RZ 3611 (# 99621) 06/06/05
Computer Science 10 pages

Research Report

Instruction-Set Synthesis for Reactive Real-Time Processors: An ILP Formulation

Gero Dittmann and Paul Hurley

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{ged,pah}@zurich.ibm.com

Instruction-Set Synthesis for Reactive Real-Time Processors: An ILP Formulation

Gero Dittmann and Paul Hurley
IBM Research, Zurich Research Laboratory
Säumerstrasse 4 / Postfach
8803 Rüschlikon, Switzerland
{ged,pah}@zurich.ibm.com

Abstract

Today's design methodologies for application-specific instruction-set processors (ASIPs) focus on the data-dominated domain characterized by computation-intensive applications such as digital signal processing. There is, however, a lack of methods for control-dominated domains such as network-protocol processing. These domains are characterized by branch-intensive applications with fine-grained timing constraints imposed by frequent interactions with the ASIP environment. The main challenge here is not to speed up over-all application runtime, but to meet the many timing constraints. This challenge can be addressed by introducing special instructions that speed up the timing-critical paths.

In this paper we propose the first ASIP design methodology for the control-dominated domain. We divide the task into two separate optimization problems, each formulated as an integer linear program (ILP). The first is to find a complete set of operation sequences that must be implemented as macro instructions in order to meet given timing constraints. The second is to reduce the number of parallel instruction issues, achieved by bundling pairs of instructions. An ILP solver can automate the instruction-set synthesis, enabling the analysis of large benchmarks and yielding optimal solutions.

1 Introduction

Owing to the ever-decreasing feature size of today's semiconductor processes, the cost of a mask set has crossed the one-million-dollar line. Given this investment, a design must be applicable for multiple purposes. This flexibility is commonly provided by programmable elements. A fine-grained and gradual trade-off between flexibility and performance can be achieved with *application-specific instruction-set processors (ASIPs)*. The instruction set of an ASIP is specialized for a particular class of applications by compound instructions that speed up critical parts of the applications without compromising the flexibility of the processor *in its application domain*. In this way, ASIPs combine the flexibility of general-purpose processors (GPPs) with the performance of hard-wired logic.

Most research publications on ASIPs concentrate on the design of digital signal processors (DSPs) or, more generally, on the data-dominated application domain [1, 2, 3]. Data-dominated applications are characterized by long arithmetic sections between control-flow boundaries, i.e., between branches. Furthermore, they typically contain many computation-intensive loops. Processing often starts with receiving a sample of data and ends with sending out a resulting frame. Inbetween start and end, there is no other I/O to be handled. Hence, there is only one deadline to be met per algorithm run: The resulting frame has to be output in time. This kind of timing constraint is called a *rate constraint* because the overall running time of an algorithm is constrained to guarantee that a required rate of samples per time unit can be processed.

Control-dominated applications, in contrast, feature many branches interleaved with short computation blocks, and loops are rare. In most control-dominated real-time systems, such as protocol processors [4], there is not only one deadline at the end of a run but many I/O interactions with the environment, many of which have a deadline associated with them. As a consequence of these fine-grained timing constraints, the focus moves away from patterns that occur frequently and therefore provide an overall speed-up. Instead, patterns must be implemented as instructions in order to meet the fine-grained timing constraints, even if they occur only once in an application. Table 1 contrasts the properties of the two application domains.

Table 1: Characteristics of application domains.

Properties	Data-dominated	Control-dominated
Examples	DSP, media processor	NP, microcontroller
Arithmetic sections	long	short
Branches	few	many
Loops	many	few
Memory size	large	small
Arithmetic type	fractional	integer
Timing	rate constraints	fine-grained
Data-dependent wait	no	yes
Pattern purpose	speed-up	forced by timing
Pattern metric	occurrence frequency	meets timing constraints

The first step in any application-driven ASIP design methodology is to define a set of benchmark applications that is representative of the targeted domain. Often there already is a code base in a commodity programming language, such as C. This code, however, is lacking the timing constraints required by the ASIP environment. In [5], we proposed methods to specify fine-grained timing constraints for reactive systems in ANSI C and to translate them to a timing layer in an intermediate representation (IR). Graph edges in the timing layer connect I/O operands and are annotated with the minimum and maximum time between these nodes.

