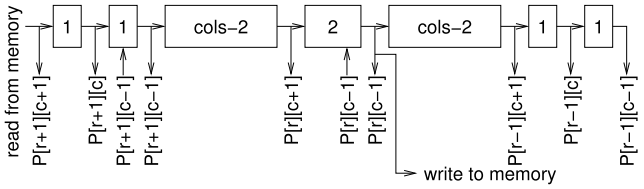

Fig. 37. Writing data from the last write tap of the buffer.

```
for (iteration domain)
  if (iteration in read_domain)
    READ_DATA_INTO_REUSE_BUFFER()
  if (iteration in shift_without_read_domain)
    SHIFT_REUSE_BUFFER_WITHOUT_READ()
  if (iteration in exec_domain)
    EXECUTE_LOOP_BODY_WITH_BUFFERED_DATA()
  if (iteration in write_iteration_domain)
    WRITE_BUFFERED_DATA_TO_MAIN_MEMORY()
```

Fig. 38. Pseudocode of the loop control in the case of read/write array accesses.

TABLE IV
EXPERIMENTAL RESULTS: RECTANGULAR

	II	Latency (cycles)	Slices	FFs	RAM (reuse buffer)
not pipelined					
Our tool, unopt. code		**39,602**	**92**	**156**	**2 (8-bit wide)**
HLS, unoptimized code		105,745	73	231	none
HLS, optimized code		39,701	91	313	1 (16-bit wide)
pipelined					
Our tool, unopt. code	1	10,002	136	180	**2 (8-bit wide)**
HLS, optimized code	1	10,397	154	469	1 (16-bit wide)

from the same source code and is slightly better than the one generated from HLS and hand-optimized code. The area is in the same range for all circuits. With pipelining of the inner loops, the HLS manages to generate a circuit with an II of 1. The optimization included the implementation of a reuse buffer in the source code, with some constructs to get around limitations of Catapult's scheduler. The latency of 10 397 cycles is higher than optimal as scheduling fails when pipelining of the outer loops is attempted. With a reuse buffer and a pipelined loop controller generated by our tool, the same II is achieved and the latency is 10 002 cycles.

The comparison for the triangular data domain is shown in Table V. The minimum latency is, in theory, 5145 cycles. Without pipelining, our design outperforms the designs generated entirely with the HLS, even if the source code for the HLS was optimized. Our design uses 404 FPGA slices, which is more than the circuit from the HLS. With pipelining, the latency of our design is close to the theoretic minimum. As in the rectangular case, Catapult C requires considerable hand-crafted optimizations to enable pipelining of the inner loops and is unable to pipeline the outer loop. Simulation has shown that in this case, the design generated with Catapult C has a performance penalty of seven clock cycles per iteration of the outer loop (i.e., about 700 clock cycles in total) with respect to our design.

A second set of experiments compares this paper with [10]. We have used the same benchmarks: 2-D bicubic interpolation, 2-D and 3-D denoise, 3-D segmentation, and 2-D Sobel edge detection. The iteration domain was always rectangular, but the shape of the data windows causes variable reuse

TABLE V
EXPERIMENTAL RESULTS: TRIANGULAR

	II	Latency (cycles)	Slices	FFs	RAM (reuse buffer)
not pipelined					
Our tool, unopt. code		**19,800**	**404**	**531**	**2 (8-bit wide)**
HLS, unoptimized code		53,461	85	248	none
HLS, optimized code		26,528	132	306	1 (16-bit wide)
pipelined					
Our tool, unopt. code	**1**	**5,152**	**292**	**252**	**2 (8-bit wide)**
HLS, unoptimized code	8	38,818	89	326	none
HLS, optimized code	1	5,831	165	500	1 (16-bit wide)

TABLE VI
EXPERIMENTAL RESULTS: BENCHMARKS

		BRAM	Slices	f_{clk} (MHz)
Bicubic	[10]	2	493	238.3
	ours	**1**	**95**	**218.0**
Denoise 2D	[10]	2	636	221.3
	ours (woc)	**0**	**199**	**213.0**
	ours (wc)	**0**	**196**	**208.2**
Sobel	[10]	2	1,088	235.9
	ours	**0**	**104**	**268.9**
Denoise 3D	[10]	2	895	210.0
	ours	**2**	**230**	**213.4**
Segmentation 3D	[10]	2	2,251	200.6
	ours	**2**	**398**	**201.2**

distances in most of the benchmarks. We have targeted a Xilinx Virtex7 FPGA (XC7VX485T) using the ISE 14.2 tools with a target clock rate of 200 MHz. The II for all designs is 1. These designs all have the correct data access pattern, but they contain a dummy datapath and do not perform the actual calculations from the algorithm. The results are shown in Table VI.

All designs meet the target clock rate of 200 MHz and have an II of 1, so they perform equally well in terms of speed and throughput. The size of our designs, expressed in numbers of FPGA slices, is between 68% and 90% smaller than the designs in [10]. Some of our designs also use fewer block RAMs (BRAM), which is due to the small size of the iteration and data domains in these benchmarks, causing small memories to be synthesized as a distributed RAM instead of BRAM. With larger iteration domains, the BRAM counts would be the same.

