EURASIP Journal on Image and Video Processing
a SpringerOpen Journal

RESEARCH Open Access

Pipeline synthesis and optimization of FPGA-based video processing applications with CAL

Ab Al-Hadi Ab Rahman*, Anatoly Prihozhy and Marco Mattavelli

Abstract

This article describes a pipeline synthesis and optimization technique that increases data throughput of FPGA-based system using minimum pipeline resources. The technique is applied on CAL dataflow language, and designed based on relations, matrices, and graphs. First, the initial as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules, and the corresponding mobility of operators are generated. From this, operator coloring technique is used on conflict and nonconflict directed graphs using recursive functions and explicit stack mechanisms. For each feasible number of pipeline stages, a pipeline schedule with minimum total register width is taken as an optimal coloring, which is then automatically transformed to a description in CAL. The generated pipelined CAL descriptions are finally synthesized to hardware description languages for FPGA implementation. Experimental results of three video processing applications demonstrate up to 3.9× higher throughput for pipelined compared to non-pipelined implementations, and average total pipeline register width reduction of up to 39.6 and 49.9% between the optimal, and ASAP and ALAP pipeline schedules, respectively.

1 Introduction

Data throughput is one of the most important parameters in video processing systems. It is essentially a measure of how fast data passes from input to output of a system. With increasing demands for larger resolution images, faster frame rates, and more processing requirements through advanced algorithms, it is becoming a major challenge to meet the ever-increasing desirable throughput.

For algorithms that can be performed in parallel, such as the case with most digital signal processing (DSP) applications, parallel platforms such as multi-core CPU, many-core GPU, and FPGA generally results in higher throughput compared to traditional single-core systems. Among these parallel platforms, FPGA systems allow the most parallel operations with the highest flexibility for programming parallel cores. However, register transfer level (RTL) designs for FPGA are known to be difficult and time consuming, especially for complex algorithms [1]. As time-to-market window continues to shrink, a new high-level program that synthesizes to efficient parallel hardware is required to manage complexity and increase productivity.

* Correspondence: alhadi.abrahman@epfl.ch
SCI-STI-MM, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland

The CAL dataflow language [2] was developed to address these issues, specifically with a goal to synthesize high-level programs into efficient parallel hardware (see Section 3.2). CAL is an *actor* language in which program executes based on *tokens*; therefore, suitable for data intensive algorithms such as in DSP that operates on multiple data. The language was also chosen by the ISO/IEC[a] as a language for the description and specification of video *codecs*.

CAL design environment was initiated and developed by Xilinx Inc. and later became Eclipse IDE open source plugins called *OpenDF* and *OpenForge* [3] which allow designers to simulate CAL models and synthesize to hardware description languages (HDL). The tools only perform basic optimizations for a given CAL actor for HDL synthesis; the final result highly depends on the design style and specification. Reference [4] presents coding recommendations for CAL designers to achieve best results. However, some optimizations are best performed automatically rather than manually, for example pipeline synthesis and optimization of CAL actors.

In CAL designs, actions execute in a single-clock cycle (with exception to while loops and memory access). Large actions, therefore, would result in a large combinatorial logic and reduces the maximum allowable operating frequency which in turn decreases throughput.

The pipeline optimization strategy is to partition this large action into smaller actions that satisfy a required throughput requirement, but with a minimum resource penalty. Finding a pipeline schedule that minimizes resource is a nonlinear optimization problem, where the number of possible solutions increases exponentially with a linear increase of operator mobility.

This study presents an automatic non-pipelined CAL actor transformation to resource-optimal-pipelined CAL actors that meet a required stage-time constraint. The objective is to allow designers to rapidly design complex DSP hardware systems using CAL dataflow language, and use our tool to obtain higher throughput with optimized resources by pipelining the longest action in the design. In order to evaluate the efficiency of our methodology, three video processing algorithms are designed and used for pipeline synthesis and optimization.

Figure 1 shows CAL to HDL design flow methodology with our CAL to CAL pipeline optimization strategy. Starting with an initial CAL design, it is first synthesized to HDL, then to a specific FPGA technology where the critical path and maximum allowable frequency information can be obtained. If the throughput requirement is met, the design can be implemented directly into the FPGA. In the case when a higher throughput is required, the action with the critical path is extracted from the design, and automatically pipelined with the required delay (for that actor) with minimum resource penalty. The original non-pipelined CAL actor is then replaced by the newly generated pipelined CAL actors. This process is repeated until the desired system throughput is achieved.

This article is organized as follows. The next section provides background and related study on pipeline synthesis and optimizations. Section 3 presents the basics of dataflow modeling in CAL. Following this, in Sections 4 and 5, we present our approach to pipeline synthesis and optimization using mathematical formulations. Then, in Section 6, experimental results are shown for several video processing applications, and finally, the last section concludes the article.

2 Pipeline synthesis and optimization: background

In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are executed in parallel or in time-sliced fashion; in this case, some amount of buffer storage (pipeline registers) is inserted in between elements. The time between each clock signal is set to be greater than the longest delay between pipeline stages, so that when the registers are clocked, the data that is written to the following registers is the final result of the previous stage. A pipelined system typically requires more resources (circuit elements, processing units, computer memory, etc.) than one that executes one batch at a time, because each pipeline stage cannot reuse the resources of the other stages.

Key pipeline parameters are number of pipeline stages, latency, clock cycle time, delay, turnaround time, and throughput. A pipeline synthesis problem can be constrained either by resource or time, or a combination of both [5]. A resource-constraint pipeline synthesis limits the area of a chip or the available number of functional units of each type. In this case, the objective of the scheduler is to find a schedule with maximum performance, given available resources. On the other hand, a time-constraint pipeline synthesis specifies the required throughput and turnaround time, with the objective of the scheduler is to find a schedule that consume minimum resources.

Sehwa [6] is the first pipeline synthesis program. For a given constraint on the number of resources, it implements a pipelined datapath with minimum latency. Sehwa minimizes time delay using a modified list scheduling algorithm with a resource allocation table. HAL [7] performs a time-constrained, functional pipelining scheduling using the force directed method which is modified in [8]. The loop winding method was proposed in the Elf [9] system. A loop iteration is partitioned horizontally into several pieces, which are then arranged in parallel to achieve a higher throughput. The percolation-based scheduling [10] deals with the loop winding by starting with an optimal schedule [11] which is obtained without considering resource constraints. Spaid [12] finds a maximally parallel pattern using a linear programming formulation. ATOMICS [13] performs loop optimization starting with estimating a latency and inter-iteration precedence. Operations which cannot be scheduled within the latency are folded to the next iteration, the latency is decreased, and the folding is applied again. The above-listed tools support resource sharing during pipeline optimization.

SODAS [14] is a pipelined datapath synthesis system targeted for application-specific DSP chip design. Taking signal flow graphs (SFG) as input, SODAS-DSP generates pipelined datapaths through iteratively constructive variation of the list scheduling and module allocation processes that iteratively improves the interconnection cost, where the measure of equidistribution of operations among pipeline partitions is adopted as the objective function. Area and performance trade-off in pipeline designs can be achieved by changing the synthesis parameters, data initiation interval, clock cycle time, and number of pipeline stages. Through careful scheduling of operations to pipeline stages and allocation of hardware modules, high utilization of hardware modules can be achieved.

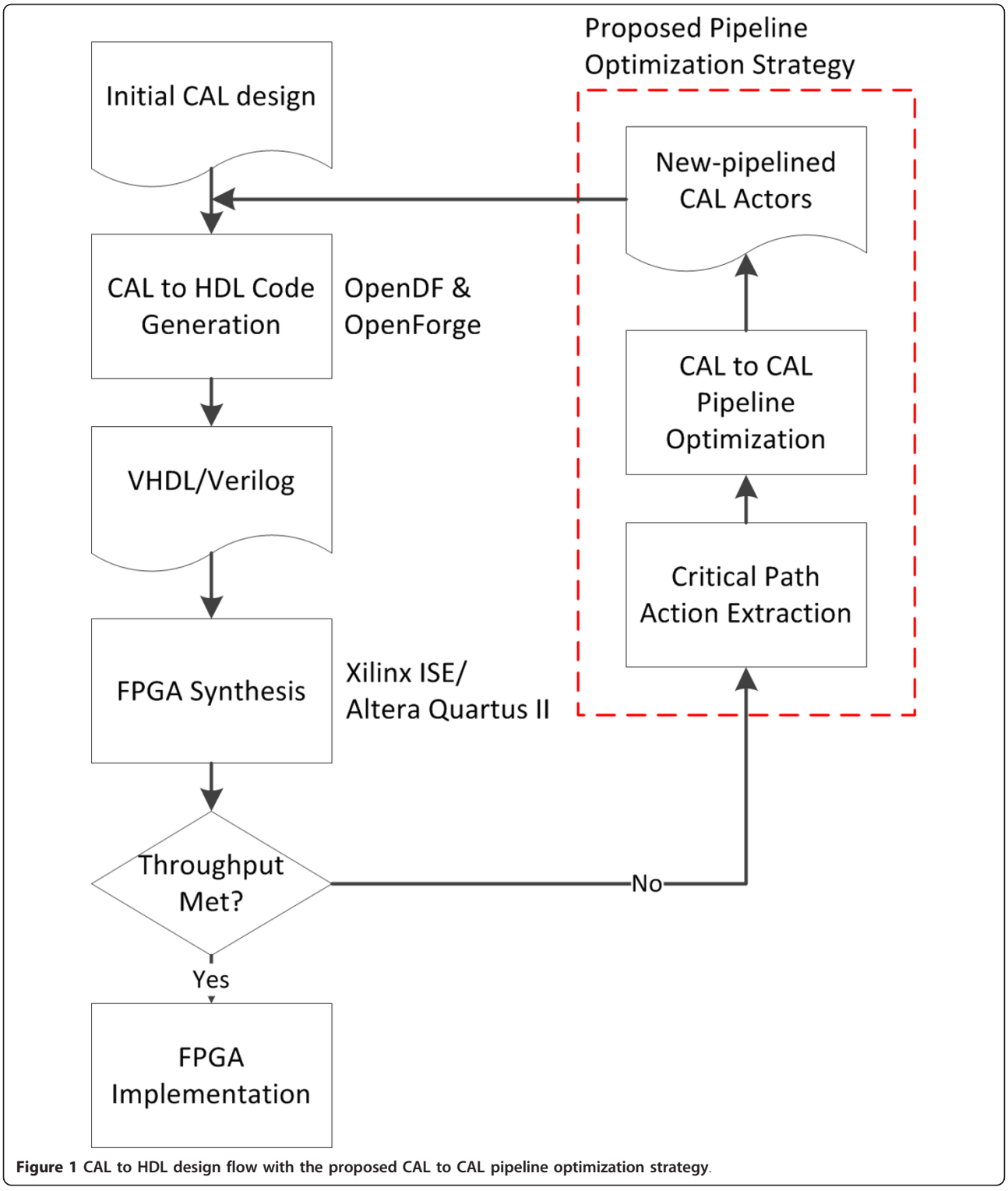


Figure 1 CAL to HDL design flow with the proposed CAL to CAL pipeline optimization strategy.

Pipelining is an effective method to optimize the execution of a loop with or without loop carried dependencies, especially for DSP [8]. Highly concurrent implementations can be obtained by overlapping the execution of consecutive iterations. Forward and backward scheduling is iteratively used to minimize the delay in order to have more silicon area for allocating additional resources which in turn will increase throughput.

Another important concept in circuit pipelining is Retiming, which exploits the ability to move registers in the circuit in order to decrease the length of the longest path while preserving its functional behavior [15-17]. A sequential circuit is an interconnection of logic gates and memory elements which communicate with its environment through primary inputs and primary outputs. The performance optimization problem of pipelined circuits is to maximize the clocking rate or equivalently minimize the cycle time of the circuit. The aim of constrained min-area retiming is to constrain the number of registers for a target clock period, under the assumption that all registers have the same area, the min-area retiming problem reduces to seeking a solution with the minimum number of registers in the circuit. In the retiming problem, the objective function and constraints are linear, so linear programming techniques can be used to solve this problem. The basic version of retiming can be solved in polynomial time. The concept of retiming proposed by Leiserson et al. [15] was extended to peripheral retiming in [16] by introducing the concept of a "negative" register. These studies assume that the degree of functional pipelining has already been fixed and consider only the problem of adding pipeline buffers to improve performance of an asynchronous circuit.

The studies discussed are mainly targeted at the generation and optimization of hardware resources from behavioral RTL descriptions. As to our knowledge, there is no available tool that performs these functions at the level of a dataflow program. The recent development of the CAL dataflow language allows the application of these techniques at a higher abstraction level, thus provide the advantages of rapid design space exploration to explore pipeline throughput and area trade-off, and simpler transformation of a non-pipelined to a pipelined behavioral description, compared to low abstraction level RTL. The next section presents background on dataflow networks, high-level modeling for hardware synthesis, and the CAL actor language.

3 Dataflow modeling and high-level synthesis

Early studies on dataflow modeling are based on the Kahn process network introduced by Kahn in 1974 [18], which is a dataflow network with a local sequential process and global concurrent processes. This has been extended to graph models with a number of variants such as the directed acyclic graphs (DAG) [19-21] where each node represents an atomic operation, and edges represent data dependencies. The extension of the DAG is the synchronous dataflow graphs (SDF) [22] that annotates the number of tokens produced and consumed by the computation node, thus allowing feasible actor scheduling. Another type of dataflow graph is the control dataflow graphs (CDFG) [23] which describes static control flow of a program using the concept of a director that regulates how actors in the design *fire* and how tokens are used.

Several dataflow implementation methodologies have been proposed to use pre-configured IP blocks in a dataflow environment such as the PICO framework [24], simpleScalar [23], and the study of Lahiri et al. [25]. There exist also commercial tools to aid DSP hardware designs such as Cadence SPW [26], Altera DSP Builder [27] and Xilinx AccelDSP [28]. Some of these offer integration with Mathworks MATLAB and SIMULINK [29]. These methods, however, constraint the design to a given class of architecture and put restrictions on designers.

In contrast to block-based DSP, C language, on the other hand, offers higher flexibility. Synthesis from C to hardware has been a topic of intensive research with developments such as the Spark framework [30], GAUT tool of LABSTICC [31], and Catapult C from Mentor Graphics [32]. However, C program is designed to execute sequentially, and it still remains a difficult problem to generate efficient HDL codes from C, especially for DSP applications. Furthermore, C programs are also difficult to analyze and identify for potential parallelism because of the lack of concurrency and the concept of time [33]. In the context of RTL, SystemC was introduced but mainly restricted to system level simulations and offered limited support for hardware synthesis. Transaction level modeling raises the abstraction level one step above systemC, and has gained popularity, but the level of abstraction remains quite low for effective designs.

High-level synthesis methodologies have also been used to generate pipeline schedules in RTL, for example in [34], where a variation of the Modulo scheduling algorithm has been used to exploit loop-parallelism by means of executing operations from consecutive iterations of a loop in parallel. The technique is applied on the level of an assembly language for generating pipelined RTL descriptions. However, besides the limitation of the technique on loop algorithms, the level of the input description is sequential and again, faces the analyzability problem for effective pipelining. The study reported an improvement of up to 35% between pipelined and non-pipelined implementations.

In order to overcome these issues in the state of the art of high-level modeling and synthesis, the Ptolemy project at the University of California-Berkeley led to the development of the CAL dataflow language based on the concept of *actors*.

3.1 Actor-based dataflow modeling

Actors were first introduced in [35] as means of modeling distributed knowledge-based algorithms. Actors have

since then become widely used [1-4,36-41], especially in embedded systems, where actor-oriented design is a natural match to the heterogeneous and concurrent nature of such systems.

Many embedded systems have significant parts that are best conceptualized as dataflow systems, in which actors execute and communicate by sending each other packets of data. It is often useful to abstract a system as a structure of cooperating actors. Many such systems are dataflow-oriented, i.e. they consist of components whose ability to perform computation depends on the availability of sufficient input data. Typical signal processing systems, and also many control systems fall into this category.

Component-based design is an approach to software and system engineering, in which new software designs are created by combining pre-existing software components. Actor-oriented modeling is an approach to systems design, where entities called actors communicate with each other through ports and communication channels. From the point of view of component-based design, actors are the components in actor-oriented modeling.

Figure 2 shows a simple dataflow network. Several actors are composed into a network, a graph-like structure (often referred to as a model) in which output ports of actors are connected (typically with FIFO buffers) to input ports of the same or other actors, indicating that tokens produced at those output ports are to be sent to the corresponding input ports. Such actor networks are of course essential to the construction of complex systems. The encapsulation of each actor means that they are treated as a separate entity that works independently, but concurrently in a network. Increasing the number of actors in the network implies more concurrent operations; which is analogous to pipelining.

3.2 CAL dataflow language

CAL is a domain-specific language for writing dataflow actors, with the final language specification released at the end of 2003 [36]. The language describes an algorithm using an encapsulated actor, which communicates with another actor by passing data *tokens*. An actor then performs its algorithm specified in its *action* if there is token available and if it is enabled by one or more of the following: *guard, priority*, and *scheduling* conditions. If an action is performed, it is said to be *fired*, which consumes the input token, modify its internal states (variables, guard, schedule) and produces an output token which can be passed to another actor, itself or the system output [2]. An example of a CAL actor is given in Section 4.