In this paper we introduce methods that analyze an IR graph to select operation patterns for implementation as special instructions, tailored to the control-dominated domain. The resulting instruction set has to enable an implementation of the benchmark applications such that the following two kinds of constraints are met:

- the timing constraints given by the timing layer of the IR, and
- a maximum number of parallel instruction issues, specified by the ASIP designer.

An instruction set that meets these constraints is to be optimized in two respects:

- The primary goal is to minimize the maximum latency of any instruction in the instruction set. This improves the implementability of the instruction set with the required cycle time.
- The secondary goal is to minimize the number of instructions in the instruction set. This minimizes the number of bits needed for the instruction encoding.

We formulate these optimizations as two consecutive scheduling problems. The first is to segment each path covered by a timing constraint into patterns such that the constraint is met while balancing the latency of the patterns to work towards the primary optimization goal. The second is to bundle parallel patterns such that the constraint on parallel issues is met while keeping the number of incurred instructions low in order to work towards the secondary optimization goal. Figure 1 shows the entire design flow.

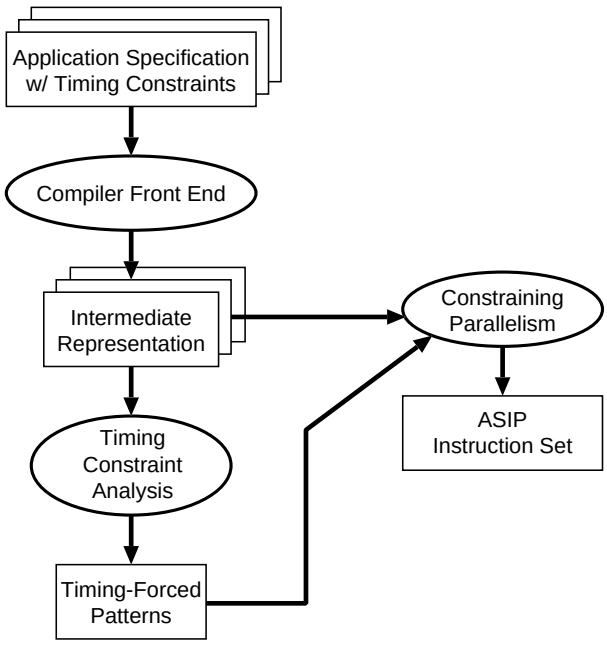

Figure 1: ASIP design flow.

This paper is structured as follows. We start by giving an overview of scheduling algorithms commonly used in high-level synthesis (HLS) in Section 2. Section 3 states our assumptions on the ASIP architecture. In Sec-

tion 4 we present our integer linear program (ILP) formulation of finding timing-forced sequential operation patterns. We introduce our ILP for reducing the number of parallel instruction issues in Section 5. In Section 6 we propose a new, low-complexity on-chip communication scheme that only becomes possible with the real-time guarantees that our design methodology provides. Finally, we conclude in Section 8.

To our knowledge, this is the first complete instruction-set design flow for control-dominated applications.

2 Related Work

The simplest scheduling algorithms used in HLS are *as-soon-as-possible (ASAP)* and *as-late-as-possible (ALAP)*. ASAP positions each operation in the first step in which all its inputs are available. Similarly, the ALAP schedule positions each operation just before all operations that read its output and in the latest control step possible without adding another step to the total schedule. In both cases, the total schedule length is equal to the length of the critical paths. The term *mobility* for the difference between ASAP and ALAP schedules was coined in [6]. ASAP and ALAP schedules do not take resource constraints into consideration.

An early algorithm for resource-constrained scheduling is *list scheduling (LS)* [7]. In LS, operations are ordered according to their dependencies on other operations. A priority function assigns precedence values to the operations. Based on these values, the operations are then iteratively assigned to control steps. LS and its many variations are widely used in synthesis systems because they are simple and efficient.