Both sets of experiments show that our tool performs better than state-of-the-art commercial and academic HLS design tools, without requiring the designer to make difficult optimizations in the source code.

X. CONCLUSION

In this paper, we have presented an automated method to detect data reuse in a loop nest, to select a data reuse buffer template and to generate the RTL design of data reuse buffers from the template that are tuned for the application. The tool also generates the design of a loop controller that coordinates calculations and memory operations. Several use cases are supported, including different data access patterns, constant and variable reuse distances, as well as mixed read and write operations. Experiments demonstrate that in comparison with commercial HLS, our tool does not need manual source

code optimization to achieve an equal or a better circuit performance.

ACKNOWLEDGMENT

The authors would like to thank Cong *et al.* [10] for sharing their benchmarks.

REFERENCES

[1] P. Coussy and A. Takach, "Guest editors' introduction: Raising the abstraction level of hardware design," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 4–6, Jul. 2009.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul./Aug. 2009.

[3] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, 2012.

[4] W. Meeus and D. Stroobandt, "Automating data reuse in high-level synthesis," in *Proc. Design Autom. Test Eur. Conf.*, 2014, p. 298.

[5] University of California at Riverside. *Riverside Optimizing Compiler for Configurable Computing*. [Online]. Available: http://www.jacquardcomputing.com/roccc

[6] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *Proc. ACM SIGPLAN/SIGBED Conf. Lang., Compil., Tools Embedded Syst. (LCTES)*, Jul. 2004, pp. 249–256.

[7] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, New York, NY, USA, 2012, pp. 1233–1238.

[8] K. Campbell, W. Zuo, and D. Chen, "New advances of high-level synthesis for efficient and reliable hardware design," *Integr., VLSI J.*, vol. 58, pp. 189–214, Jun. 2017.

[9] S. Verdoolaege, "Polyhedral process networks," in *Handbook of Signal Processing Systems*, S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds. New York, NY, USA: Springer-Verlag, 2013.

[10] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *Proc. 51st Annu. Design Autom. Conf. Design Autom. Conf. (DAC)*, New York, NY, USA, 2014, pp. 77:1–77:6.

[11] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops," in *Proc. 35th Int. Conf. Comput.-Aided Design (ICCAD)*, 2016, pp. 77:1–77:8.

[12] C. Alias, A. Darte, and A. Plesco, "Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*, 2013, pp. 575–580.

[13] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2013, pp. 29–38.

[14] H. Kung and C. Leiserson, "Systolic array apparatuses for matrix computations," U.S. Patent 4 493 048, Jan. 8, 1985.

[15] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst*, vol. 24, no. 1, pp. 65–109, 2000.

[16] S. Girbal *et al.*, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, 2006.

[17] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral loop transformations to work," in *Proc. Int. Workshop Lang. Compil. Parallel Comput. (LCPC)*, Oct. 2003, pp. 209–225.

[18] S. Verdoolaege. (Jun. 2007). *Barvinok: User Guide*. [Online]. Available: http://freshmeat.net/projects/barvinok

[19] Reservoir Labs. *Jolylib*. Accessed: Mar. 24, 2017. [Online]. Available: https://www.reservoir.com/

[20] IRISA. *Polylib*. Accessed: Mar. 24, 2017. [Online]. Available: http://www.irisa.fr/polylib

[21] R. Sass and M. Mutka, "Enabling unimodular transformations," in *Proc. ACM/IEEE Conf. Supercomput.*, Los Alamitos, CA, USA, Nov. 1994, pp. 753–762.

[22] J. Xue, "Unimodular transformations of non-perfectly nested loops," *Parallel Comput.*, vol. 22, no. 12, pp. 1621–1645, 1997.

[23] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, E. H. D'Hollander, and D. Stroobandt, "Finding and applying loop transformations for generating optimized FPGA implementations," in *Transactions on High-Performance Embedded Architectures and Compilers I* (Lecture Notes in Computer Science), vol. 4050. Berlin, Germany: Springer, 2007, pp. 159–178.

[24] S. Verdoolaege. (Dec. 2014). *ISCC Tutorial*. [Online]. Available: http://barvinok.gforge.inria.fr/tutorial.pdf

Wim Meeus received the M.Sc. degree in electrotechnical engineering from Ghent University, Ghent, Belgium, in 1996.

He is currently a Researcher at the Computer Systems Laboratory, Department of Electronics and Information Systems, Ghent University. His current research interests include digital circuit design, high-level synthesis, and more specifically the automatic generation of application-specific memory architectures.

Dirk Stroobandt (S'92–M'98) received the Ph.D. degree in electrotechnical engineering from Ghent University, Ghent, Belgium, in 1998.

He was a Visiting Researcher at the University of California at Irvine, Irvine, CA, USA, in 1997, and at the University of California at Los Angeles, Los Angeles, CA, USA, from 1999 to 2000. He is currently a Full Professor at the Computer Systems Laboratory, Department of Electronics and Information Systems, Ghent University, where he also leads the research group Hardware and Embedded Systems with interests in semiautomatic hardware design, run-time field-programmable gate array reconfiguration, and reconfigurable multiprocessor networks.