CAL, however, is not a general purpose or full-fledged programming language; one of its key goals is to make actor programming easier by providing a concise high-level description with explicit dataflow keywords, unlike traditional programming languages. It is also designed to be platform independent and retargetable to a rich variety of target platforms, for example single-core and multi-core CPUs [1,36,41], FPGAs [1,37,39], and ASICs [38]. CAL provides a strict semantics for defining actor computational operations, ports and parameters and its composite data structures. But it leaves certain issues to the embedding environment, such as the choice of supported data types and the definition of the target semantics.

3.3 CAL to HDL synthesis

The synthesis of CAL program to HDL is one of the core components of the CAL dataflow language. It was

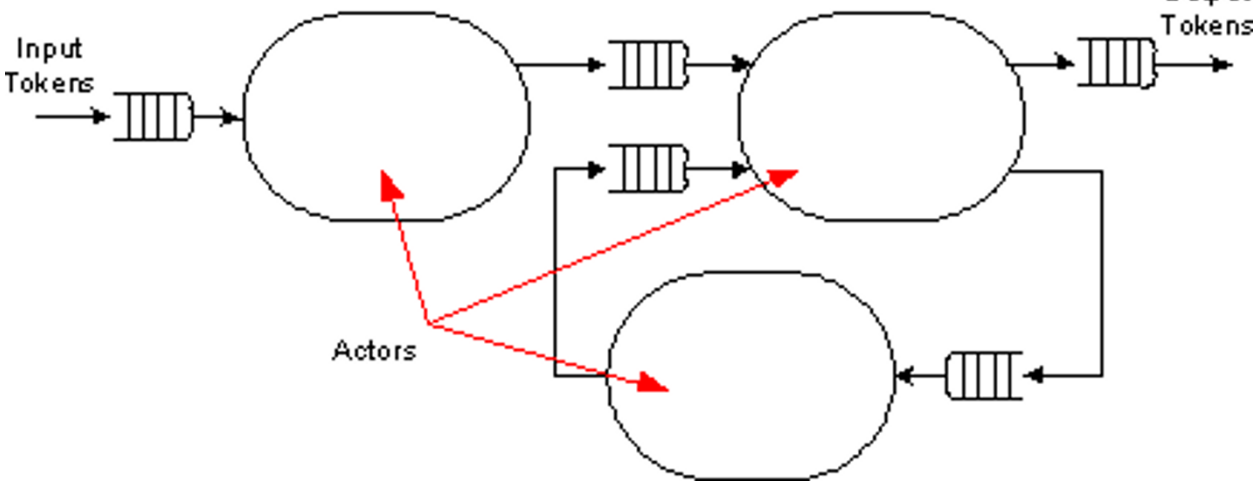


Figure 2 Dataflow network with actors, tokens, and buffers.

pioneered by Xilinx Inc. and now available as Eclipse IDE opensource plugins called OpenDF and OpenForge [3]. The CAL to HDL code generator is essentially an XML processing and transformation engine using Java. The two main steps are:

1. *Generation of top level VHDL from a flattened* CAL*dataflow network*. The tool takes in a flattened CAL network called *XDF*, and transforms it into a top-level VHDL file. Some of the operations include port evaluation, data width, fanout, and buffer size annotation, and instance name addition.
2. *Generation of Verilog files for each* CAL*actor*. CAL actors are first checked syntactically, and then parsed into various XML representations that include several basic optimization steps. The final XML representation is called SLIM, which is a representation in a single-static assignment (SSA^b) form. SLIM file is then loaded into a Java *Design* class that represents top-level hardware implementation. The Java object representing the actor is optimized for hardware which includes operator constant rule, loop unrolling, variable re-sizer, memory reducer, splitter and trimmer. Next, a hardware scheduler is also generated based on the specification in the SLIM representation. Finally, a completed design object for an actor is written as a Verilog file.

HDL code generation from CAL actors has proven to generate efficient hardware. As reported in [37] for the hardware implementation of MPEG-4 Simple Profile Decoder, CAL design results in less coding, smaller implementation area, and higher throughput compared to classical RTL methodology.

The strength of the CAL dataflow language, especially for parallel DSP application, and its HDL synthesis makes it interesting for further optimization. As described, the CAL to HDL synthesis tool optimizes and generates code for each actor; no study has been done on actor partitioning for pipelining, which is the focus of this article.

4 Mathematical modeling of pipeline synthesis and optimization

In order to clearly present our mathematical formulation of the pipeline synthesis and optimization, the theoretical model will be complemented with a simple example–the YCrCb to RGB converter actor. A brief introduction to this actor will be given first.

4.1 The YCrCb to RGB conversion actor

Figure 3 shows a CAL description of a 30-bit YCrCb to 24-bit RGB, based on Xilinx XAPP930 [42]. It is typically used in high quality down-sampling and decoding of color spaces. The actor contains a single action that

```
actor YCrCbtoRGB()
 int(size=10) Y, int(size=10) Cr, int(size=10) Cb ⇒
 int(size=8) R, int(size=8) G, int(size=8) B :

 int(size=13) rv = 292;
 int(size=13) gu = 101;
 int(size=13) gv = 149;
 int(size=13) bu = 520;
 int(size=11) t1 := 1023;

 action
  Y:[y], Cr:[cr], Cb:[cb] ⇒
  R:[r], G:[g], B:[b]
 var
  int(size=10) r, int(size=10) g, int(size=10) b, int(size=10)rt,
  int(size=10) gt, int(size=10)bt, int(size=11) yt, int(size=11)crt,
  int(size=11) cbt
 do
  //signed to unsigned representation
  yt := bitand(y, t1);
  crt := bitand(cr, t1);
  cbt := bitand(cb, t1);
  //core algorithm
  rt := (((yt-64) << 8) + rv*(crt-512)) >> 10;
  gt := (((yt-64) << 8) - gu*(cbt-512) - gv*(crt-512)) >> 10;
  bt := (((yt-64) << 8) + bu*(cbt-512)) >> 10;
  //clip output r
  if (rt > 0) then
   if (rt < 255) then r := rt;
   else r := 255; end
  else
   r := 0;
  end
  //clip output g
  if (gt > 0) then
   if (gt < 255) then g := gt;
   else g := 255; end
  else
   g := 0;
  end
  //clip output b
  if (bt > 0) then
   if (bt < 255) then b := bt;
   else b := 255; end
  else
   b := 0;
  end
 end
end
```

Figure 3 CAL actor example–actor *YCrCbtoRGB*.

first converts 10-bit inputs into an explicit 11-bit unsigned representation using the *bitand* operation. Following this, the core algorithm is performed using 11 adders/subtractors, 4 multipliers, and 6 shifters. Finally, the RGB output has to be clipped if the result exceeds the 8-bit per output dynamic range. This utilizes six *if* statements with comparators.

The general idea in our pipeline synthesis is to partition this relatively large action into several actions in separate actors. The first step is to make the action body (i.e. operations) more analyzable. This is achieved by limiting each arithmetic operator to two operands, and assigning a unique output variable for each operator, essentially transforming each operator to a two-operands-single-assignment form. The dataflow graph of this transformation is given in Figure 4. Twenty extra variables (*z1* to *z20*) are introduced to represent intermediate results of 35 operations.

The remainder of this section provides relations, graphs, and algorithms that define pipeline synthesis and optimization problem from a generic dataflow graph, with an example using the graph of Figure 4.

4.2 Dataflow graph relations

4.2.1 Operator precedence relation on dataflow graph

Let $N = \{1, ..., n\}$ be a set of algorithm operators and $M = \{1, ..., m\}$ be a set of algorithm variables. The following

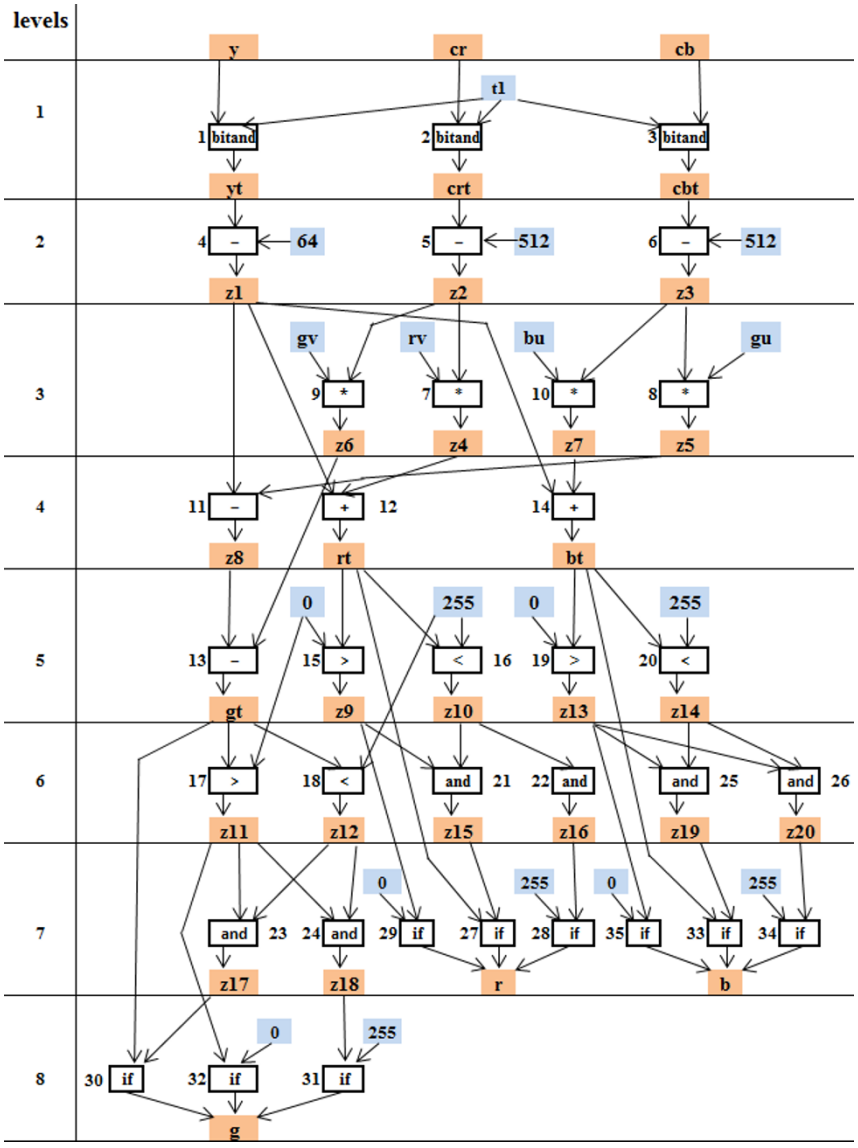


Figure 4 Dataflow graph of the action in the *YCrCbtoRGB* actor in the two-operands-single-assignment form.

matrices describe operator-variable and precedence relations.

1. The *operators/input variables relation*. The operators/input variables relation is described with the $F(n, m)$ matrix:

$$F = \begin{bmatrix} f_{1,1} & \cdots & f_{1,m} \\ \vdots & \ddots & \vdots \\ f_{n,1} & \cdots & f_{n,m} \end{bmatrix},$$

where $f_{i,j} \in \{0, 1\}$ for $i \in N$ and $j \in M$. If $f_{i,j} = 1$, then the j variable is an input for the i operator, otherwise it is not. In the CAL language, input tokens are considered as input variables of operators in all actions of one actor.

2. The *operators/output variables relation*. This relation describes which variables are outputs of the operators. It is represented with the $H(n, m)$ matrix:

$$H = \begin{bmatrix} h_{1,1} & \cdots & h_{1,m} \\ \vdots & \ddots & \vdots \\ h_{n,1} & \cdots & h_{n,m} \end{bmatrix},$$

where $h_{i,j} \in \{0, 1\}$ for $i \in N$ and $j \in M$. If $h_{i,j} = 1$, then the j variable is an output for the i operator,

otherwise it is not. In the CAL language, output tokens are considered as output variables of operators in all actions of one actor.

3. The *operator direct precedence relation*. This relation describes a partial order on the set of operators derived from analysis of the data dependencies between operators on the data flow graph. The relation is represented with the $P_{\text{direct}}(n, n)$ matrix:

$$P_{\text{direct}} = \begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{n,1} & \cdots & p_{n,n} \end{bmatrix},$$

where $p_{i,j} \in \{0, 1\}$ for $i, j \in N$. If $p_{i,j} = 1$, then the i operator is a direct predecessor for the j operator, otherwise it is not. Usually, this is due to the j operator that consumes a value produced by the i operator. For the single-assignment model of an acyclic algorithm, the direct precedence is defined over the F and H matrices as

$$P_{\text{direct}} = H \times F^t, \quad (1)$$

where $\times$ is matrix multiplication operation, and H^t is a transpose of the H matrix.

4. The *operator precedence relation*. The direct/indirect precedence P_{total} relation between operators can be inferred by applying the transitive closure operation to the $P_{\text{direct}}(n, n)$ matrix:

$$P_{\text{total}} = P_{\text{direct}} \cup P^2_{\text{direct}} \cup \cdots \cup P^i_{\text{direct}} \cup \cdots \cup P^n_{\text{direct}}, \quad (2)$$

where P^i_{direct} is P_{direct} in power of i. We will say that P_{direct} defines the *direct* precedence relation and P_{total} defines the precedence relation.

4.2.2 Estimation of operator delays

The operator delay depends on the method of implementation. Different implementations of the same operator give different parameters including time delay and area of the functional units that implement the operators.

In order to perform pipeline synthesis and optimization, relative time delay may be used. Table 1 shows relative time delay of an adder which is assumed to be 1.00. The delays of other operators are estimated compared to the delay of the adder. Thus, the delay of multiplication operator is estimated to be 3.00, and the delay of if-operator is estimated at 0.05.

Table 1 CAL operator relative delays

No.	CALoperator type	Time delay
1	+/-	1.00
3	*	3.00
4	>/<	0.10
6	bitand/bitor	0.02
8	not	0.01
11	if	0.05
12	other	...

It should be noted that operator relative delays have to be recalculated depending on the operand widths. For example, a 32-bit variable would use a 32-bit adder, which typically has a higher delay compared to an 8-bit variable that only uses an 8-bit adder. For more accurate results, operand widths have to be taken into account when estimating operator delays.

Another issue with operator delay estimation is the total delay on a path. The total delay along path L is usually estimated by

$$delay(L) = \sum_{i \in L} delay(i). \quad (3)$$

This simplification can imply significant inaccuracy in pipeline stage delay estimation. For example, if two addition operations i and j are executed sequentially, and each of them is implemented, for instance, by a ripple carry adder, the total delay satisfies the inequality as follows:

$$delay(i, j) < delay(i) + delay(j). \quad (4)$$

In order to increase the accuracy in the pipeline stage delay estimation, a more precise technique is required that takes into account the operation implementation methods. Furthermore, delay recalculation techniques have to be analyzed for various operators executed sequentially. Together with the delay recalculation based on operand widths, technique for evaluating accurate operator delays is an important part of the pipeline synthesis and optimization tool.

4.2.3 Variable and register widths

In CAL programming, the following objects are possible: constants, variables, input, and output. Their sizes expressed in the number of bits can be defined explicitly in the code. In the case, when a size is not defined, a default size of 32-bit is given.

Object widths are essential parameters during hardware synthesis. Extra bits may imply larger implementation area, larger delays, and reduced frequency. For this reason, the object widths must be defined with minimum possible size for a given algorithm and required accuracy of output values. The minimum sizes can be

estimated automatically by the synthesis tool or manually by the designer. The bus and register widths completely depend on the object widths. Minimization of the object widths minimizes the total register width in the pipeline under synthesis. For the YCrCb to RGB converter algorithm described in Figure 3, the object, width, and type are given in Table 2.

4.2.4 Longest path delays between operators on acyclic operator precedence graph

The longest path delays between operators constitute a basis for describing pipeline execution time constraints.

We introduce the G matrix that describes the maximum time delays (critical path lengths) between operators on the data flow graph that can be derived from the analysis of the data dependencies between operators and the operator execution times:

$$G = \begin{bmatrix} g_{1,1} & \cdots & g_{1,n} \\ \vdots & \ddots & \vdots \\ g_{n,1} & \cdots & g_{n,n} \end{bmatrix},$$

where $g_{i,j}$ at $i, j \in N$ is a real value. If $g_{i,j} = 0$, then there exists no path between i and j operators on the data flow graph, and the corresponding element of the P_{total} matrix is also equal to zero. If $g_{i,j} > 0$, then there is a path between the operators. The G matrix can be computed from the vector of operator delays and the P_{direct} matrix. An algorithm for evaluating longest and shortest path on directed cyclic and acyclic graphs are described in [43].

We present an alternative algorithm for computing the longest path length on DAG, based on the idea that at each step we take an operator for which the longest path lengths of all direct predecessors are evaluated and evaluate the longest path lengths between the taken operator and all its predecessors in two cases:

1. as a sum of delays of the taken operator and its direct predecessor;
2. as a sum of delay of the taken operator and the longest path length between its direct predecessor and the predecessors of the direct predecessor.

An example of the G matrix for the YCrCb to RGB converter is shown in Figure 5. It should be noted that the longest path between *variables* may also be used for pipeline synthesis and optimization, in which case a similar G matrix can be derived. The methodology in this article considers path length based on operators.

