Abstract—High computational effort in modern signal and image processing applications often demands for special purpose accelerators in a system on chip (SoC). New high level synthesis methodologies enable the automated design of such programmable or non-programmable accelerators. Loop tiling is a widely used transformation in such methodologies for dimensioning of such accelerators in order to match inherent massive parallelism of considered algorithms with available functional units and processor elements. Innately, the applications are data-flow dominant and have almost no control flow, but the application of tiling techniques has the disadvantage of a more complex control and communication flow. In this paper, we present a methodology for the automatic generation of the control engines of such accelerators. The controller orchestrates the data transfer and computation. The effect of tiling on area, latency, and power overhead of the controller is studied in detail. It is shown that the controller has a substantial overhead of up to 50% in for different tiling and throughput parameters. The energy-delay product is also used as a metric for identifying optimal accelerator designs.

I. INTRODUCTION

In order to meet demanding challenges of high performance embedded computing and stringent power constraints, there is a gradual trend towards programmable or special purpose accelerator-based heterogeneous multi-processor system-on-chip (MPSoC) designs [1]. The major challenge to be surmounted for these architectures is the lack of mapping tools. Several electronic system level (ESL) synthesis tools and compilers partly answer the problem by enabling automatic generation of RTL descriptions or program code for domain specific acceleration engines from high level languages. They usually focus on compiling computationally intensive loop programs for the accelerators. The techniques used in such tools have their origins in compiler transformations like scheduling, allocation, loop unrolling, loop tiling (partitioning), etc.

A formidable problem on integration of accelerators in MPSoCs is the interface synthesis. In this context number of questions have to be answered: What should be the number and size of the local buffers? Accelerators hide the memory latency to provide high I/O bandwidth by including multiple local buffer banks as shown in Figure 1. How would data transfer to/from local buffer correspond to the static schedule and dynamic system behavior? A controller generates and processes status signals (start, done, empty, full, ..) for interaction with external components. The I/O controller is also responsible for generating enable signals, address and evaluating FIFO status flag for data transfer from the memory to the processor array of accelerator. What is the effect of memory, I/O channels and their controller on the accelerator in terms of area, power and throughput? Often, the performance bottleneck lies in clock delay of the critical path in controller architecture. Also the area requirements of controller could be larger than the accelerator data-path in certain cases. Therefore, it is important to study the controller overhead and its relation to compiler transformations like loop tiling, and scheduling. In this paper, we

- present an efficient methodology for automatic generation of the controller engines, which orchestrates data transfer and computation for the accelerators.
- analyse the effect of loop tiling and throughput requirements on area, latency, and power overhead of the controller.
- study the problem of selection of an optimal accelerator design using the energy-delay product as metric after initial pruning of design based on minimum throughput and maximum power budget.

In Section II, a brief overview of considered accelerator structures and compilation techniques is given. The methodology for controller generation for coordinating I/O access and data computations is presented in section III. The effect of compiler transformations on controller overhead and selection
of optimal accelerator design is dealt with in Section IV and Section V. Finally, after discussing related work, the conclusion is presented in Section VI and VII.

II. A Generic Accelerator and Compilation Model

In this section, we present an overview of components making up an accelerator. The different compiler transformations for obtaining or programming different architecture infrastructures and their influence on different design objectives is also discussed.

A. Accelerator Scheme

Accelerators may vary in terms of granularity ranging from RTL descriptions on fine-grained FPGAs to programs on coarse-grained processors like PACT-XPP[2], WPPA[3]. Here, we consider accelerators generated from loop program as shown in Figure 1. The computation unit of accelerators is usually a 2D processor array consisting of (non)-programmable, reconfigurable processor elements (PEs). The data-path of the PEs may contain a varying number of functional units (FUs), registers, data memory for different architectures. The binding of operations of a given loop kernel on limited resources in the PE causes multiplexers in front of FUs or registers which in turn leads to overhead in the local control path. The control path contains a finite state machine (FSM) logic for controlling these multiplexers. In case of programmable accelerators, the local control path may consist of a program memory, program counter and instruction decoder. In addition, a global control and counter unit which track the schedule order of iterations is recommended for address generation. For instance, in image processing applications, the counters can keep a track of pixel co-ordinates and generate control signals for special border processing. These signals are propagated to the PEs and I/O control path through the interconnect network. Several banks of addressable memory or FIFOs are available at the border for providing the necessary data bandwidth. The global signals are used in I/O controller for address generation and enable signals for the correct synchronization of communication with external SoC components.