Another popular algorithm is *force-directed scheduling (FDS)* [8]. FDS is time-constrained, i.e., it tries to minimize the resources required to achieve a given maximum schedule length. The priority function in FDS is based on the mobility of operations and the resource requirements in each control step. Operations with the lowest mobility, the least effect on the mobility of other operations, and the lowest resource increase are scheduled first.

More complex scheduling algorithms include iterative scheduling [9] and the formulation of scheduling as an ILP [10]. Solving an ILP provides optimum schedules. A comprehensive introduction to the scheduling problem followed by a survey of the most popular scheduling algorithms for HLS can be found in [11].

Most HLS systems only allow the specification of static timing constraints, neglecting minimum, maximum, and range constraints for optimization and scheduling. A notable exception of a constructive scheduling algorithm that considers these types of timing constraints for the synthesis process has been described in [12].

3 Processor Architecture

In our research, we make the following assumptions about the ASIP architecture:

VLIW format. To avoid the hardware overhead that superscalar architectures entail, we opt for the very long instruction word (VLIW) approach in order to obtain a small footprint for the ASIP. With VLIW processors, it is up to the compiler to pack multiple instructions into one memory word that will then be executed in parallel. The resulting binary incompatibility between processor models with different issue widths is usually not a problem for embedded systems, as it is possible to recompile code for a new processor. Moreover, controlling parallelism by the compiler simplifies the scheduling under timing constraints because the scheduler does not need to estimate the behavior of the hardware parallelization.

No pipelining. Control-dominated applications have many branches, which is the classical stumbling block for processor pipelines. To keep the pipeline filled after a branch, speculative execution can be employed but it is only effective if the speculation is correct. Branch prediction is used to improve the rate of correct speculations but the rare occurrence of loops in control-dominated applications renders branch prediction largely ineffective. Furthermore, the short arithmetic sections in control-dominated applications constrain opportunities for speculation. Finally, speculation is a probabilistic technique that interferes with hard timing constraints. We conclude that pipelining and control-dominated applications do not match well because frequent stalls make pipelining largely inefficient in this domain. Consequently, the overhead in logic and the increased instruction completion time due to pipeline registers and imbalances between the stages would not be justified.

Unlimited registers. The control-dominated applications we analyze typically do not have large storage requirements so that the required number of registers will be low. Not imposing constraints on the number of available registers, on the

other hand, significantly reduces the complexity of scheduling.

Single-cycle instructions. Control-dominated applications typically do not use floating-point data types but rely exclusively on integers. Hence, it is not a severe restriction to disregard multi-cycle instructions.

With these assumptions, the complexity of our methods is kept under control for this first proof-of-concept methodology. However, our methodology can be combined with existing methods to overcome its restrictions. In particular, it can be complemented with pipeline design methods and register scheduling approaches to support pipelined architectures with a limited number of registers.

4 Timing-Forced Patterns

4.1 Problem Statement

The main concern in developing an ASIP for control-dominated applications is to meet the timing constraints specified by the benchmarks. The generated instruction set must be able to implement the applications with the required timing. Therefore, our first step is to find the operation patterns that are *forced* to be part of the pattern set by the timing constraints. In this selection process, it is important to balance the size of patterns in the pattern set because all patterns will have to be implemented in a single processor cycle and therefore the most complex pattern will determine the critical path in the processor design. In order to maintain control-flow dependencies, a pattern does not cross branches.

The scheduling algorithms mentioned in Section 2 determine how to distribute operations over a given or minimum number of time steps with a given or minimum number of resources. None of them, however, addresses the question of how to bundle operations in an instruction to obtain a lean instruction set that meets all constraints and in which instruction latencies are balanced. Therefore, our objective is different from the general scheduling problem.

To analyze applications, we transform them to a control-data flow graph (CDFG), $G = (V, E)$, with V a set of nodes and E a set of directed edges $e = (u, v) \in E$ with $u, v \in V$. There are two types of nodes in the set $V = V_{\text{op}} \cup V_{\text{dmy}}$: the set of operation nodes in the data-flow layer, V_{op}, and dummy nodes, V_{dmy}, that connect the control layer with the data-flow layer.