4.2.5 Operator conflict graph

For a given pipelined network, we say that T_{stage} is its stage time delay, which is the worst time delay of one pipeline stage. Among the pipeline stages, the operator longest path gives maximum stage delay. In the G matrix of the operator longest paths in the dataflow graph, the value $g_{i,j}$ must be less than or equals to T_{stage} in order for the i and j operators to be included in one stage. If the $g_{i,j}$ value is greater than the T_{stage}, then we say that there is a conflict between i and j, and the operators must be scheduled to different stages. Taking such pair of operators, we obtain the operator conflict relation for a given stage delay:

$$ConflictRelation = \{(i,j) | i,j \in N \text{ and } g(i,j) > T_{\text{stage}}\} \quad (5)$$

The operator conflict relation satisfies the requirement as follows:

$$ConflictRelation \subseteq PrecedenceRelation \quad (6)$$

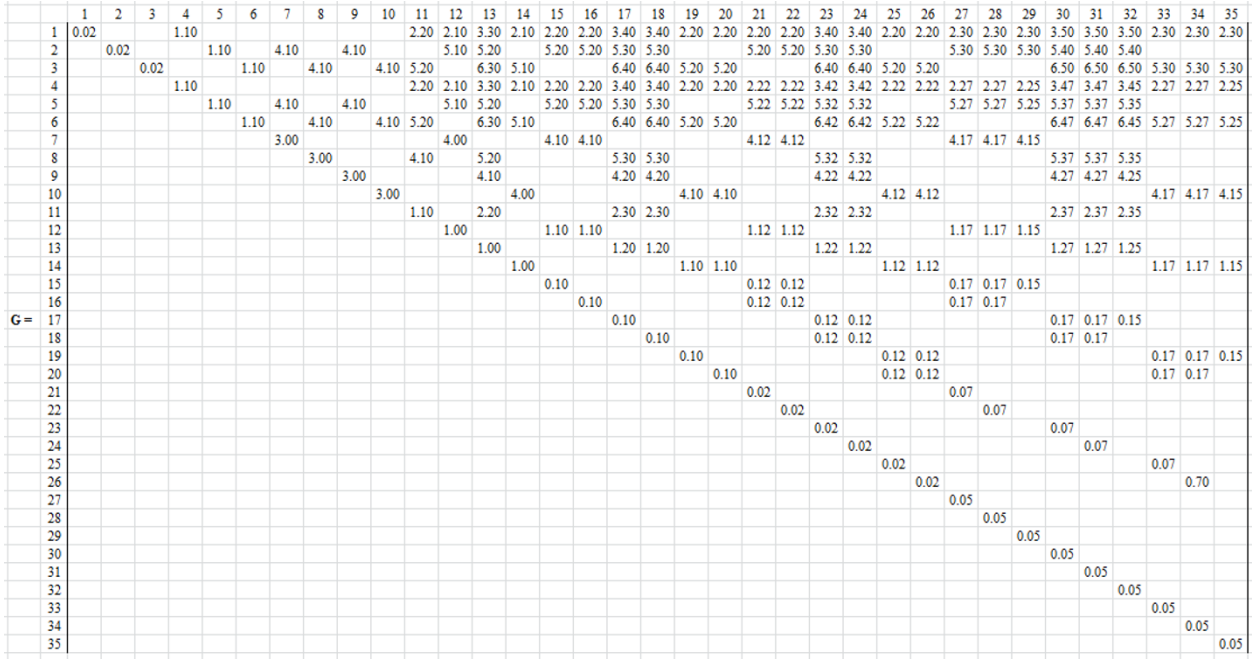
It is obvious that if T_{stage} is larger than the length of the longest path in the algorithm, then *ConflictRelation* = ⊠. If the inequality $delay(i) + delay(j) > T_{\text{stage}}$ holds for any two adjacent operators i and j on the dataflow graph, then *ConflictRelation* = *PrecedenceRelation*. Therefore, the *ConflictRelation* essentially depends on the value of T_{stage}. By varying the value of T_{stage} we can generate different pipelines for the same dataflow graph description.

The *ConflictRelation* represents operator conflict directed graph by means of interpreting the pairs *(i, j)* of operators included in the relation as the graph edges. It should be noted that the conflict graph configuration and the accuracy of the final pipeline synthesis results essentially depend on the accuracy of the operator relative time delay estimation.

Similar to the G matrix, *variable* conflict matrix and graph can also be obtained and used for pipeline synthesis and optimization.

Table 2 Object width and type in the YCrCb to RGB converter algorithm

Object	Width	Type
rv, gu, gv, bu	13	Constant
t1	10	Constant
y, cr, cb	10	Input
r, g, b	8	Output
rt, gt, bt, z2, z3	10	Variable
yt, crt, cbt	11	Variable
z1, z4, z7, z8	19	Variable
z5	17	Variable
z6	18	Variable
z9, z10, z11, z12, z13, z14, z15, z16, z17, z18, z19, z20	1	Variable



G =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
1	0.02			1.10							2.20	2.10	3.30	2.10	2.20	2.20	3.40	3.40	2.20	2.20	2.20	2.20	3.40	3.40	2.20	2.20	2.30	2.30	2.30	3.50	3.50	3.50	2.30	2.30	2.30
2		0.02			1.10		4.10		4.10			5.10	5.20		5.20	5.20	5.30	5.30			5.20	5.20	5.30	5.30			5.30	5.30	5.30	5.40	5.40	5.40			
3			0.02			1.10		4.10		4.10	5.20		6.30	5.10			6.40	6.40	5.20	5.20			6.40	6.40	5.20	5.20				6.50	6.50	6.50	5.30	5.30	5.30
4				1.10							2.20	2.10	3.30	2.10	2.20	2.20	3.40	3.40	2.20	2.20	2.22	2.22	3.42	3.42	2.22	2.22	2.27	2.27	2.25	3.47	3.47	3.45	2.27	2.27	2.25
5					1.10		4.10		4.10			5.10	5.20		5.20	5.20	5.30	5.30			5.22	5.22	5.32	5.32			5.27	5.27	5.25	5.37	5.37	5.35			
6						1.10		4.10		4.10	5.20		6.30	5.10			6.40	6.40	5.20	5.20			6.42	6.42	5.22	5.22				6.47	6.47	6.45	5.27	5.27	5.25
7							3.00					4.00			4.10	4.10					4.12	4.12					4.17	4.17	4.15						
8								3.00			4.10		5.20				5.30	5.30					5.32	5.32						5.37	5.37	5.35			
9									3.00				4.10				4.20	4.20					4.22	4.22						4.27	4.27	4.25			
10										3.00				4.00					4.10	4.10					4.12	4.12							4.17	4.17	4.15
11											1.10		2.20				2.30	2.30					2.32	2.32						2.37	2.37	2.35			
12												1.00			1.10	1.10					1.12	1.12					1.17	1.17	1.15						
13													1.00				1.20	1.20					1.22	1.22						1.27	1.27	1.25			
14														1.00					1.10	1.10					1.12	1.12							1.17	1.17	1.15
15															0.10						0.12	0.12					0.17	0.17	0.15						
16																0.10					0.12	0.12					0.17	0.17							
17																	0.10						0.12	0.12						0.17	0.17	0.15			
18																		0.10					0.12	0.12						0.17	0.17				
19																			0.10						0.12	0.12							0.17	0.17	0.15
20																				0.10					0.12	0.12							0.17	0.17	
21																					0.02						0.07								
22																						0.02						0.07							
23																							0.02							0.07					
24																								0.02							0.07				
25																									0.02								0.07		
26																										0.02								0.70	
27																											0.05								
28																												0.05							
29																													0.05						
30																														0.05					
31																															0.05				
32																																0.05			
33																																	0.05		
34																																		0.05	
35																																			0.05

Figure 5 Longest operator path lengths of the YCrCb to RGB converter.

4.2.6 Operator nonconflict graph

By means of subtraction of the *ConflictRelation* from the *PrecedenceRelation*, we obtain a so-called nonconflict operator relation:

$$NonConflictRelation = PrecedenceRelation \backslash ConflictRelation \quad (7)$$

In the relation, a pair *(i, j)* of operators does not constitute a conflict because the operators may be included in the same pipeline stage. For the operators, it is possible that *stage*(*i*) <*stage*(*j*), but it is not possible that *stage* (*i*) >*stage*(*j*). The NonConflictRelation varies in the range

$$\emptyset \subseteq NonConflictRelation \subseteq PrecedenceRelation \quad (8)$$

When *ConflictRelation* is empty then *NonConflictRelation* equals *PrecedenceRelation*. When *ConflictRelation* is equal to *PrecedenceRelation* then *NonConflictRelation* is empty.

4.2.7 As soon as possible (ASAP) and as late as possible (ALAP) scheduling

ASAP and ALAP are well-known scheduling techniques that schedule operations in a dataflow graph based on the earliest and latest possible sequence [43]. In this study, we use N set of operators and the *ConflictRelation* to generate an ASAP (and ALAP) scheduling that gives the earliest (and latest) stage that each operator can be scheduled. Tables 3 and 4 show ASAP and ALAP scheduling results for the YCrCb to RGB converter example for T_{stage} = 4.12.

4.2.8 Mobility-based operator ordering

The ASAP and ALAP results give crucial information on the mobility of an operator, which is defined as its possibility to be scheduled to various pipeline stages. We call the earliest stage that an operator *i* may be scheduled as *asap (i)*, and the latest as *alap(i)*. Hence, the mobility of operator *i* is given by *alap(i)-asap(i)*. If an operator may be scheduled to only one stage, then the mobility equals to zero. Table 5 shows the mobility of each operator for the YCrCb to RGB converter example for T_{stage} = 4.12. The two non-zero mobility operators, 1 and 4, imply that they can be moved to either pipeline stage-1 or stage-2. The optimization problem is then to determine which of the solutions give optimal results. The next section formulates the optimization problem.

4.3 Pipeline optimization tasks

Let N = {1, ..., n} be a set of algorithm operators and K = {1, ..., k} be a set of pipeline stages. The number of

Table 3 ASAP schedule for the YCrCb to RGB converter for T_{stage} = 4.12

Stage	Operators
1	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2	11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35

Table 4 ALAP schedule for the YCrCb to RGB converter for T_{stage} = 4.12

Stage	Operators
1	2, 3, 5, 6, 7, 8, 9, 10
2	1, 4, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35

pipeline stages is determined by the stage time delay T_{stage}. Variations in the stage delay imply variations in the pipeline stage count. We describe the distribution of operators onto pipeline stages with the X matrix:

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{k,1} & \cdots & x_{k,n} \end{bmatrix}.$$

In the matrix, the number of rows is equal to the number k of pipeline stages, and the number of columns is equal to the number n of operators. A $x_{i,j} \in \{0, 1\}$ variable for $i \in N$ and $j \in K$ takes one of two possible values. If $x_{i,j} = 1$, then the i operator is scheduled to the j stage, otherwise it is not scheduled to the stage. The X matrix describes a distribution of the operators on the stages.

In some cases, the $x_{i,j}$ variable can be determined in advance. For example, if $1 \le i < asap(j)$, then $x_{i,j} = 0$. Similarly, $x_{i,j} = 0$ for $alap(j) < i \le n$. If $i = asap(j) = alap(j)$, then $x_{i,j} = 1$. In order to develop efficient synthesis and optimization techniques, we replace the variables with their known values in the X matrix. The rest of the unassigned variables may be replaced with values 0 or 1 in such a way as to obtain a valid X matrix. One X matrix describes one possible pipeline schedule. The upper bound S^{upper} of the total number of X matrix can be estimated as

$$S^{\text{upper}} = \prod_{j \in N} \mu(j), \tag{9}$$

where $\mu(j)$ is the number of variables with unknown values in the j column of the X matrix.

For the YCrCb to RGB converter example with T_{stage} = 4.12, the asap and alap pipeline stages computed on the operator conflict graph are shown in Figure 6. Operators 1 and 4 may be scheduled to both first and second stages. The other operators are scheduled either to the first stage or to the second stage. The corresponding X matrix is presented in Figure 7. Four elements of the matrix are variables (denoted by x), the other elements are constants. The upper bound on the total number of X matrix (pipelined schedules) is $S^{\text{upper}} = 2^2 = 4$. However, actual number of schedules could be less than the upper bound since there are strong dependencies among the values of the matrix variables.

4.3.1 Objective function in the optimization task

For a given T_{stage} requirement, we can obtain several pipeline schedules. Different schedules give Different parameters. The most important is the number and total width of registers inserted in between neighboring pipeline stages. Minimization of the total register width will save the implementation area. Furthermore, the operating frequency could also possibly be increased with minimization of pipeline registers.

Figure 8 illustrates register usage from pipelining for an example of a 4-stage pipeline. Between the same stage, no registers are used since a particular stage circuit logic is purely combinatorial (indicated by W). Between stage k and $k+1$, registers are required if an output of an operation in stage k is used in the following $k+1$ stage (indicated by R). If the output of stage k is used by stage $k+2$ and beyond, then transmission registers are required (indicated by T). Our goal is to find the minimum total R and T registers from all possible schedules for a given T_{stage} constraint.

Let Ω be a set of possible X matrix. For the single-assignment model of the source algorithm, the objective function as follows minimizes the total pipeline register width over all elements of set Ω:

$$\min_{X \in \Omega} \sum_{s=1}^{k} \left\{ \sum_{j=1}^{m} [\max_{i \in N}(f_{i,j} \times x_{s,i}) - \max_{i \in N}(h_{i,j} \times x_{s,i})] \times width(j) + \sum_{j=1}^{m} [\max(\tau_j, \max_{e=s+1,\ldots,k, i \in N}(f_{i,j} \times x_{e,i})) - \max_{e=s,\ldots,k, i \in N}(h_{i,j} \times x_{e,i})] \times width(j) \right\}, \tag{10}$$

where $\tau_j = 1$ if the j variable is an output token and $\tau_j = 0$ otherwise; $\times$ is the arithmetic multiplication operation.

There are two parts in Equation 10. The first one estimates for each stage s the width of registers inserted in between the stage and the previous neighboring stage. The second one estimates for each stage the width of transmission registers.

Table 5 Operator mobility for the YCrCb to RGB converter for T_{stage} = 4.12

Mobility	Operators
0	2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35
1	1, 4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
asap	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
alap	2	1	1	2	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Figure 6 ASAP and ALAP pipeline stages for the scheduled operators for the YCrCb to RGB converter example with T_{stage} = 4.12.

4.3.2 Optimization task constraints

There are three constraints related to our optimization tasks–operator scheduling, time, and precedence constraints.

The operator scheduling constraint describes the requirement that each operator should belong to only one pipeline stage:

$$\sum_{s=asap(i)}^{alap(i)} x_{s,i} = 1 \quad \text{for } i \in N, \tag{11}$$

where s is a pipeline stage from the range *asap(i)* to *alap(i)*.

The time constraint describes the requirement that the time delay between two operators i and j must not be larger than T_{stage} if the operators are scheduled to one pipeline stage s:

$$x_{s,i} \times x_{s,j} \times g_{i,j} \leq T_{stage} \quad \text{for } i, j \in N \text{ and } s \in K, \tag{12}$$

where $g_{i,j}$ is the longest path between i and j operators on the algorithm dataflow graph. It is easy to see that if the operators are in the same stage and $x_{s,i} = x_{s,j} = 1$, then the inequality as follows must hold: $g_{i,j} \leq T_{stage}$. If the operators are not in the same stage, then the longest path length may be larger than the stage delay.

The operator precedence constraint describes the requirement that if the i operator is a predecessor of the j operator on a dataflow graph, then i must be scheduled to a stage whose number is not greater than the number of stage which j operator is scheduled to

$$\sum_{s=asap(i)}^{alap(i)} (s \times x_{s,i}) - \sum_{s=asap(j)}^{alap(j)} (s \times x_{s,j}) \leq 0 \text{ for } (i,j) \in PrecedenceRelation, \tag{13}$$

where $PrecedenceRelation \subseteq N \times N$ is described by the P_{total} matrix. Constraints 11, 12, and 13 together define the structure of the optimization space.

4.3.3 Operator conflict and nonconflict directed graphs coloring

The constraints formulated in the previous section describe the rules that must be followed to generate a valid pipeline schedule. For each pipeline schedule of a given T_{stage}, a *coloring* technique is used on the operator conflict and nonconflict graphs to assign an operator to a particular pipeline stage. Reference [43] explains the node coloring technique of an *undirected* graph $G(V, E)$, which colors the nodes such that no edge $(i, j) \in E$, $i, j \in V$ has two end-points with the same color. For any two adjacent nodes i and j, the inequality as follows holds: $color(i) \neq color(j)$. A chromatic number $\chi(G)$ of the undirected graph G is the minimum number of colors over all possible colorings.

However, since our conflict and nonconflict graphs are directed graphs, we introduce coloring on *directed* graphs using the following additional requirement: for directed edge $(i, j) \in E$ the inequality as follows should hold: $color(i) < color(j)$. In the pipeline optimization task, if the directed operator conflict graph has a chromatic number $\chi(G)$, then the pipeline can be constructed on $\chi(G)$ stages. We reduce the problem of purely directed graph chromatic number to the problem of longest directed path length in the operator conflict graph. This problem has polynomial complexity.