B. Compiler transformations

For design entry and programming of the above accelerators a modified version of C and other functional languages are accepted for data flow-based algorithm descriptions [4]. The typical design trajectory to generate such an accelerator is shown in Fig. 2. The starting point is a data flow graph representing loop level parallelism and operation level parallelism (see Fig. 3). The local allocation and scheduling takes into account resource constraints (e.g. number of FUs in each PE) or throughput requirements to decide the binding and execution time of the operations in loop kernel, respectively. Whereas, the global allocation of processor elements and memory requirements is determined by tiling (also known as partitioning) schemes. It is a well known transformation which covers the iteration space of a given nested loop specification using congruent tiles. Loop tiling for processors is executed for better register and memory reuse through different schedule orders of loop iterations. In context of high level synthesis, different tiling schemes and tile sizes determine the size and dimension of the processor array, individual PEs, and number of memory banks in the accelerator. A schedule may be obtained by solving a latency minimization problem formulated as a mixed integer linear program similar as in [5]. As a result, one obtains:

- an optimal schedule vector, $\lambda$ which determines the order of execution of iterations. The minimal latency $L$ and start times of each operation within one loop iteration, $\tau(v_i)$ and their resource binding.
- The iteration interval (II): The iteration interval II is the number of time steps (clock cycles) between the evaluation of two successive instances of a variable within one processing element which is representative of the throughput [5].
- Loop matrix: A loop matrix $R = (r_1 \ r_2 \ldots r_s) \in \mathbb{Z}^{s \times s}$ determines the schedule ordering of index points within a tile. Index points in direction of $r_1$ are executed sequentially and then incremented in direction $r_2$. The ordering is similar to a sequential nested loop program where the loop index $i_k$ corresponds to iterations in direction of $r_k$. The inner loop index is $i_1$, and the outermost loop index is $i_s$.

Therefore, overall start time of node $v_i$ at iteration point $I$ is: $t(v_i(I)) = \lambda \cdot I + \tau(v_i)$. The hardware synthesis produces an intermediate RTL representation which is retargeted to VHDL or program code in C or machine language by the backend. The freedom in choosing partitioning and scheduling parameters leads to different designs implementing a given algorithmic specification. Now, a global controller and counters are required for address generation, I/O control, and execution of processor independent program-conditionals corresponding to global schedule, $\lambda$ for different designs. The operation scheduling, $\tau(v_i)$ leads to local controller overhead for directing the multiplexers before functional units and registers. Therefore, the automatic generation of a controller is inevitable for orchestrating static schedule and dynamic interaction of the
accelerator with other system components. Before explaining a solution approach, we explain the partitioning and scheduling with help of the following example for understanding the methodology and their effect on the controller for resulting hardware accelerator.

**Example 1:** The pseudo-code shown in Algorithm 1 represents a loop nest of an image filter. The image dimension is $M \times M$. $N$ denotes the size of the one dimensional filter mask. For sake of simplicity equations denoting border handling have been neglected. The corresponding data flow graph is shown in Figure 3(a) with $M = 8$ and $N = 4$. The variable $A, U, Y$ denote the input mask coefficients, input image pixels, and output image pixels respectively. The FORALL loop represents the computation domain and loop level parallelism of the algorithm. The iteration conditions determine the conditional execution of statements including input and output assignment statements. The tiles denote partitioning of iteration space which lead to allocation of two corresponding processors. Where, each processor executes the points within the tile sequentially. Formally, this introduces a extra loop dimension, $m_1$ and $m_2$ instead of $m$, similar as in loop blocking [6]. Due to selected partitioning, the iteration variables $m_2, i, j$ are executed sequentially. Whereas iteration with same $m_1$ are executed on same processor, whose processor index is given by value of $m_1$. The scheduling for the problem in Figure 3 gives $\lambda = (4 3 2 3 2)$ for the iteration vector $I = (i j m_1 m_2)$. The start time of each operation in loop kernel is shown in Figure 3(b). The obtained iteration interval is $II=2$ which can be explained by availability of only a single adder.