The dummy nodes serve as unified entry or exit points for the control nodes, i.e., for the basic blocks. An entry node connects all leaves of all data flow graphs (DFGs) in a control node, and an exit node connects all roots of the DFGs. The edges from the control layer connect the exit node of their source with the entry node of their destination in the CDFG. Basic blocks with a conditional branch get one exit node for each outgoing control edge.

The set of edges in G therefore consists of edges E_{dmy} between dummy nodes and leaves or roots of DFGs, and regular control and data-flow edges: $E = E_{\text{dmy}} \cup E_c \cup E_d$. An edge in the CDFG is denoted $v_i \rightarrow v_k$, where v_i is an immediate predecessor of v_k.

The timing constraints specified with the benchmark applications are represented in a timing layer [5] on top of the CDFG. The edges in the timing layer impose maximum or minimum constraints on the time between the CDFG operands at their ends. This represents an optimization problem on the CDFG, namely, how to schedule the operation nodes along each timing edge.

4.2 ILP Formulation

We formalize the optimization problem in the form of an ILP [13]. The result of the optimization will be a set of m patterns $S_t = \{I_1, \ldots, I_m\}$, with each pattern being a DFG. The latency of pattern $s \in \{1, \ldots, m\}$ is the length of its critical path, denoted $|I_s|$. Our global objective is to balance the latency of the selected patterns, i.e., to minimize the maximum latency in the pattern set:

$$\min\{\ \max(|I_s| : 1 \leq s \leq m)\ \}. \tag{1}$$

Let $X = [x_{i,j}]_{|V_{\text{op}}|, j_{\max}}$ be a scheduling matrix of 0-1 integer variables with $v_i \in V_{\text{op}}$, $j \in \{1, \ldots, j_{\max}\}$, and $x_{i,j} = 1$ iff v_i is scheduled in time step j (see Figure 2). $|V_{\text{op}}|$ is the number of operation nodes in G. Note that the matrix considers only operation nodes. The dummy nodes, in contrast, cannot be scheduled as they do not represent any operation.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,j_{\max}} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,j_{\max}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{|V_{\text{op}}|,1} & x_{|V_{\text{op}}|,2} & \cdots & x_{|V_{\text{op}}|,j_{\max}} \end{bmatrix} \begin{matrix} \uparrow \\ \text{operation} \\ \text{nodes} \\ \downarrow \end{matrix}$$

$$\longleftarrow \text{ time steps } \longrightarrow$$

Figure 2: Scheduling matrix.

Let $G_t = (V_t, E_t)$ be the subgraph of G between the endpoints of a timing edge t. Let P_t be the set of all

acyclic paths p through G_t from one timing-edge endpoint to the other. A path is defined by a vector $p \in \mathbb{B}^{|V_{\text{op}}|}$ such that $p_i = 1$ iff v_i is on the path. Componentwise multiplication of the vector $(x_{1,j}, \ldots, x_{|V_{\text{op}}|,j})$ for one time step j with a path vector p yields a *characteristic vector* of those operation nodes on the path that are scheduled in that time step. As these nodes must be implemented by the same pattern, the number of 1's in the characteristic vector is equivalent to the number of operation nodes of the path in that time step, which corresponds to the latency of the path segment. We can write this latency as the scalar product of the path vector with the column vector x_j of the time step j in the scheduling matrix:

$$|p(j)| = p_1 x_{1,j} + \ldots + p_{|V_{\text{op}}|} x_{|V_{\text{op}}|,j} = p \cdot x_j. \quad (2)$$

The path segment with the largest latency corresponds to the critical path in the slowest pattern. Therefore, the objective in Eq. (1) can be recast as a function of all path vectors of all timing edges, which are combined in the set P:

$$\min\{\, \max(\sum_{i=1}^{|V_{\text{op}}|} p_i\, x_{i,j},\ \forall\, p \in P,\ j \in \{1, \ldots, j_{\max}\})\,\}$$

$$= \min\{\, \max(p \cdot x_j, \forall p \in P,\, j \in \{1, \ldots, j_{\max}\})\,\}. \quad (3)$$