Node coloring of the YCrCb to RGB converter operator conflict graph is illustrated in Figure 9. The longest node path length equals to 2, therefore the graph chromatic number $\chi(G) = 2$. In this case, two colors are used for the two stages, light and dark colors. Note that nodes 1 and 4 are not colored since they can be colored with either color. However, in order to check which color combinations are valid, the nonconflict graph also needs to be analyzed and colored.

Compared to the operator conflict graph coloring, the operator nonconflict directed graph $G_n(V, E_n)$ is colored in a Different way. The inequality as follows must hold:

$$\max_{i \in \mu^{in}(d)} color(i) \leq color(d) \leq \min_{i \in \mu^{out}(d)} color(i), \tag{14}$$

where $d \in V$, $\mu^{in}(d)$ (or $\mu^{out}(d)$) is the set of adjacent nodes of d that are incident to *incoming* (or *outgoing*) edges of d. We may also color the nodes from range 1 to $\chi(G)$, where $\chi(G)$ is the chromatic number of the operator conflict graph. The only restriction in such

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
X =	stage-1	x	1	1	x	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	stage-2	x	0	0	x	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 7 Operator distribution matrix for the YCrCb to RGB converter example with T_{stage} = 4.12.

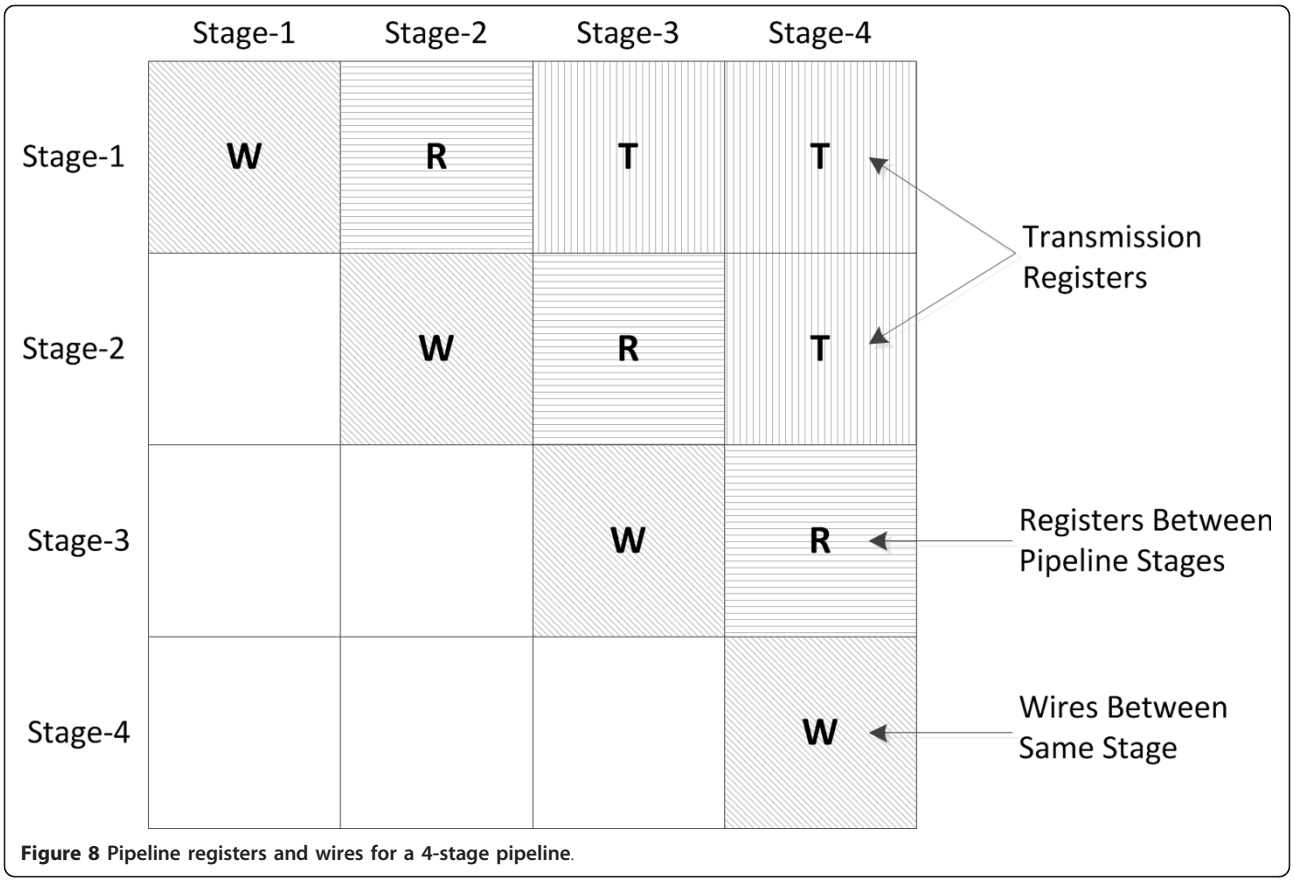


Figure 8 Pipeline registers and wires for a 4-stage pipeline.

coloring is that *color(i)* may not be larger than *color(j)* if $(i, j) \in E_n$. Moreover, the nonconflict graph enables coloring the nodes that are not colored in the conflict graph.

Going back to the example, we can now color nodes 1 and 4 with either one of the following: node 1 with light color and node 4 with light color; node 1 with light color and node 4 with dark color; node 1 with dark color and node 4 with dark color. Note that as revealed in the nonconflict graph in Figure 10, the coloring of node 1 with dark color and node 4 with light color is not valid.

5 Pipeline synthesis and optimization methodology and algorithms

This section presents methodology and key algorithms for our pipeline synthesis and optimization technique. Based on the formulations described in Section 4, a program was developed in Java under the Eclipse IDE that transforms a non-pipelined CAL actor into pipelined CAL actors.

The general overview is given in Figure 11. Starting from a non-pipelined CAL actor, the matrices *F, H,* P_{direct}, P_{total}, and *G* as well as the list $[T_{min}, ..., T_{max}]$ of the possible stage time T_{stage} values are computed. The T_{min} value equals the operator highest execution time, and the T_{max} value equals the longest path weight in actor dataflow graph. Optimization of pipelines is performed in a loop on various stage numbers. We start with one-stage pipeline ($K = 1$) and stage time $T_{stage} = T_{max}$. For the current T_{stage}, the conflict and nonconflict operator relations and directed graphs *Gc* and *Gnc* are generated from the *G* matrix and P_{total} relation. The chromatic number of the graphs is computed using a polynomial complexity algorithm. If the chromatic number is larger than the stage number *K*, then the successor value of T_{stage} is taken in the ascending list of stage time values. Owing to this, we use the lowest value of T_{stage} for each number *K* of stages and thus generate the fastest *K*-stage pipeline. If the chromatic number is larger than the stage number *K*, then the predecessor value of T_{stage} in the list is taken as its current value if $T_{stage} > T_{min}$, and *0* is taken otherwise. If for the updated value $T_{stage} < T_{min}$, then the optimization result is a set of pipelined networks of CAL actors for various stage numbers. Otherwise, the conflict and nonconflict graphs are generated again for an updated value of T_{stage}. In order to evaluate the operator mobility and to perform the critical path-based arrangement of graph colorings, the ASAP and ALAP schedules are generated. We propose ordered vertex coloring to order the

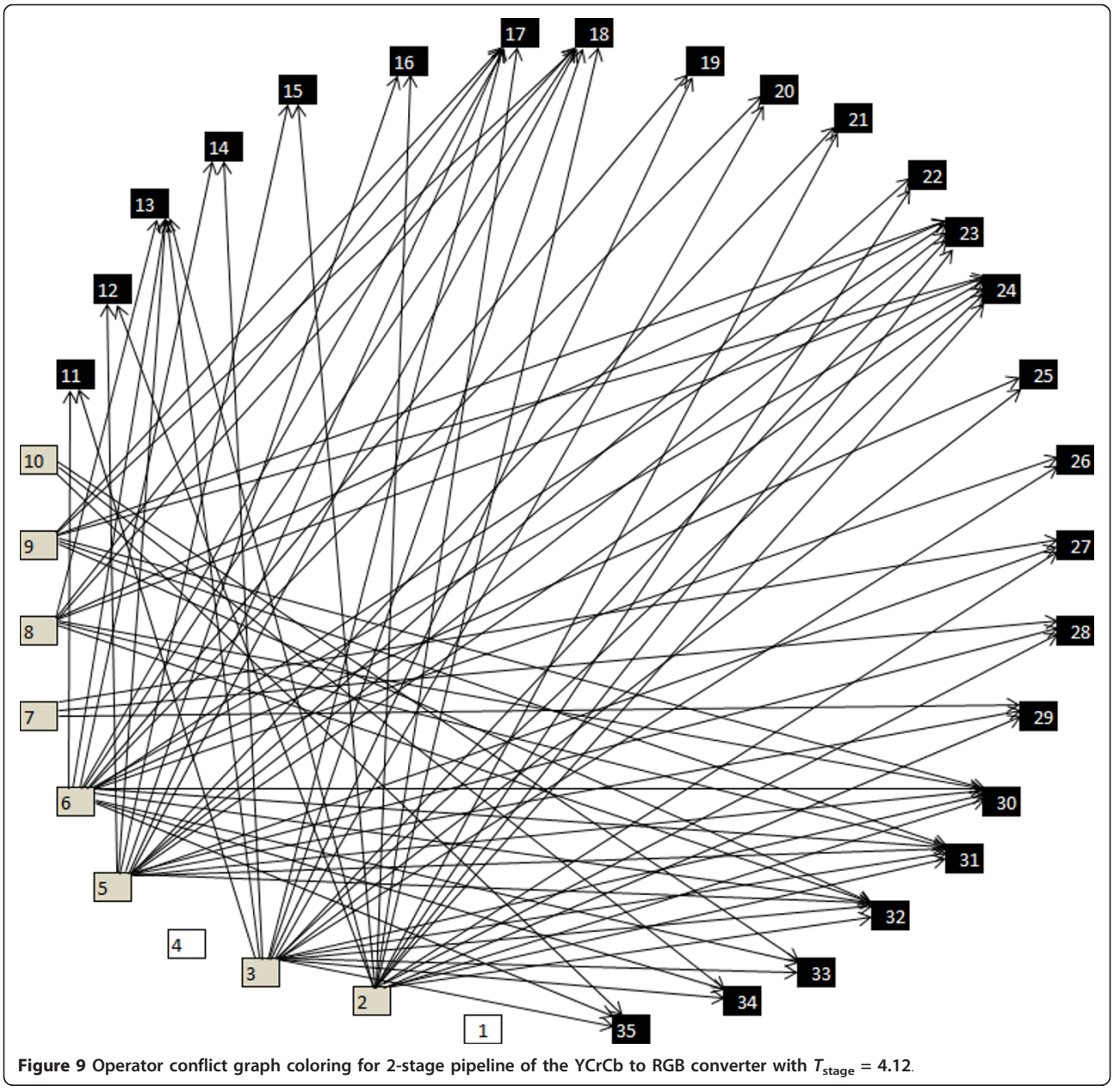


Figure 9 Operator conflict graph coloring for 2-stage pipeline of the YCrCb to RGB converter with T_{stage} = 4.12.

generation of solutions. The vertices in the critical (longest) paths are colored first. Owing to this approach, preferable solutions are generated first. Among them, the best (optimal or proximate) solution is selected using the pipeline register total width estimated with Equation 10. The best solution is generated with a branch and bound algorithm and finally used to generate pipelined CAL actors which are then synthesized to HDL for FPGA implementation.

In the remainder of this section, key algorithms for generating valid operator colorings on the conflict and nonconflict directed graphs and searching for an optimal pipeline schedule will be presented.

The technique for generating various operator colorings is based on recursive function and explicit stack mechanism. Figure 12 shows a top level recursive function Reg-WidthColoringStep which is used to generate pipeline schedules, and minimize the total pipeline register width. The algorithm takes in three inputs:

1. *asap*, which is an array of operators with the corresponding pipeline stage using the ASAP algorithm;
2. *alap*, which is an array of operators with the corresponding pipeline stage using the ALAP algorithm;
3. *order*, which is an array of operators ordered according to its mobility over pipeline stages;

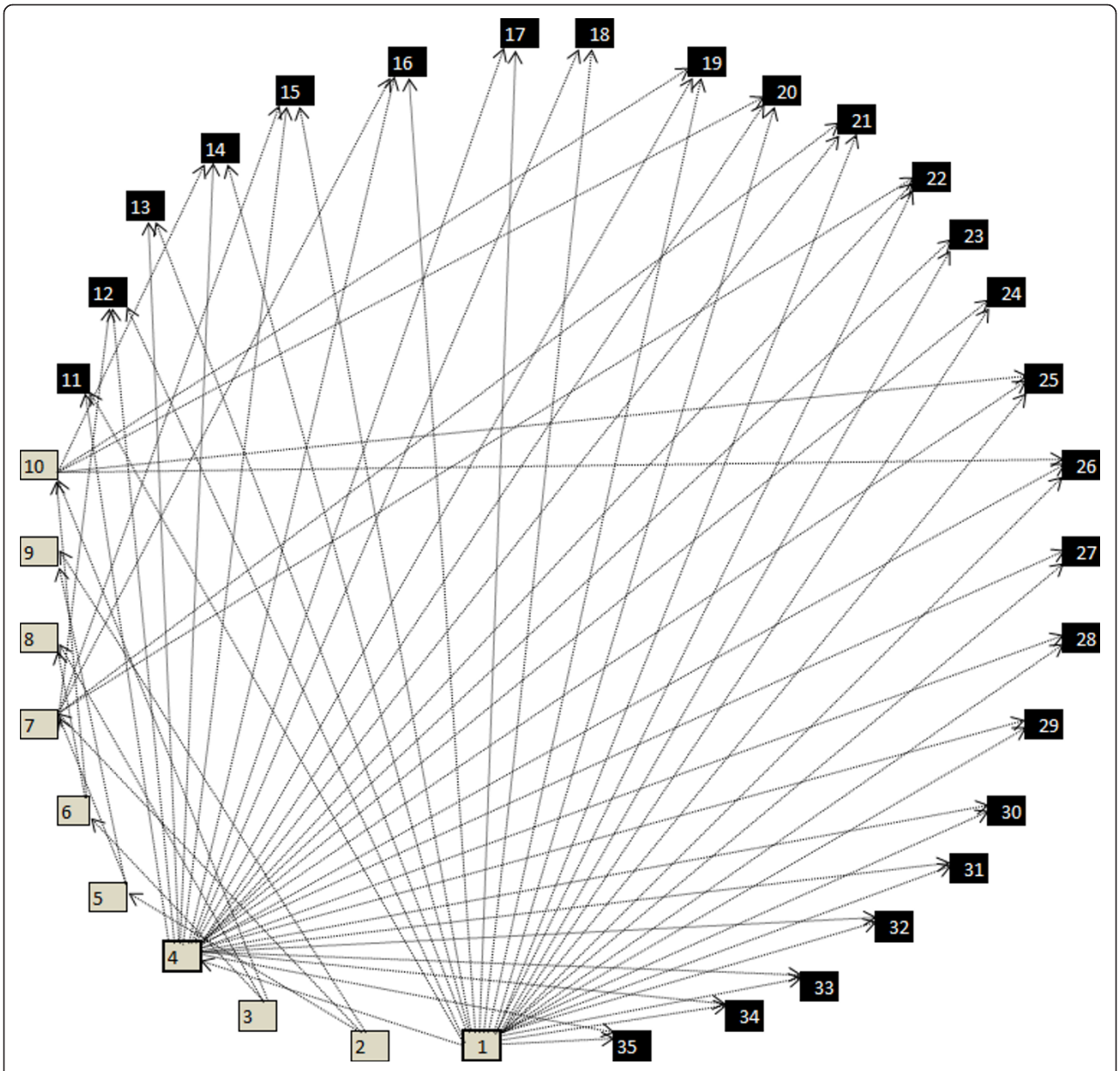


Figure 10 Operator nonconflict graph coloring for 2-stage pipeline of the YCrCb to RGB converter with T_{stage} = 4.12.

and generates the following output:

1. *pipelineCount*, which is the number of generated pipelines;
2. *optimalColor*, which is the optimal pipeline schedule as an array of operators with the corresponding pipeline stage;
3. *minRegWidth*, which is the minimum total register width of the optimal pipeline schedule.

The algorithm in Figure 12 works as follows. The recursive function takes in an input parameter *top*, which indicates the top record in the stack of operators. Depending on the *top* value, the function can return the control, generate the next complete coloring solution and compare it with the best current one, choose the next correct color of the current operator and generate the next record in the stack for procedure recursive call. In the next *top*+1 record, the minimum and maximum colors of the next operator are determined. If the minimum color is larger than the maximum color, then recoloring of the current operator is performed. The computations of minimum and maximum colors for operators are performed for both the conflict and the nonconflict graphs.