Therefore, linear transformations are used for allocation and scheduling[7] in order to assign a processor index $p = QI$ (space) and a sequencing index $t = \lambda I$ (time) to index vectors $I \in \mathcal{I}$ of the iteration space $\mathcal{I}$ defined by the loop program. $Q$ is a linear allocation matrix which is determined by the tiling strategy (like LPGS, LSGP and hierarchical partitioning) and matrices. This matrix contains information on which loop variable is executed sequentially or in parallel respectively.

**III. AUTOMATIC GENERATION OF CONTROLLER ENGINE**

The controller engine is not only responsible for evaluation of global conditions, but also synchronization of I/O communication from memory banks. Its automatic generation requires knowledge of undertaken scheduling order. partitioning and dependence graph. In the next subsections, we present different aspects of its generation.

**A. BUFFER MODELLING AND SYNTHESIS**

The partitioning determines the allocation of the loop iterations to the PEs through the allocation matrix, $Q$. Therefore, the iteration conditions corresponding to I/O variables, $I_i$ in the loop program specify the processor elements $(Q \cdot I_i)$ responsible for I/O communication. For Example 1, the input and output variables, $U$ and $Y$ are associated with iteration conditions, $m = 0$ and $m = N - 1$, respectively. This corresponds to $m_1 = 0/m_2 = 0$ and $m_1 = 1/m_2 = 1$ for $N = 4$ partitioned program. Hence, the input, $U$ and output variables, $Y$ are mapped to two memory banks connected to PE(0) and PE(1) corresponding to processor dependent part of conditional i.e. $m_1 = 0$ and $m_1 = 1$, respectively. The connection of memory banks to PEs are also shown in

---

**Algorithm 1** The pseudo-code in this example represents a loop nest of an image filter.

```plaintext
variable A in 2 integer <16>
variable U in 2 integer <16>
variable Y out 2 integer <16>
FORALL (i >= 0 and i <= M-1)
{FORALL (j >= 0 and j <= M-1)
    {FORALL (m >= 0 and m <= N-1)
        {a[i,j,m] = A[0,j,m] \* LUT(u[i,j,m]-u[i,j,0])
            IF (m=0) THEN
            {u[i,j,m] = U[i,j] // Read input
                z[i,j,m] = a[i,j,m] \* u[i,j,m]
                s[i,j,m] = 0 + a[i,j,m]
                y[i,j,m] = 0 + z[i,j,m]
            } ELSE
                {z[i,j,m] = a[i,j,m] \* u[i,j,m]
                    s[i,j,m] = s[i,j,m-1] + a[i,j,m]
                    y[i,j,m] = y[i,j,m-1] + z[i,j,m]
                }
            }
        IF (i=0) THEN
            u[i,j,m] = 0
        ELSE
            u[i,j,m] = u[i-1,j,m-1] //data reuse
        }
        IF (m == N-1)
            Y[i,j] = y[i,j,m]/s[i,j,m] // Output
    }
}
```
Figure 4. The problem of mapping input and output array variables onto the accelerator memory requires the consideration of an allocation problem. The programmable accelerator architectures provide the programmer only a restricted number of memory banks for assignment of input and output variables. For a given data flow graph, the selected tiling determines not only the computation allocation (number of PEs) but also the data allocation (memory buffers). If $N$ is the available number of memory banks, a valid partitioning which does not violate the memory constraints and must satisfy the following equations.

$$n_{tot} \leq N, \quad n_{tot} = \sum_{i \in I/O} n_i$$  \hfill (1)

where $n_i$ is the number of memory banks for storing $i^{th}$ input or output variable. The number of memory banks, $n_i$, for each individual I/O array variables can be found as number of elements in the polytope $Q \cdot I_i$. This constraint can also be part of integer linear program for the scheduling problem. The selection of the buffer size depends on many factors such as data dependencies between loop nests, communication and synchronization overhead. In case of fine granular HW-HW communication between accelerators, communication and synchronization overheads are negligible. Exact buffer requirement analysis considering data dependencies in the polytope model as presented in [8] is being integrated. The buffer sizes for each input or output variable are chosen such as all memory accesses within one tile fit in the buffer. Thus, the size corresponds to the transformed volume\(^1\) of the tile, $n = |\{Q I | I \in I_{tile}\}|$. Since the matrix $Q$ is not square it seems that the transformed set cannot be represented by a dense polytope. But the Hermite kernel [9] of matrix $Q$ has a determinant of one and thus the space can be projected to a dense polytope using the Fourier-Motzkin elimination [10]. Subsequently, the number of points (volume) can be easily obtained by counting or the application of sophisticated techniques such as the computation Ehrhart polynomials [11].