This is the objective function for the ILP. Now the constraints a valid schedule must meet can be developed. The first requires that each operation node be scheduled in exactly one time step:

$$\sum_{j=1}^{j_{\max}} x_{i,j} = 1, \quad \forall\, i : v_i \in V_{\text{op}}. \quad (4)$$

The precedence of nodes in the CDFG must be preserved. We achieve this by requiring that the time step of a node be equal to or higher than the time step of its predecessors. If g_l is the time step for v_l, we obtain

$$g_i \leq g_k, \quad \forall\, (i,k) : v_i \rightarrow v_k. \quad (5)$$

The time step g_l of a node l is expressed by the sum

$$g_l = \sum_{j=1}^{j_{\max}} j\, x_{l,j}. \quad (6)$$

Transforming the inequality to $g_i - g_k \leq 0$ and substituting with Eq. (6), we get the precedence constraint:

$$\sum_{j=1}^{j_{\max}} j\, x_{i,j} - \sum_{j=1}^{j_{\max}} j\, x_{k,j} \leq 0, \quad \forall\, (i,k) : v_i \rightarrow v_k,\ v_i, v_k \in V_{\text{op}}. \quad (7)$$

This constraint only applies to operation nodes because dummy nodes are not assigned to any time step. Therefore, another constraint to preserve precedence across dummy nodes is needed:

$$\sum_{j=1}^{j_{\max}} j\, x_{i,j} - \sum_{j=1}^{j_{\max}} j\, x_{k,j} \leq -1, \quad \forall\, (i,k) : v_i \rightarrow v_{\text{exit}} \rightarrow v_{\text{entry}} \rightarrow v_k,\ v_i, v_k \in V_{\text{op}},\ v_{\text{exit}}, v_{\text{entry}} \in V_{\text{dmy}}. \quad (8)$$

The left-hand side of the inequality must be negative because operation nodes connected across dummy nodes belong to different basic blocks and thus must not be scheduled in the same time step. As a consequence of this constraint, the ILP has no solution if there exists a path having more control nodes than the timing edges allow cycles. In this case, transformations such as if-conversion [14] must be used to decrease the number of control nodes and resolve the situation.

Finally, the timing constraints must be considered. For each maximum time $v_i \xrightarrow{t_{\max}} v_k$ between two operation nodes v_i and v_k with I/O nodes as operands, $g_k - g_i \leq t_{\max}$ is required for their assigned time steps g_i and g_k. Similarly, from each minimum time $v_i \xrightarrow{t_{\min}} v_k$ follows $g_k - g_i \geq t_{\min}$. Substituting with Eq. (6) results in:

$$\sum_{j=1}^{j_{\max}} j\, x_{i,j} - \sum_{j=1}^{j_{\max}} j\, x_{k,j} \leq t_{\max}, \quad \forall\, (i,k) : v_i \xrightarrow{t_{\max}} v_k,\ v_i, v_k \in V_{\text{op}} \quad (9)$$

$$\sum_{j=1}^{j_{\max}} j\, x_{i,j} - \sum_{j=1}^{j_{\max}} j\, x_{k,j} \geq t_{\min}, \quad \forall\, (i,k) : v_i \xrightarrow{t_{\min}} v_k,\ v_i, v_k \in V_{\text{op}}. \quad (10)$$

This completes our formulation with objective function (3) and constraints (4) for assignment, (7) and (8) for precedence between operation nodes, and (9) and (10) for timing. In a solution to the optimization problem, each set of nodes scheduled in the same time step and connected by data dependencies in the CDFG represents a pattern in the pattern set S_t.

5 Constraining Parallel Instruction Issues

5.1 Problem Statement

The methods to find timing-forced patterns described in Section 4 consider constraints on the number of instructions in a sequence, defined in the form of timing

constraints. Another type of constraint provided by the designer in our methodology is the maximum number of instructions issued in parallel by the ASIP to be designed, $k_{\max}$. Our approach to meet this constraint is to bundle patterns that frequently occur in parallel.