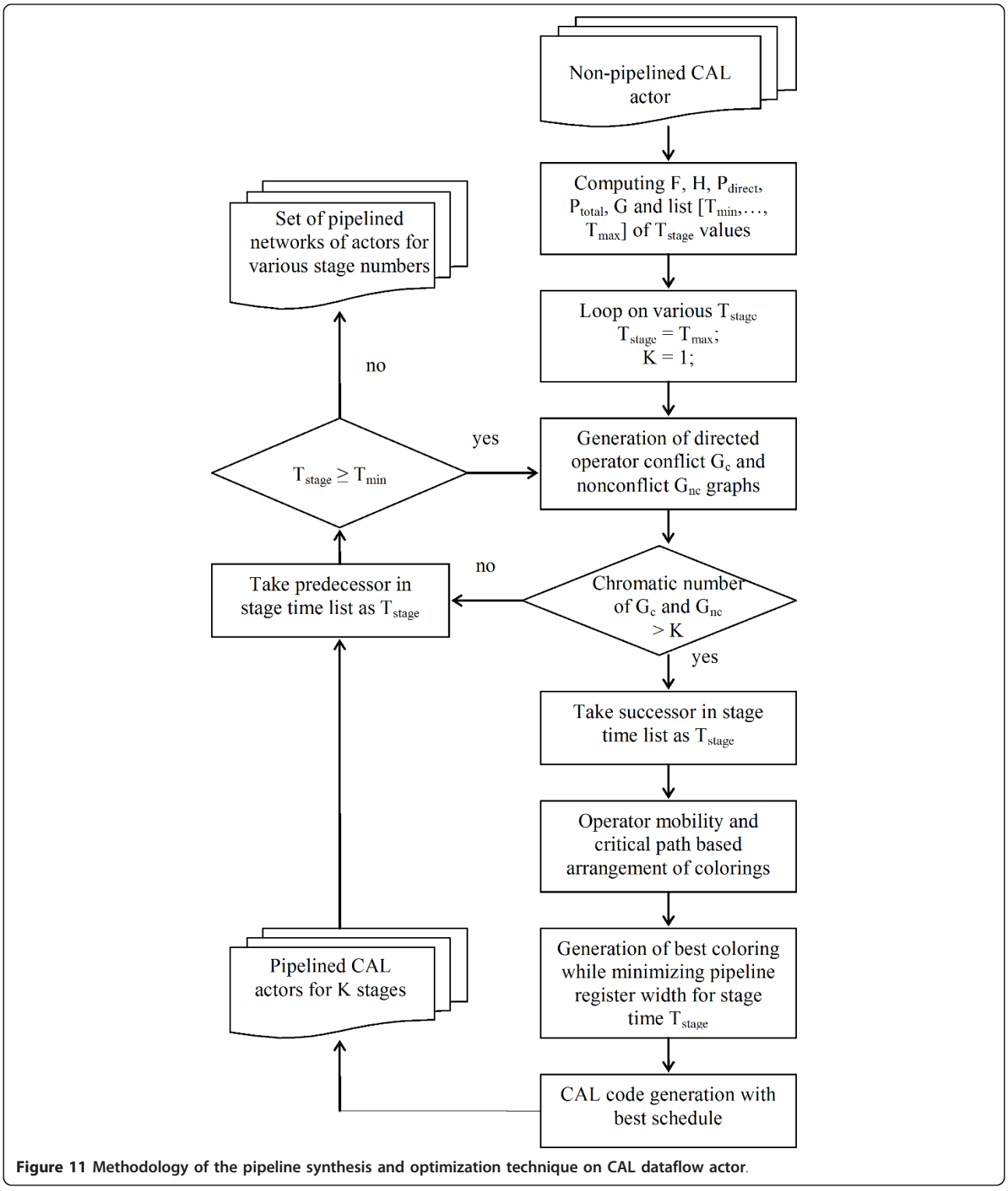


Figure 11 Methodology of the pipeline synthesis and optimization technique on CAL dataflow actor.

Figure 13 shows an algorithm to estimate minimum colors from a conflict graph. Among all operators that are recorded in the stack as predecessors and are in conflict relation with the given operator *op*, the operator with maximum color gives the value of *minC* that is returned by the algorithm as minimum color of *op* operator. The computations of maximum color from a conflict graph, minimum color from a nonconflict

graph, and maximum color from a nonconflict graph are performed in a similar way. Once all operators have been colored and a valid pipeline schedule is generated, the total register width is estimated to evaluate the efficiency of the schedule. The function *totalRegister-Width (colors)* performs this, which takes in a pipeline schedule, and returns the total register width. The function sums the width of all required pipeline and transmission registers of a pipeline schedule. From all possible pipeline schedules, the smallest total register width is stored in the variable *minRegWidth* with the corresponding *optimal-Colors* as the best schedule.

The final step is to generate CAL actors from the optimal coloring. This is done by taking the *optimalColors* array, partition the operators according to the scheduled stage, and print the required operations, variables, registers, inputs, and outputs declarations according to the syntax of the CAL dataflow language. The top level *XDF* network of pipelined CAL actors is also automatically generated based on the required number of pipeline stages.

It should be noted that our program is designed to generate potentially all possible valid pipeline schedules for a given T_{stage} constraint, therefore results in a global optimum solution. The number of possible schedules depends on the mobility of operators; an algorithm with many operators that can be moved among various stages would generate many possible schedules, therefore could potentially take a long time to find a global optimum. The *RegWidthColoringStep* function is a basic one for creating modifications which would restrict the number of generated solutions. Thus, it is modified to a branch-and-bound algorithm by means of introducing a *RegWidthLowerBound* function, which estimates a lower bound of total pipeline register width using partial operator coloring that is recorded in the stack, and ASAP and ALAP colorings. The number of generated solutions is also restricted with *MeetOptimizationTimeConstraint* function which takes into account the spent CPU time or the number of produced partial and complete colorings.

6 Experimental results

This section presents experimental results of our pipeline synthesis and optimization technique. Three video processing algorithms with relatively large combinatorial logic are selected for pipelining–they are the YCrCb to RGB converter, 8 × 8 1D IDCT, and Bayer filter. It is assumed that these algorithms constitute a critical path in a larger design, therefore, by pipelining these algorithms, a throughput increase can be obtained for the overall system.

Each design starts with an initial single CAL actor description, automatically pipelined using our tools to obtain multiple-actor description, and synthesized to HDL. For hardware implementation, two different 65-nm process node FPGAs have been used; Xilinx Virtex-5 and Altera Stratix III, synthesized using Xilinx XST and Altera Design Compiler tools, respectively.

6.1 YCrCb to RGB converter based on Xilinx XAPP930 [42]

This design was introduced in Section 3.2 for illustrating our methodology. A single actor was constructed that converts YCrCb to RGB color space. The total number of operators is 35.

The first step is to analyze valid T_{stage} constraints by determining the minimum and maximum T_{stage} from the dataflow graph (Figure 4). This is done by looking at Table 1 for estimating the delay of operators. From the dataflow graph, the minimum T_{stage} is defined by the multiplication operator which is equals to 3.00. The maximum T_{stage} is defined by the longest path length, given in Figure 5 which is 6.50. As a result, a T_{stage} constraint of 3.00 synthesizes to a 3-stage pipeline, while a stage delay of 6.50 and above gives a non-pipelined implementation. Further analysis of the dataflow graph shows dependency of the multiplication operators to the previous operations of bitand and subtraction. Therefore, a T_{stage} of 4.12 (bitand-subtract-multiply) is the minimum for which the pipeline would synthesize to 2-stages.

Figure 14 shows a graph of number of pipeline stages versus T_{stage} constraint. T_{stage} specification of between 3.00 and 4.12 synthesizes to a 3-stage pipeline, between 4.12 and 6.50 to a 2-stage pipeline, and 6.50 and above gives a 1-stage pipeline (i.e. non-pipelined) to obtain best performance for a particular number of pipeline stages, the minimum T_{stage} should be selected.

The results for 2-stage and 3-stage pipelines are given in Table 6. For T_{stage} = 4.12 with a synthesis to 2-stage pipeline, the optimal schedule (best) results in total register width of 83, while in the worst case, total register width is 92. This results in 10.8% reduction in total register width. For T_{stage} = 3.00 with a synthesis to 3-stage pipeline, minimum total register width is 122 compared to 131 in the worst case, with a reduction of 7.4%. Note that reduction of register widths between best and worst case are relatively small because of the limited optimization space for this example, with just three for each pipeline stages.

All designs (best and worst cases for comparison) have been synthesized to HDL for FPGA implementation. Figures 15 and 16 show graphs of resource versus throughput for Virtex-5 and Stratix III FPGAs, respectively. For Virtex-5, a 2-stage and a 3-stage pipeline designs require roughly 3× and 3.5× more slices, respectively, compared to a non-pipelined design. Between the best and worst case pipelined implementations, the difference is less than 1% because of the sharing of slice

```
RegWidthColoringStep(top) begin
 if top >= n then
  completeColorings := completeColorings + 1;
  regWidth := totalRegisterWidth(ColorStack);
  if minRegWidth > regWidth then
   optimalColors := colorStack; minRegWidth := regWidth;
   if not MeetOptimizationTimeConstraint() then exit;
  end if
  return;
 end if
 for c in minColor(top) to maxColor(top) do
  colorStack(top) :=c;
  if RegWidthLowerBound(colorStack,asap,alap) >= minRegWidth then
   cutBranches := cutBranches + 1; continue;
  end if
  if top < n-1 then
   oper := order(top+1);
   minC := estimateMinConflictColor(ColorStack,top,oper,ConflictRelation);
   maxC := estimateMaxConflictColor(ColorStack,top,oper,ConflictRelation);
   minP := estimateMinNonConflictColor(ColorStack,top,oper,
           NonConflictRelation);
   maxP := estimateMaxNonConflictColor(ColorStack,top,oper,
           NonConflictRelation);
   minColor(top+1) := maximum(asap(order(top+1)),minC+1,minP);
   maxColor(top+1) := minimum(alap(order(top+1)),maxC-1,maxP);
   if minColor(top+1) > maxColor(top+1) then continue; end if
  end if
  coloringStep(top+1);
 end for
end
```

Figure 12 The algorithm of register width minimization on set of operator colorings.

registers and LUTs. In terms of throughput, the optimized 2-stage and 3-stage pipelines are roughly 65% higher compared to a non-pipelined implementation. As for Stratix III, a slightly different result is observed. Similar to Virtex-5, there is a very little difference in resource between the best and the worst case pipelines. Compared to a non-pipeline implementation, 2-stages pipeline utilizes roughly 22% more ALUT, with 59% higher throughput. For the 3-stage pipeline, ALUT is increased by up to 44% with 57% more throughput.

In both FPGAs, it can be seen that the throughput is almost similar for 2-stages and 3-stages pipeline implementations. In other words, 3-stage pipeline does not result in significant increase in throughput compared to a 2-stage pipeline. The reason is due to the saturation of throughput, because at this point, the critical path is now in the control (i.e. registers) and hardware interconnection rather than the operators as in the non-pipelined implementation.

It should be noted that the ASAP and ALAP pipeline schedules can also be generated and compared. However, because of the small optimization space of this design, the ASAP pipeline schedule is found to be the same as the worst case schedule, and ALAP to be the same as the best case schedule. The next two examples present designs with significantly larger optimization space.

6.2 8 × 8 1D IDCT based on ISO/IEC 23002-2 [44]

The IDCT, or the Inverse Discrete Cosine Transform, is used in almost all image and video decompression

```
estimateMinConflictColor (ColorStack,top,op,ConflictRelation) begin
 minC := 0;
 for i in 0 to top do
  c := colorStack(i);
  nd := order(i);
  if (nd, op) is in ConflictRelation then
   if minC < c then minC := c; end if
  end if
 end for
 return minC;
end
```

Figure 13 The algorithm for estimating minimum color from conflict graph.

standard, for example in classical JPEG, MPEG-1, MPEG-2, MPEG-4, H.261, H.263, and JVT (H.26L) [45]. The reason for this is because of the strength of its inverse, the DCT, in which images are coded with inter-pixel redundancies, therefore offers excellent de-correlation for most natural images. The DCT also packs energy in the low frequency regions, which allows the removal of high frequency regions without significant quality degradation.

Image and video decompression systems use two-dimensional (2D) version of the IDCT, which is two one-dimensional (1D) IDCTs arranged serially with a transpose memory element in between. In the context of RTL, the two 1D IDCTs are normally treated as separate entities; therefore, the critical path is defined as the longest path of a 1D IDCT. For a parallel implementation of the 1D IDCT with large combinatorial logic, pipelining is an interesting strategy for improving data throughput.

Recently, the international standard organizations, ISO/IEC released the 23003-2 standard for coding and decoding MPEG video technology using fixed-point 8 × 8 IDCT and DCT. Among others, it provides approximation methods to ease implementation of codecs, ensure that the codecs are implemented in full conformance to specification, specifies single deterministic results as the output of an image or video encoding and decoding process, and improve the quality of delivered video and image representations.

Table 6 The YCrCb to RGB converter: exploration of pipeline optimization space

N_{stage}	2	3
T_{stage}	4.12	3.00
Reg-width best	83	122
Reg-width worst	92	131
Reg-width reduction (%)	10.8	7.4
Feasible schedules	3	3

Figure 17 shows the dataflow graph of the 23002-2 8 × 8 1D IDCT in a single-assignment form. The algorithm uses 25 subtractors and 19 adders, where we assumed the same delay of 1.00 for both the operators. It takes in eight inputs in parallel (i.e. one line of an 8 × 8 block), and produces eight parallel outputs. All variables are set to 26 bits, including input and output ports. The total number of variables is 52. The algorithms also use 21 shifters, which are not considered in the dataflow graph, since this element is considered to have no cost in the context of RTL.

The first step is to determine valid T_{stage} constraints by finding the largest single operator delay (minimum T_{stage}) and longest path length (maximum T_{stage}). Since the algorithm consists of only adders and subtractors, the minimum T_{stage} is found to be 1.00. The longest path length is found by analyzing the dataflow graph, which is 7.00. As shown in Figure 18, a T_{stage} = 1.00 synthesizes to a 7-stage pipeline, T_{stage} = 2.00 to a 4-stage pipeline, T_{stage} = 3.00 to a 3-stage pipeline, T_{stage}

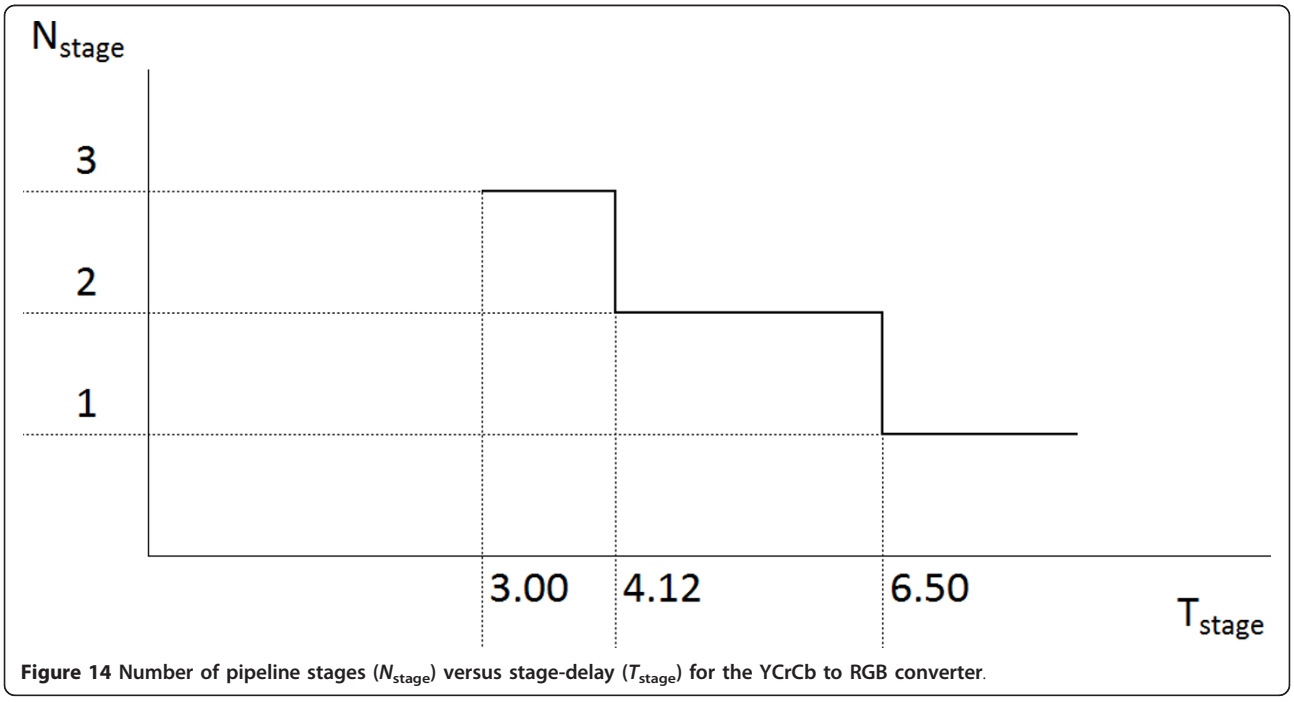


Figure 14 Number of pipeline stages (N_{stage}) versus stage-delay (T_{stage}) for the YCrCb to RGB converter.