The memory can then be synthesized or configured in FIFO mode as it is viable for fine granular communication between accelerators. The HW-SW communication between accelerator and processors can take place over different communication alternatives. However, different rates of data production and control overhead for the exchange of status signals for each data-transfer would be a significant bottleneck. This recommends applying a burst model of data transfer. The configuration of memory as addressable memory (RAM) is apt for such scenario. Furthermore, out of order communication can be handled only with addressable memory. In the next subsections, we discuss address generation, FIFO control, synchronization mechanisms for both the modes.

B. Local Controller and Global Controller

Scheduling is carried out in two phases, i.e. for operations inside the loop kernel and for determining the global execution order of iterations, respectively. Therefore, scheduling and resource binding map operations and variables onto a minimum number of FUs and registers [12]. The combinational logic inside the FSM of the local controller determines the select signal for the multiplexers in front of a functional unit or a register. It also determines processor dependent iteration conditions which arises due to special conditions (e.g., border handling in image processing, threshold decisions). The global counter shown in Fig. 4 is responsible for generating loop index variables corresponding to the executed iterations. It is determined by the global scheduling order. These variables are responsible for address generation, memory control signals, and evaluation of processor independent iteration conditions. These iteration conditions arise mostly due to data-reuse opportunities from partitioning. I.e., whether one should read data from memory, propagated value from neighboring processor, or internal registers. The global decoder uses the index variables from the global counter to evaluate the processor independent iteration conditions of all PEs. The control signals are then propagated into the processor array using interconnect delay registers [13]. For example 1, the global counter keeps track of loop counters for the image co-ordinates $i$, $j$ and mask $m_2$. The strides are stored in a look-up table (LUT) which is responsible for incrementing the counters, i.e., $(0,0,1), (1,0,-1)$ and $(-M+1,1,-1)$. The increments are selected on satisfaction of condition $m_2 == 1$, $i < M - 1$ and $i == M - 1$ every $I$ cycles, respectively. The local decoder is realized by an FSM with two states corresponding to $I=2$. The global decoder determines the if conditionals in the loop kernel. The methodology for determining strides, increment conditions, and global decoder conditions is also presented in [13].

The key characteristic of our methodology is the use of combined global and local control facilities. This hybrid version of control path lies between complete global control path for SIMD architectures and a local control path for multi-
processor architectures. This strategy reduces the required control overhead area at cost of delay registers for propagation and improves the clock frequency.

### C. Memory Controller Synthesis

All the statements in the given loop specification which assign I/O variables are related to the generation of memory control signals. They are generated with help of output boolean signals evaluated according to the variables iteration condition calculated in the global decoder. In case of FIFO mode, these signals are evaluated to generate read/write enable and valid signals. For the RAM interface, the address values are needed in addition to the R/W enable. The scheme of a memory controller is shown in Fig. 5. The mutual components of the memory controller for both FIFO and addressable memory are

- Modulo-II counter: Since I/O access takes place every II cycle.
- Hold signal: In case the counter is disabled, the I/O access is also not allowed. This situation arises in case of non-tight schedules.
- Global control signals such as conditions from decoder, counter enable are evaluated together for valid I/O access. The counter enable indicates whether the values from loop counters are still in the iteration space. I/O access will be forbidden, if this condition is not satisfied.

These common signals for FIFOs/RAMs are and-ed together for enabling/disabling the memory controller. The valid signal from FIFO state check logic shown in Fig. 5 is responsible for identifying the state of computation kernel, whether input/output data can be read/written from/to FIFOs. The W/R enable and full flag/empty flag from each FIFO are connected to its I/O controller as input of state check logic. Each input and output variable can be mapped onto multiple memory banks. The schedule order of a variable access is same for all memory banks. Hence, only a single controller or address generation unit for each variable is needed. The memory signals are propagated with delays to the memory banks. These delays correspond to the offset in start-time of the FIFOs which belong to same variable. The length of the delay register can be defined by following equation: $t_d = \lambda \cdot d$, where $d$ is the iteration dependence vector of the variable. This optimization avoids dedicated I/O controller for each memory bank.