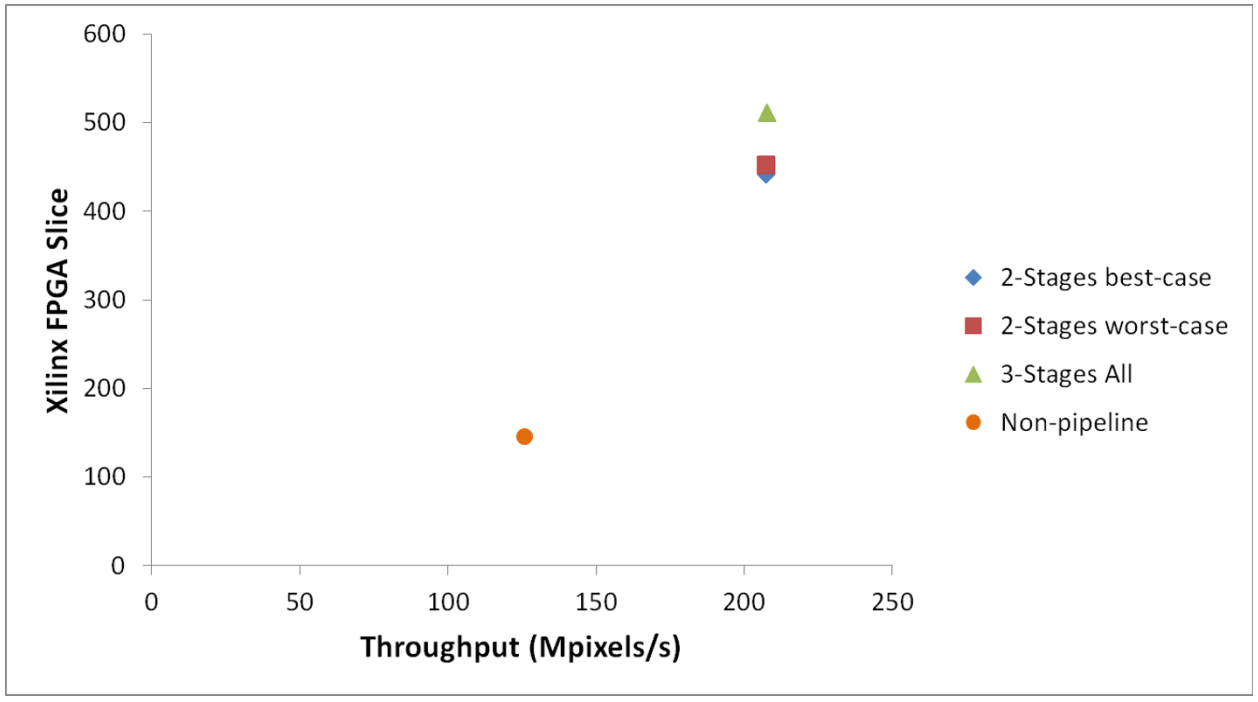


Figure 15 Slice versus throughput for all implementations of the YCrCb to RGB converter for Xilinx Virtex-5.

= 4.00 to a 2-stage pipeline, and $T_{\text{stage}} \geq 7$ to a non-pipelined implementation.

For each of the n-stage pipeline for $n = \{2, 3, 4, 7\}$, ASAP, ALAP, best, and worst schedules are generated. Table 7 summarizes the result. For a 2-stage pipeline of $T_{\text{stage}} = 4.00$, the highest total register width is the worst-case with 494, followed by ASAP with 364, ALAP with 312, and the best case with only 260. This results

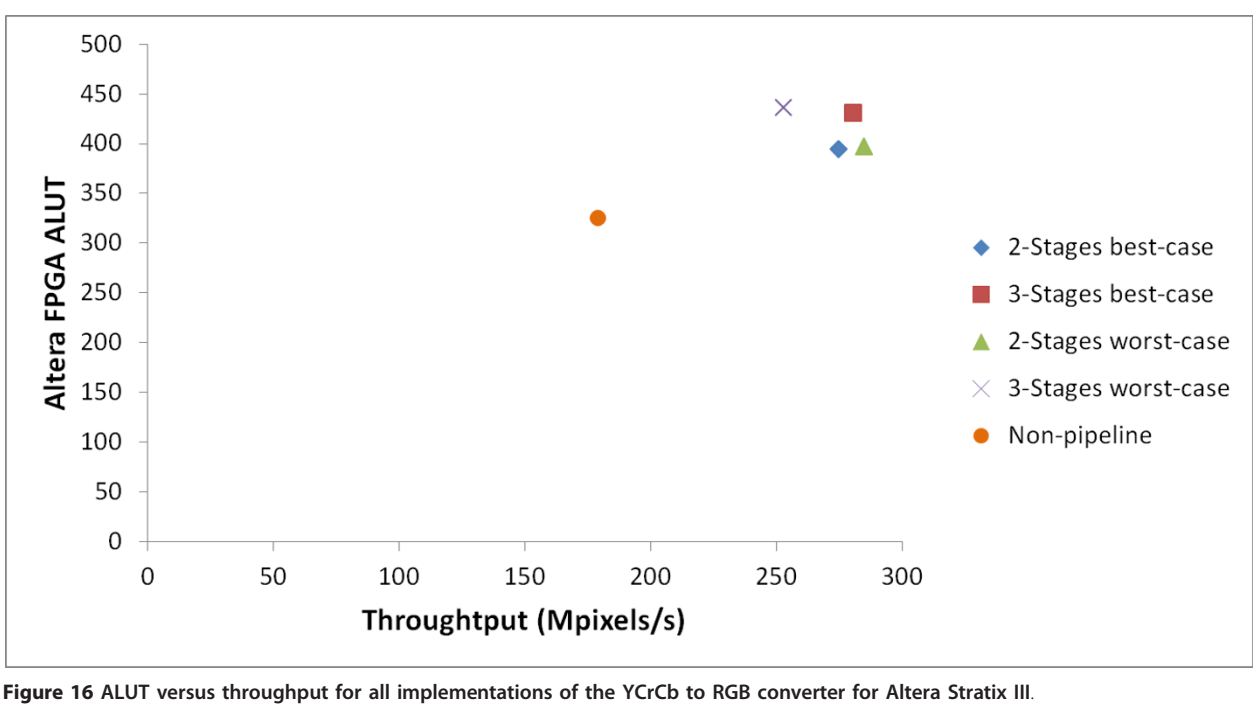


Figure 16 ALUT versus throughput for all implementations of the YCrCb to RGB converter for Altera Stratix III.

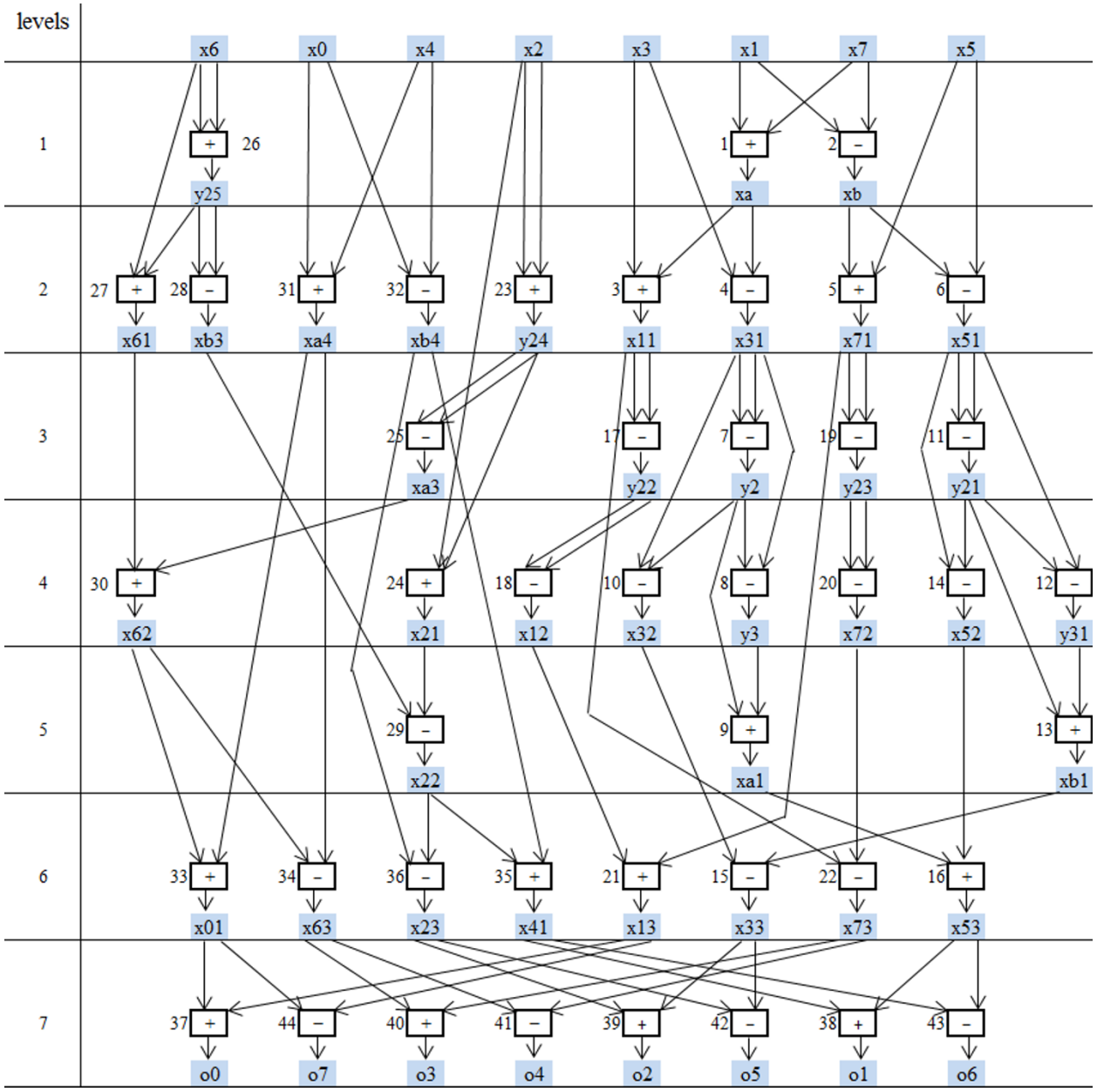


Figure 17 Dataflow graph of the 23002-2 8 ×8 1D IDCT in a single-assignment form.

in a register-width reduction of 90% compared to the worst-case. The optimization space for this pipeline configuration is 24, 336. For a 3-stage pipeline, register width reduction between best and worst cases is almost similar, with 88.9%. However, the optimization space is significantly more with 29, 555, 604 possible pipeline schedules. The 4-stage design shows the highest number of optimization space with more than 63 million schedules, with register width reduction of 43.8%. The smallest reduction is in the 7-stage pipeline with only 21.9%. This configuration also results in the most total register width with up to 2, 028 in the worst case. For this example, although the number of feasible schedules is large, our branch and bound algorithm generated only 5, 3, 1, and 1 complete colorings (schedules) for 2, 3, 4, and 7 pipeline stages, respectively, and cut all other branches in the search tree.

All designs have been synthesized to HDL, and then to Xilinx Virtex-5 and Altera Stratix III FPGAs for implementation. The results are shown in Figures 19 and 20. For Virtex-5, non-pipeline implementation results in 1, 650 total slice with throughput of 764

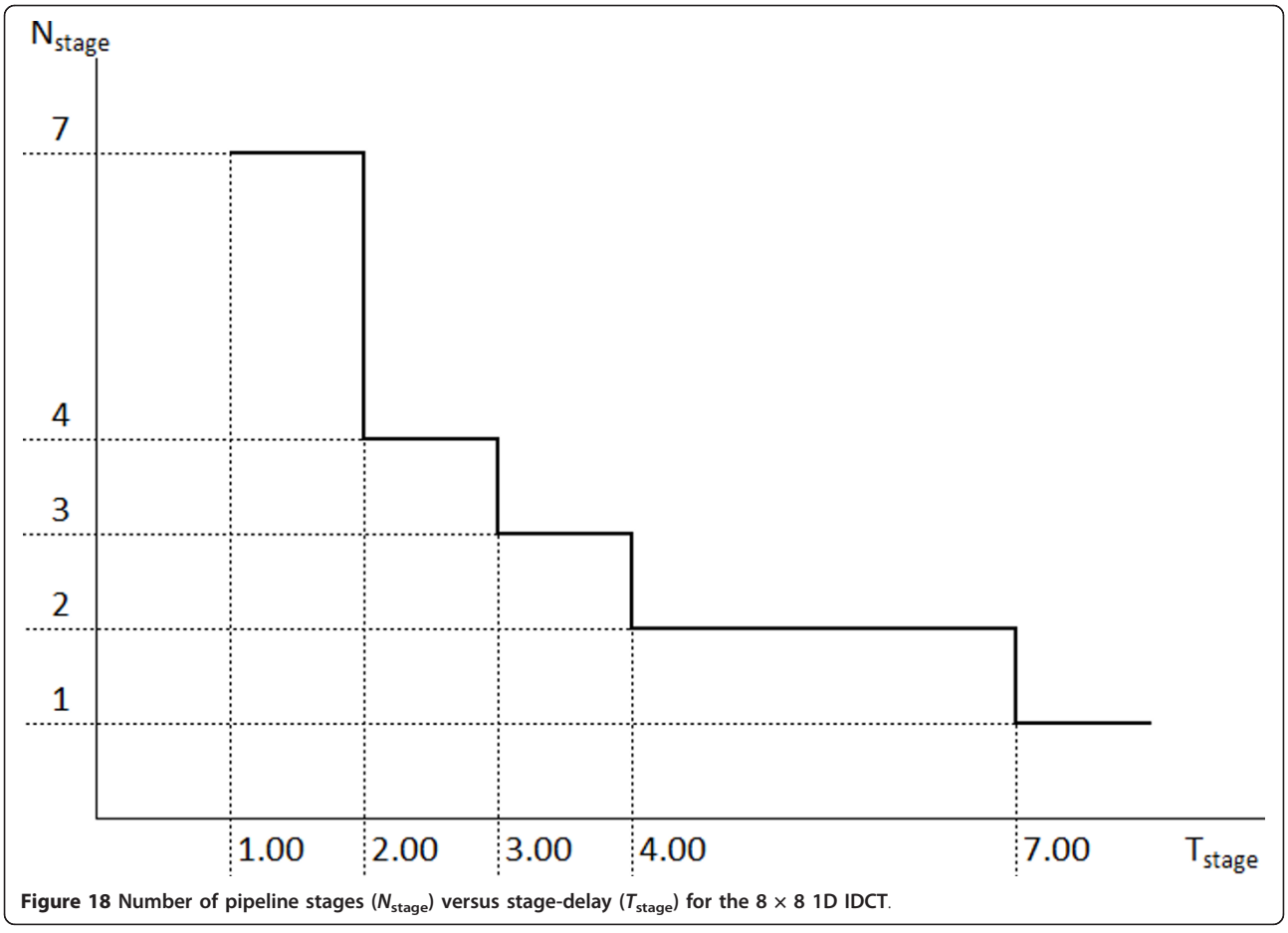


Figure 18 Number of pipeline stages (N_{stage}) versus stage-delay (T_{stage}) for the 8 × 8 1D IDCT.

Mpixels/s, while for Stratix III, it utilizes 1, 571 ALUT with throughput of 922 Mpixels/s. In both FPGAs, resource and throughput show nearly a linear increase from 2-stages to 4-stages pipeline. However, the throughput of 7-stage pipeline for Virtex-5 saturates at roughly the throughput of 3-stages and 4-stages pipeline, while this is not the case for Stratix III. The maximum throughput using Virtex-5 FPGA is 1654 Mpixels/s with total slice of 3, 419 for 4-stages pipeline, which corresponds to 2.07× more slice and 2.16× higher throughput compared to non-pipeline implementation. However, for the best case (resource optimized) 4-stages pipeline, it utilizes only 70% more slice with a throughput increase of 2.08×. As for Stratix III, the highest throughput is the optimal solution (i.e. least resource) for a 7-stage pipeline with 2457 Mpixels/s and ALUT of 3632, which corresponds to 2.31× more ALUT and 2.66× higher throughput. However, higher throughput-to-resource ratio can be obtained in the 4-stages pipeline with only 56% more ALUT and throughput increase by 2.38× compared to non-pipeline implementation. At this level as well (4-stages pipeline), the worst-case design utilizes 15% more ALUT compared to the optimal solution.