All valid signals guarantee that the functional units inside computation kernel can execute further even though some input/output FIFOs are empty/full, if the I/O access to the FIFOs are not required. Only when the I/O controller generates valid signal and common signal is true, the read or write enable is generated. In case of RAM, the loop counter variables are used for address generation. The address of input and output data corresponds to its iteration position in the polytope. Our methodology combines traditional incremental address generation and custom address generation for the sake of optimization. For the 1D address space, only a single loop counter is required to increment the address every II cycles. For 2D or 3D address space, all loop counters are summed together for address generation: In our example 1, the global decoder generates control signals corresponding to $m = 0$ and $m = N - 1$ for the input and output FIFO controller, respectively (i.e. $m_2 \equiv 0$ and $m_2 \equiv 1$ for the partitioned program). The conditional is true every 4 cycles. In case of FIFO interface, As shown in Fig. 6, variable A is mapped onto PE(0) and PE(1). The read enable for each FIFO and its empty flag are required to feedback its state to I/O controller A for correct next access. All these signals along with clock enable (CE) from outside are evaluated to determine clock enable (CE') of computation kernel. The loop counter, $m_2$, $i$, and $j$ are used for address generation for memory U and Y. Since, after the partitioning, $m$ is replaced by $m_1$ and $m_2$, whereas only $m_2$ denotes processor independent part. The execution order of processor array $m_2$, $i$, $j$. The Address of variable U is given by $m_2 + (N/2 - 1) \cdot i + (N/2 - 1) \cdot M \cdot j$. It is to be noted that $m_2 = 0$ for variable U. In the next subsection, we will discuss the coupling of an accelerator in an SoC environment.

### D. SoC Integration

The accelerators can be embedded into an SoC by coupling it to a processor or another IP in a multi-accelerator system.
The coupling can take place over point-to-point FIFO links, bus, or in some cases a network on chip (NoC) interfaces. Therefore, external communication requires synchronization mechanisms, which in case of FIFO takes place with help of status flags (empty, full) stored in control registers. In case of a RAM interface, the external control for reading and writing from output and input ram is done by setting start and observing the finish flag as shown in Figure 7. A wrapper containing BRAM controller, glue logic and accelerator has been used to provide a generic memory-mapped interface. This provides the advantage of flexible connection to different buses. Since, all buses are provided with memory controller with similar interface. The glue logic does the address decoding of accelerator data and synchronization signals for memory mapped access. In event of input and output data communication over bus, a DMA or software running on processor is used to transfer data to and from the accelerator as defined by the schedule. The out of order communication between accelerators is supported by multi-dimensional FIFOs [8]. The RTL functionality is verified by auto-generated VHDL testbench with input and output data created by functional simulation of accelerator.

IV. IMPACT OF COMPILER TRANSFORMATIONS ON CONTROLLER OVERHEAD

Our design tool has been used to quantify the effect of loop tiling and requisite iteration interval on control overhead in terms of logic area, power, and performance for the following algorithms as benchmarks:

- **MMM**: matrix-matrix multiplication
- **BPF**: band-pass filter
- **IDCT**: inverse discrete cosine transform
- **CONV**: 2-d image convolution
- **SOBEL**: image edge detection algorithm

As a result, the fraction of control overhead in terms of logic area for these algorithms lies between 10-50% as shown in Table I. The accelerator configuration is a $4 \times 4$, $3 \times 3$, $1 \times 1$ processor array for (MMM, BPF), (CONV, SOBEL), and IDCT, respectively. The hierarchical analysis of the design was done using Xilinx floorplanner and individual synthesis of the controller. The Fig. 8 shows the effect of partitioning parameters on control area overhead for BPF benchmark. It can be seen that with tiling matrices corresponding to larger number of PEs, the control area overhead converges to a constant fraction. In this case, controller area overhead is small at around 15% of total area of the accelerator. The overhead converges for larger accelerators due to the constant cost of the global and I/O controller. Whereas the total size of local control scales linearly with the number of PEs. Therefore, it can be concluded that due to partitioning, one does not increase the fraction of area overhead of the controller. The IDCT accelerator was also synthesized with different throughput requirements. It was observed that higher II was associated with larger controller because of higher number of states for local control FSM. The accelerator with decreasing throughput (i.e., increasing iteration interval) is associated with lower resource requirements in terms of functional units like multipliers and adders. Therefore, control overhead as part of total design increases as shown by orange bars in Fig. 9. The controller cost in terms of area can take up to 50% of the IDCT accelerator design as shown in Table I and Figure 9. All synthesis results are obtained using Xilinx ISE 9.2 on the Xilinx Virtex-2 FPGA (xc2v8000-4-f1517).

The effect of partitioning parameters on achievable clock frequency is shown in Figure 10. The critical path of the clock passes through the global and local controller. Interestingly, the maximal clock frequency does not decrease with increasing
number of processor elements due to pipelining of control with interconnect delays. However, the frequency may vary for different designs varies between 100 and 120 MHz. The average power consumed by the accelerator by modelsim simulation of set of inputs involving generated accelerator RTL. The XPower tool was used for power estimation [14]. The obtained average power is multiplied with the obtained latency to obtain the total energy. For example, a energy efficiency of 20-25 MOPS/mW (considering only the dynamic power) is obtained for the matrix multiplication example as shown in Figure 10. This implies that power efficiency does not decline with larger processor arrays or controller overhead.

V. RESULTS

It has be shown that controller for I/O and execution of the accelerators for the loop programs has a substantial impact on the logic area, power and performance depending on the chosen partitioning, throughput, resource allocation as shown in previous section. For partitioning the control overhead converges to constant fraction for larger processor arrays. Whereas, an increasing iteration interval leads to higher control overhead. Therefore, the problem of finding optimal controller configurations requires finding optimal accelerator configuration (i.e., tiling, II) parameters. In order to find these parameters, we selected the energy-delay product (EDP) as a neutral metric for selecting optimal parameters [15]. The designs to be considered already satisfy a minimum performance and maximum peak power budget. The EDP is then a good metric, since the minimization of the metric results in architectural and compiler decisions contributing most to energy and area efficiency. In Figure 11, the normalized energy (pJ/op) and normalized throughput (ns/op) product for different processor array configurations again for the matrix multiplication (MMM) case study is shown. The different partitioning parameters lead to processor arrays with $2 \times 1$ to $8 \times 7$ PEs. It can be observed that for smaller accelerators the metric is quite large due to less power efficiency. The optimal value is achieved for processor arrays of size $6 \times 6$ and $8 \times 6$. Larger processor array accelerators intrinsically minimize the metric because of better energy efficiency and clock speeds. The brute force determination of optimal configuration is a tedious and time taking process because of large number of compiler, architecture parameters, as well as synthesis process for determining design objectives. This motivates the use of heuristics like simulated annealing, etc. for auto-tuning of accelerator design, similar as in code generation for multi-core architectures [16]. It also recommends the use of estimation for rapid determination of design objectives.
VI. RELATED WORK

In [17], the effect of loop unrolling on controller delay and techniques for determining optimal unroll factors meeting the controller delay budget have been presented. However, the effect of controller on power and area is not quantified. Derrien et. al. proposed models for power estimation of special purpose accelerators in [18]. The authors also find optimal tiling parameters for minimum energy consumption per PE. A detailed study of energy consumption, area, and performance has been performed exemplarily for matrix multiplication algorithm in [19]. However, no analysis of control overhead is done. The results recommend use of 1-d processor array for matrix multiplication. In [20], hardware/software interface synthesis for integration of processor accelerators, along with juggling schedules for optimal interface overhead is presented. The efficient control generation for communication in process networks is also presented in [21]. It also has a limited discussion of the effect of compiler and architecture parameters on controller.

VII. CONCLUSION

Programmable or specific acceleration engines provide a viable architecture for the high performance embedded computing algorithms. In this paper, we presented an unified methodology for controller generation and presented a way to integrate such accelerators in an SoC incorporating standard processor cores. These controllers are responsible for data transfer and synchronizing computation and communication. The synthesis results shows reasonable controller overhead in terms of area, power of the controller depending on the tiling and throughput parameters. The larger iteration interval, the higher is the control overhead. However, the controller overhead converges to constant fraction between 10 to 50% for tiling equivalent to the larger number of PEs. Furthermore, we undertake a brute force configuration space exploration of tiling parameters to determine optimal accelerator designs. In future, we would like to apply design space exploration techniques based on simulated annealing, genetic algorithms for finding optimal accelerator. In particular, better estimation and pruning techniques should be developed for fast exploration.

REFERENCES