Table 7 The 8 × 8 1D IDCT: exploration of pipeline optimization space

N_{stage}	2	3	4	7
T_{stage}	4.00	3.00	2.00	1.00
Reg-width asap	364	520	832	1664
Reg-width alap	312	624	832	1716
Reg-width best	260	468	832	1664
Reg-width worst	494	884	1196	2028
Reg-width reduction (%)	90.0	88.9	43.8	21.9
Feasible schedules	24336	29555604	63002926	4505752
Cut branches	592	1803470	12295281	1298947
Complete schedules	5	3	1	1

6.3 Bayer filter based on improved linear interpolation [46]

Bayer filter is commonly used for demosaicing of color images produced by single-CCD (charge-coupled device) digital cameras. The CCD pixels are preceded in the optical path by a color filter array in a Bayer mosaic pattern, where for each set of 2 × 2 pixels, two diagonally opposed pixels have green filters, and the other two

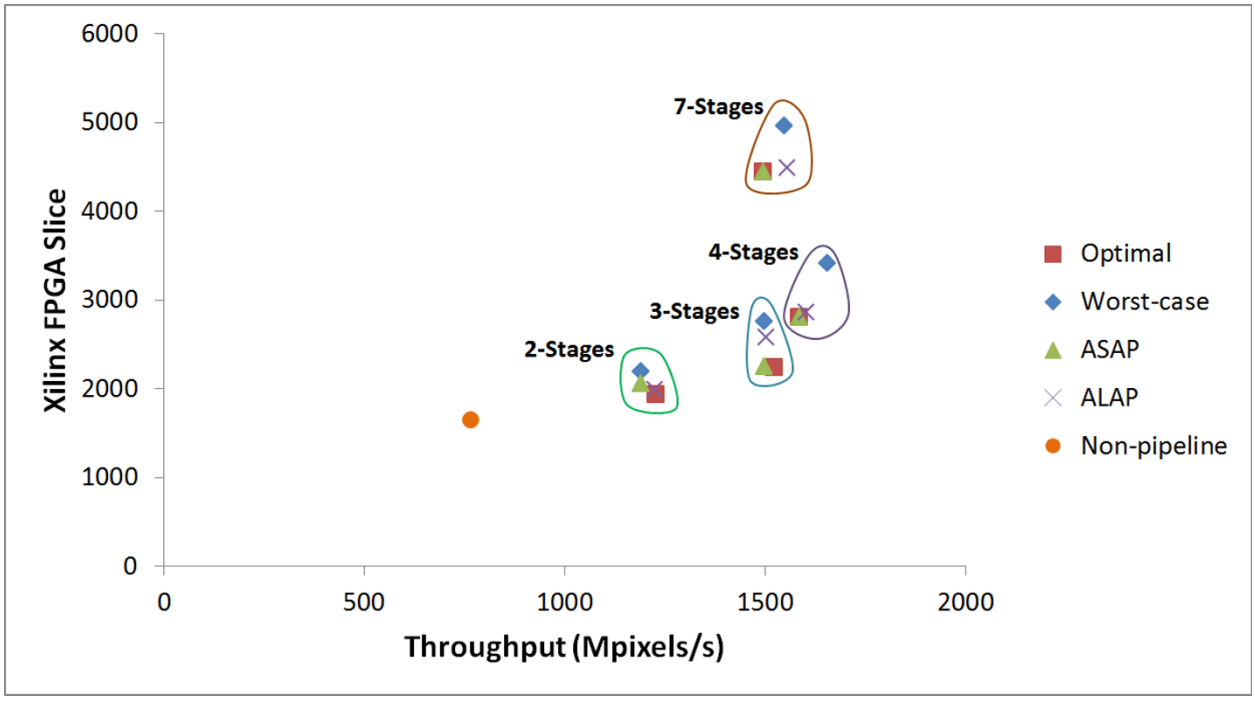


Figure 19 Slice versus throughput for all implementations of the 8 × 8 1D IDCT for Xilinx Virtex-5.

have red and blue filters. The green component is sampled at twice the rate of red and blue since it carries most of the luminance information. The Bayer filter interpolates back the image captured by the CCD sensor, so that every pixel from the sensor (RGGB) can be associated to a full RGB value.

There exists several techniques and algorithms to interpolate images from a CCD sensor. Recently, a new

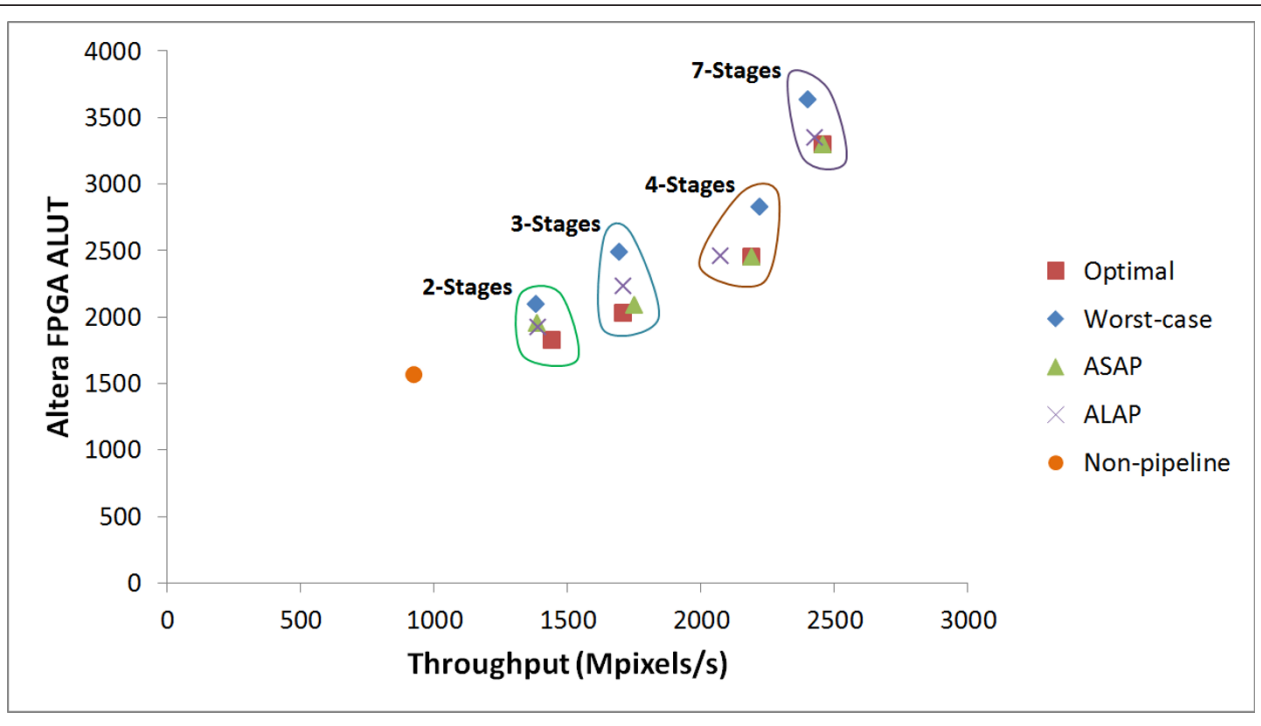


Figure 20 ALUT versus throughput for all implementations of the 8 × 8 1D IDCT for Altera Stratix III.

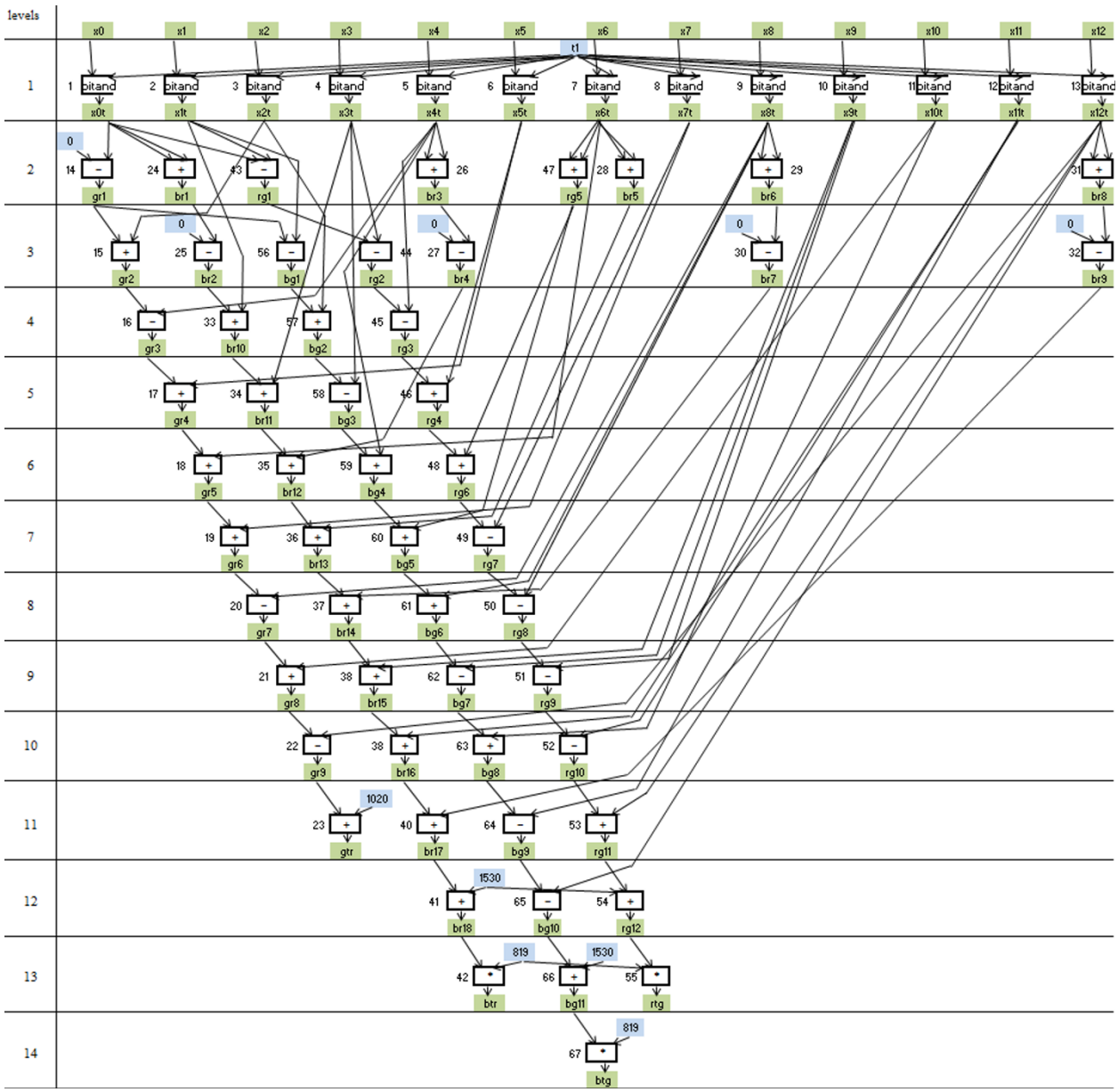


Figure 21 Dataflow graph of the bayer filter core in single-assignment form.

interpolation technique was introduced [46] that outperforms other linear and non-linear algorithms in terms of performance and complexity, and results in high quality output image. In this example, we implemented this technique using the CAL dataflow program, and use our pipeline synthesis and optimization program to find the best pipelining strategy for this design.

The dataflow architecture of the bayer filter has been designed using three separate actors; the *cache, core*, and *control*. The *cache* simply stores the incoming data up to the fifth row, since the core image filtering is done on a 5 × 5 kernel size. The *core* then takes in each kernel, performs image convolution using pre-determined constant coefficients, and outputs the filter core parameters *btr, gtr, rtg*, and *btg*. The *control* keeps track of the current location as to output the correct RGB value. For example, if the current location of the sensor is on the blue pixel, then the control simply sets $r = btr$, $g = gtr$, and $b = center$, where *center* is the blue pixel from the sensor.

From these three actors, the main computation and processing is in the core. Therefore, pipelining this actor is key to obtaining a higher throughput system. The core takes in 13 parallel 8-bit input from a 5 × 5 kernel

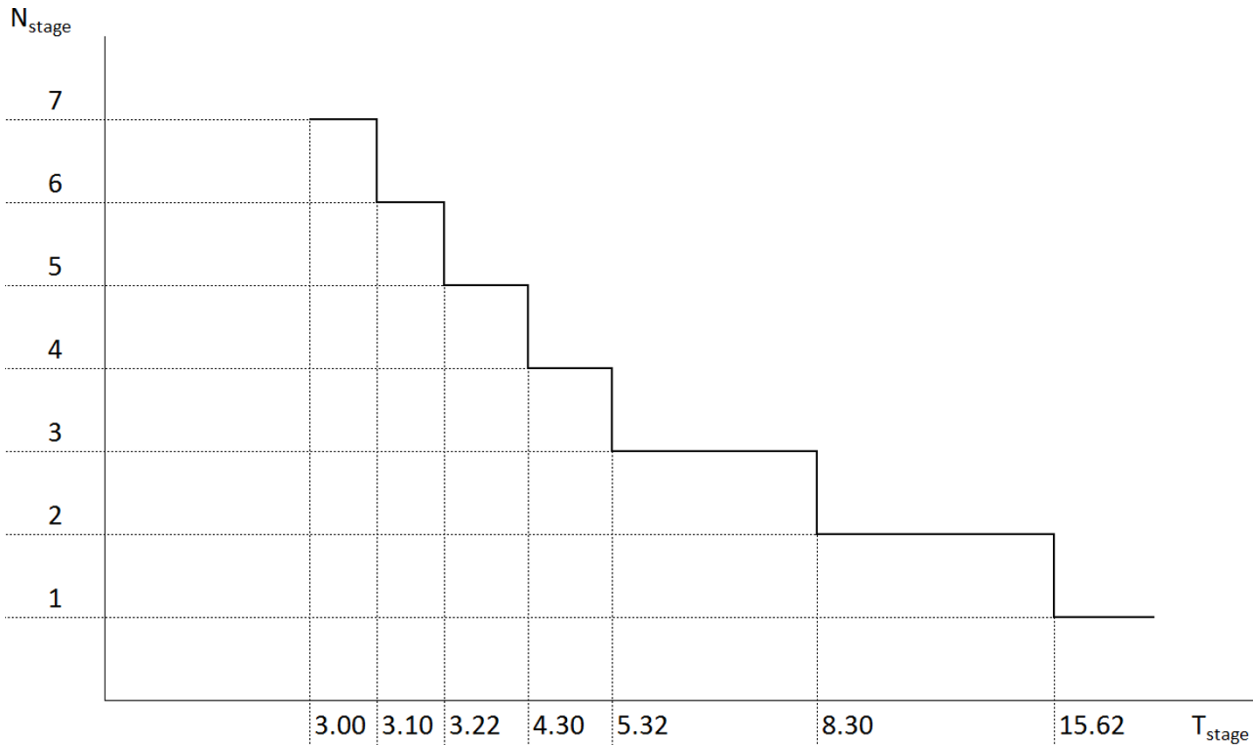


Figure 22 Number of pipeline stages (N_{stage}) versus stage-delay (T_{stage}) for the bayer filter core.

to generate the core outputs. The dataflow graph of the core is shown in Figure 21. The inputs are *x0* to *x12*. The algorithm requires 13 bitands, 20 subtractors, 31 adders, 3 multipliers, and 25 shifters. Similar to the IDCT, shifters are not included in the dataflow graph as it is assumed to have no cost for FPGA implementation.

The first step is to determine the range for valid T_{stage}. For this example, we use the operator delays as given in Table 1. The minimum T_{stage} is determined by the multiply operator, which is 3.00 that would synthesize to the maximum number of pipeline stages N_{stage} = 7. The maximum T_{stage} is found from the longest path matrix *G*, which is 15.62. Any value greater than this would synthesize to a non-pipelined implementation. Further analysis of the dataflow graph shows that for a 2-stage pipeline, minimum T_{stage} is 8.30, 3-stage pipeline for T_{stage} = 5.32, 4-stage pipeline for T_{stage} = 4.30, 5-stage pipeline for T_{stage} = 3.22, and 6-stage pipeline for T_{stage} = 3.10. The graph of N_{stage} versus T_{stage} is given in Figure 22.

Table 8 The bayer filter core: exploration of pipeline optimization space

N_{stage}	2	3	4	5	6	7
T_{stage}	8.30	5.32	4.30	3.22	3.10	3.00
Reg-width asap	215	453	737	998	1259	1428
Reg-width alap	308	501	710	949	1273	1383
Reg-width best	147	340	487	680	972	1095
Reg-width worst	376	660	1013	1320	1650	1658
Reg-width reduction (%)	156.0	94.0	108.0	94.1	69.8	51.4
Feasible schedules	1440	264384	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$

For each N_{stage} given in Figure 22, the optimization space is explored for ASAP, ALAP, best, and worst pipeline schedules. Table 8 summarizes the result. For 2-stage pipeline with T_{stage} = 8.30, the largest register width reduction (156%) is achieved for optimization space of 1440. In this case, the best register width is 147, while the worst is 376. Moving to the 3-stage pipeline, there is still a significant register width reduction of 94.0% from 660 in the worst case to 340 in the best case. For 4-stages and above, the optimization space is too large ($> 10^{10}$), therefore, only 10^8 schedules are generated to find the best proximate solution. Nevertheless, results show significant register width reduction of 108.0, 94.1, 69.8, and 94.1%, respectively, for 4, 5, 6, and 7 stages pipeline.

All the best solutions also show superior results compared to ASAP and ALAP schedules by significant margins. Compared to ASAP the reduction in the total pipeline register width is 46.2, 33.2, 51.3, 46.8, 29.5, and 30.4% for 1 to 7 stages pipelines. In average, the reduction is 39.6%. Similarly for ALAP, the reduction in the

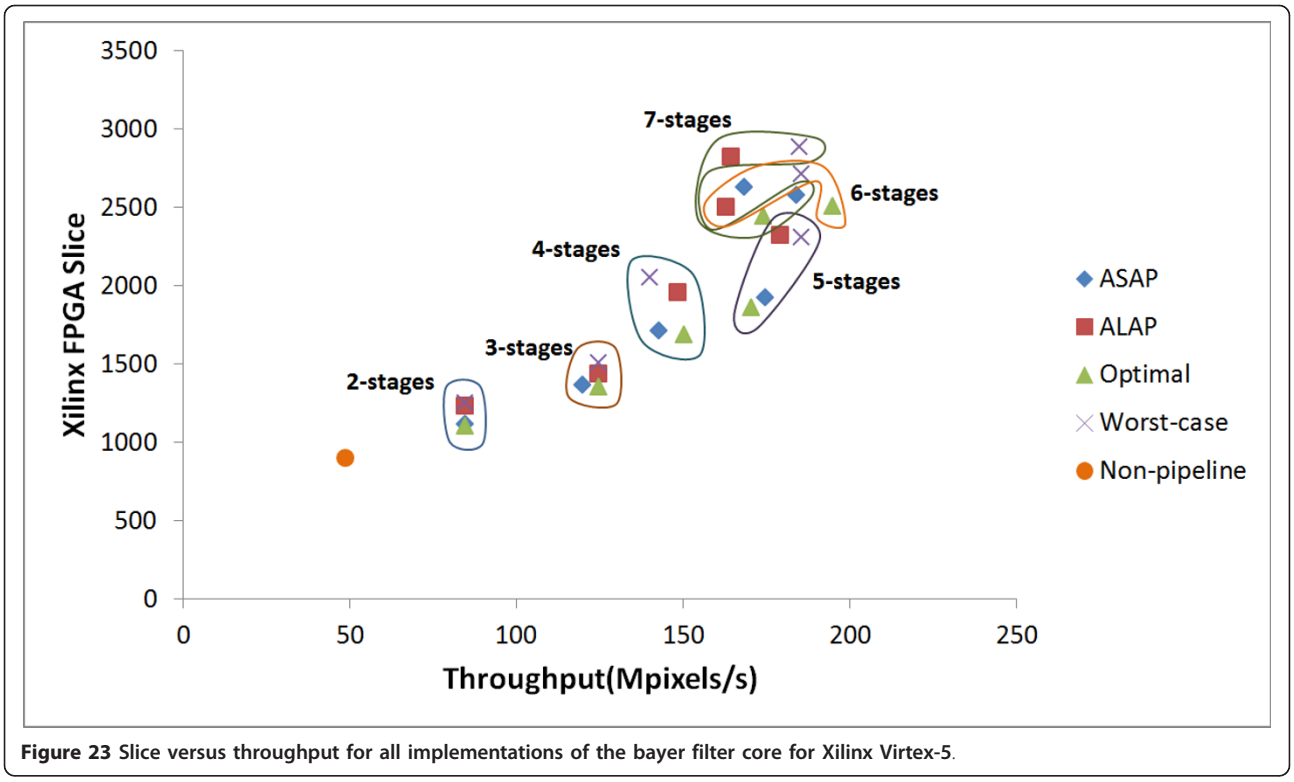


Figure 23 Slice versus throughput for all implementations of the bayer filter core for Xilinx Virtex-5.

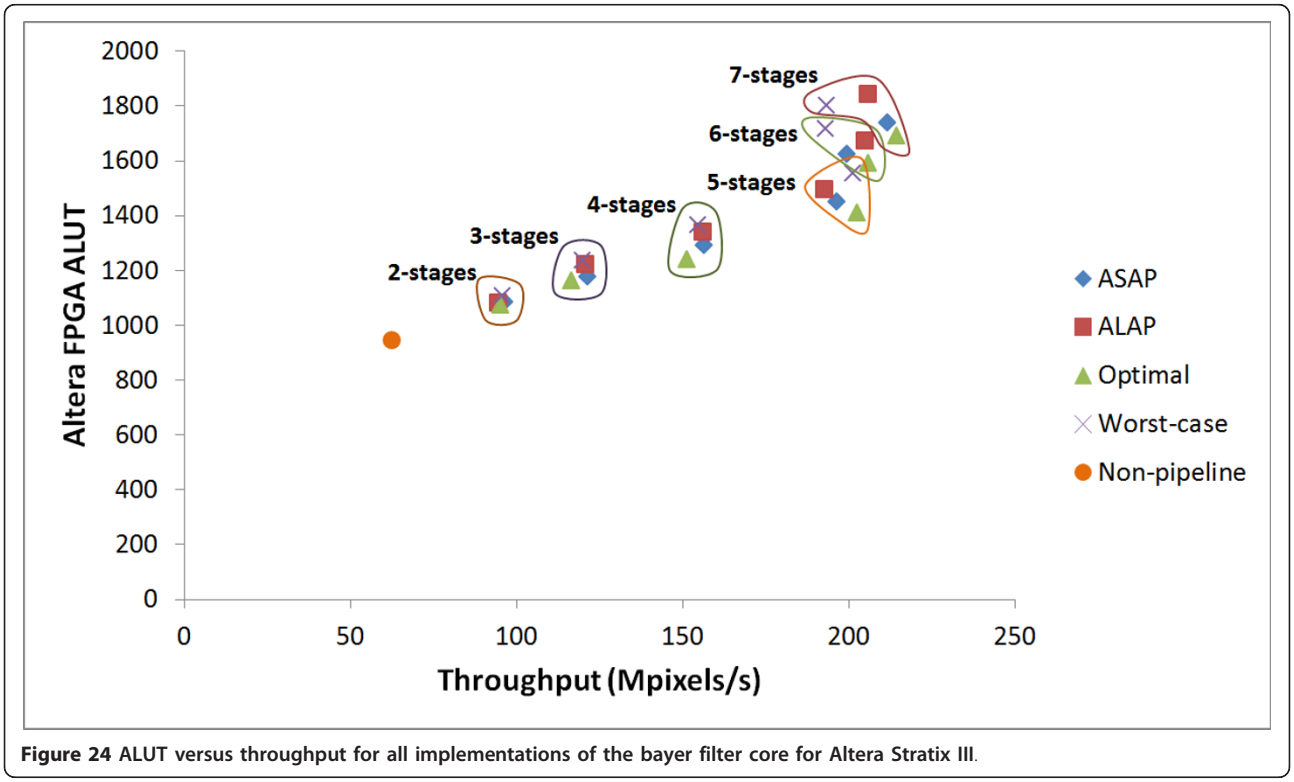


Figure 24 ALUT versus throughput for all implementations of the bayer filter core for Altera Stratix III.

total pipeline register width is 109.5, 47.3, 45.8, 39.3, 31.0, and 26.3%, with average the reduction of 49.9%.

All designs have been synthesized to HDL, and then to Xilinx Virtex-5 and Altera Stratix III FPGAs for implementation. The results are shown in Figures 23 and 24. Non-pipeline implementations are shown by the circle, with throughput of 48.7 and 62.3 Mpixels/s, and slice/ALUT of 897 and 946, respectively, for Virtex-5 and Stratix III. In both FPGAs, throughput is increased almost linearly until the 5-stages pipeline, where it gets saturated when it reaches 6-stages and 7-stages pipeline. Therefore, in terms of throughput-to-resource ratio, 5-stages pipeline is the most superior result. For Virtex-5 best case 5-stages pipeline, throughput is 170.2 Mpixels/s and slice is 1858. This corresponds to 3.5× higher throughput with 2.1× more slice compared to non-pipeline implementation. As for Stratix III, throughput is 202.5 Mpixels/s and ALUT is 1410, which corresponds to 3.3× higher throughput with 1.5× more ALUT. Between optimal and worst-case resource of the 5-stages pipeline, Virtex-5 shows 24% more slice for the worst case compared to the optimal case. As for Stratix III, 10% ALUT difference is observed for the same comparison.

7 Conclusion

In this article, we presented a pipeline synthesis and optimization technique that increases data throughput by minimizing the pipeline stage time for each number of pipeline stages and then reducing the resources by minimizing the pipeline total register width. The technique is designed based on relations, matrices, and graphs that describes an algorithm, which includes operator precedence relation, operator delay and variable width parameters, path delay between operators, and directed conflict and nonconflict graphs. Based on these formulations, a pipeline optimization task is defined with the objective to minimize resource for a given stage-time constraint. This is achieved using the coloring technique to find all possible pipeline schedules for a given number of stages. For each coloring solution, the total register width is evaluated, and the minimum is taken as the optimal pipeline schedule.

Based on the mathematical models, formulations and algorithms, we have developed a program that automatically transforms a non-pipelined CAL actor into pipelined CAL actors. In order to evaluate our technique, we performed experiments of three video processing algorithms. Various pipeline configurations in CAL have been generated from initial CAL descriptions, and then synthesized to HDL for implementation on Xilinx Virtex-5 and Altera Stratix III FPGAs. Results of the pipeline synthesis are very promising with up to 3.9× increase in throughput for Virtex-5 and 3.4× for Stratix III, as compared between pipelined and non-pipelined implementations. The optimization technique is equally effective with up to 39.6 and 49.9% average total register width reduction between the optimal, and ASAP and ALAP pipeline schedules, respectively.

Endnotes

[a]International Organization for Standardization/International Electrotechnical Commission.

[b]SSA is a form that is used extensively in compiler designs where each variable is assigned to in only one place of the source.

Competing interests
The authors declare that they have no competing interests.

Received: 1 March 2011 Accepted: 10 November 2011
Published: 10 November 2011

References

1. Ab-Rahman A, Thavot R, Mattavelli M, Faure P: **Hardware and software synthesis of image filters from cal dataflow specification.** *2010 Conference on PhD Research in Microelectronics and Electronics (PRIME)* 2010, 1-4.
2. Eker J, Janneck J: **CAL Language Report: Specification of the CAL Actor Language.** University of California-Berkeley; 2003.
3. Bhattacharyya S, Brebner G, Janneck J, Eker J, Platen CV, Mattavelli M, Raulet M: **Opendf a dataflow toolset for reconfigurable hardware and multicore systems.** *Proceedings of the Swedish Workshop on Multicore Computing* 2008, 29-35.
4. Parlour D: **CAL Coding Practices Guide: Hardware Programming in the CAL Actor Language.** Xilinx Inc.; 2003.
5. Hwang C-T, Hsu Y-C, Lin Y-L: **PIs: a scheduler for pipeline synthesis.** *IEEE Trans Comput-Aided Design Integrated Circuits Syst* 1993, **12**:1279-1286.
6. Park N, Parker AC: **Sehwa: a software package for synthesis of pipelines from behavioral specifications.** *IEEE Trans Comput-Aided Design* 1988, **7**:358-370.
7. Paulin PG, Knight JP: **Force-directed scheduling for the behavioral synthesis of asic's.** *IEEE Trans Comput-Aided Design* 1989, **8**:661-679.
8. Hwang KS, Casavant AE, Chang C-T: **MA d'Abreu, Scheduling and hardware sharing in pipelined data paths.** *Proceedings of the ICCAD-89* 1989, 24-27.
9. Girczyc EM: **Loop winding–a data flow approach to functional pipelining.** *Proceedings of the IEEE ISCAS* 1987, 382-385.
10. Potasman R, Lis J, Aiken A, Nicolau A: **Loop winding–a data flow approach to functional pipelining.** *Proceedings of the 27th Design Automation Conference* 1990, 444-449.
11. Aiken A, Nicolau A: **Optimal loop parallelization.** *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation* 1988.
12. Haroun BS, Elmasry MI: **Architectural synthesis for dsp silicon compiler.** *IEEE Trans Comput-Aided Design* 1989, **8**:431-447.
13. Goossens G, Rabaey J, Vandewalle J, Man HD: **An efficient micro-code compiler for applications specific dsp processors.** *IEEE Trans Comput-Aided Design* 1990, **9**:925-937.
14. Jun H-S, Hwang S-Y: **Design of a pipelined datapath synthesis system for digital signal processing.** *IEEE Trans Comput-Aided Design Integrated Circuits Syst* 1994, **12**:292-303.
15. Leiserson CE, Saxe JB: **Optimizing synchronous systems.** *J VLSI Comput Syst* 1983, **1(1)**:41-67.
16. Malik S, Singh KJ, Brayton RK, Sangiovanni-Vincentelli A: **Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis.** *IEEE Trans Comput-Aided Design Integrated Circuits Syst* 1993, **12**:568-578.
17. Shenoy N: **Retiming: theory and practice.** *VLSI J Integr* 1997, **22(1-2)**:1-21.
18. Kahn G: **The semantics of a simple language for parallel programming.** *Proceedings of IFIP Congress 74* 1974, 471-475.

19. Blythe SA, Walker RA: **Efficient optimal design space characterization methodologies.** *ACM Trans Des Autom Electron Syst* 2000, **5**:322-336.
20. Ascia G, Catania V, Palesi M: **Design space exploration methodologies for ip-based system-on-a-chip.** *IEEE International Symposium on Circuits and Systems, 2002. ISCAS 2002* 2002, **2**:364-367.
21. Mathur V, Prasanna V: **A hierarchical simulation framework for application development on system-on-chip architectures.** *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International* 2001, 428-434.
22. Lee EA, Messerschmitt DG: **Synchronous data flow.** *Proc IEEE* 1987, **75**:1235-1245.
23. Thiele L, Chakraborty S, Gries M, Kunzli S: **A framework for evaluating design tradeoffs in packet processing architectures.** *Proceedings - Design Automation Conferece* 2002, 880-885.
24. Benini L, Bertozzi D, Bruni D, Drago N, Fummi F, Poncino M: **Systemc cosimulation and emulation of multiprocessor soc designs.** *Computer* 2003, **36**:53-59.
25. Lahiri K, Raghunathan A, Dey S: **System-level performance analysis for designing on-chip communication architectures.** *IEEE Trans Comput-Aided Design Integrated Circuits Syst* 2001, **20**:768-783.
26. **SPW User's Manual.** *Cadence Design Systems* Foster City, CA, USA.
27. **DSP Builder User Guide Software Version 9.1.** Altera, San Jose, CA, USA;, 9.1 2009.
28. **AccelDSP Synthesis Tool User Guide Release 10.1.** Altera, San Jose, CA, USA;, 10.1 2008.
29. **Simulink 7 User Guide.** Mathworks, Natick, MA, USA;, 7 2010.
30. Gupta S, Dutt N, Gupta R, Nicolau A: **Spark: a high-level synthesis framework for applying parallelizing compiler transformations.** *International Conference on VLSI Design* 2003, 461-466.
31. Martin E, Sentieys O, Dubois H, Philippe JL: **Gaut: an architectural synthesis tool for dedicated signal processors.** *European Design Automation Conference– Proceedings* 1993, 14-19.
32. **Catapult C Synthesis.** Mentor Graphics, Wilsonville, OR, USA; 2005.
33. Demicheli G: **Hardware synthesis from c/c++ models.** *Design, Automation and Test in Europe Conference and Exhibition 1999* 1999, 382-383.
34. Gao L, Zaretsky D, Mittal G, Schonfeld D, Banerjee P: **A software pipelining algorithm in high-level synthesis for fpga architectures.** *Proceedings of the 10th International Symposium on Quality Electronic Design, ISQED 2009* 2009, 297-302.
35. Hewitt C: **Viewing control structures as patterns of passing messages.** *J Artif Intell* 1977, **8**:323-363.
36. Lucarz C, Mattavelli M, Wipliez M, Roquier G, Raulet M, Janneck J, Miller I, Parlour D: **Dataflow/actor-oriented language for the design of complex signal processing systems.** *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image processing (DASIP)* 2008.
37. Janneck JW, Miller ID, Parlour DB, Roquier G, Wipliez M, Raulet M: **Synthesizing hardware from dataflow program: an mpeg-4 simple profile decoder case study.** *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS), October 2008* 2008.
38. Olsson T, Carlsson A, Wilhelmsson L, Eker J, Von Platen C, Diaz I: **A reconfigurable ofdm inner receiver implemented in the cal dataflow language.** *2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems* 2010, 2904-2907.
39. Roudel N, Berry F, STrot J, Eck L: **A new high-level methodology for programming fpga-based smart camera.** *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010* 2010, 573-578.
40. Aman-Allah H, Maarouf K, Hanna E, Amer I, Mattavelli M: **Cal dataflow components for an mpeg rvc avc baseline encoder.** *J Signal Process Syst* 2009, **65**:1-13.
41. Wipliez M, Roquier G, Nezan J: **Software code generation for the rvc- cal language.** *J Signal Process Syst* 2009, **65**:1-11.
42. Szedo G: **Color-Space Converter: RGB to YCrCb.** Xilinx Inc.; 2007.
43. DeMicheli G: *Synthesis and Optimization of Digital Circuits.* 3 edition. McGraw- Hill, New Jersey; 1994.
44. ISO/IEC: **Information technology–MPEG video technologies–Part 2: Fixed-point 8 × 8 inverse discrete cosine transform and discrete cosine transform.** *International Standard* 2007.
45. Khayam SA: *The Discrete Cosine Transform (DCT): Theory and Application. Lecture Notes* ; 2003.
46. Malvar HS, He L, Cutler R: **High-quality linear interpolation for demosaicing of bayer-patterned color images.** *IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP* 2004, **3**.

doi:10.1186/1687-5281-2011-19
Cite this article as: Ab Rahman *et al.*: Pipeline synthesis and optimization of FPGA-based video processing applications with CAL. *EURASIP Journal on Image and Video Processing* 2011 **2011**:19.