Building upon the results of the preceding section, we replace the operation nodes in the CDFG with their associated patterns in S_t, resulting in a set of pattern nodes V_{pat}. We get a graph $G' = (V', E')$ with a set of nodes $V' = V_{\text{pat}} \cup V_{\text{dmy}}$ and edges E'. It is not necessary to migrate the timing edges as they are attached to I/O operands, which have not changed in the process.

The optimization problem to be solved on this graph is how to schedule the nodes in time steps with the minimum number of incurred bundles of parallel patterns. Again, we develop a formal definition of the problem as an ILP.

5.2 ILP Formulation

The result of the second optimization problem will be a set of instructions S_p. Our objective is to keep the number of instructions in S_p as low as possible:

$$\min\{\ |S_p|\ \}. \tag{11}$$

Let $Y = [y_{i,j,k}]_{|V_{\text{pat}}|, j_{\max}, k_{\max}}$ be a three-dimensional scheduling array of 0-1 integer variables with $v_i \in V_{\text{pat}}$, $j \in \{1, \ldots, j_{\max}\}$, and $y_{i,j,k} = 1$ iff v_i is scheduled in time step j and parallel-issue slot k (see Figure 3). $|V_{\text{pat}}|$ is the number of pattern nodes in G'. The dummy nodes are not represented in the array.

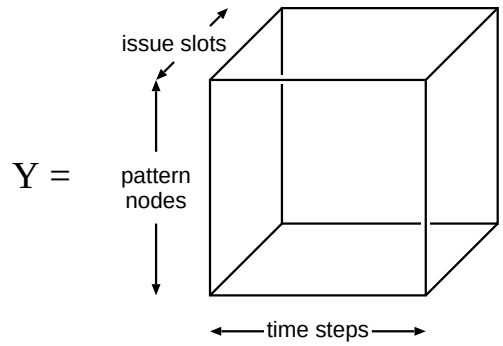

Figure 3: Three-dimensional scheduling array.

Each issue slot k in each time step j is represented by a *characteristic vector* $y_{j,k} = (y_{1,j,k}, \ldots, y_{|V_{\text{pat}},j,k})$ in the array with a 1 at each node that is scheduled in that particular slot. The pattern associated with each node is determined by a function $\tau : V_{\text{pat}} \rightarrow S_t$. The combination of patterns in one issue slot forms a pattern bundle in S_p. Hence, the number of different pattern bundles in all issue slots yields $|S_p|$ for the objective function.

The first constraint for a valid schedule is an assignment constraint, requiring that each operation node be scheduled in exactly one time step and exactly one issue slot:

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} y_{i,j,k} = 1, \quad \forall\, i : v_i \in V_{\text{pat}}. \tag{12}$$

The precedence constraint requires that each node be scheduled later than its predecessors. Unlike the problem in Section 4.2, it is not possible to schedule dependent nodes in the same cycle. If g_l is the time step for v_l we get

$$g_i < g_h, \quad \forall\, (i,h) : v_i \rightarrow v_h. \tag{13}$$

To express the time step g_l of a node l, all time steps and issue slots are scanned:

$$g_l = \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{l,j,k}. \tag{14}$$

Tranforming Eq. (13) to $g_i - g_h \leq -1$ and substituting with Eq. (14) yields the precedence constraint:

$$\begin{aligned} &\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{h,j,k} \leq -1, \\ &\forall\, (i,h) : v_i \rightarrow v_h,\ v_i, v_h \in V_{\text{op}}. \end{aligned} \tag{15}$$

In order to cover also the dummy nodes a second precedence constraint is needed:

$$\begin{aligned} &\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{h,j,k} \leq -1, \\ &\forall\, (i,h) : v_i \rightarrow v_{\text{exit}} \rightarrow v_{\text{entry}} \rightarrow v_h, \\ &v_{\text{exit}}, v_{\text{entry}} \in V_{\text{dmy}}. \end{aligned} \tag{16}$$

Finally, the timing constraints are taken into account. We derive the constraint similarly to Eqs. (9) and (10), with Eq. (14) for the scheduled time steps:

$$\begin{aligned} &\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{h,j,k} \leq t_{\max}, \\ &\forall\, (i,h) : v_i \xrightarrow{t_{\max}} v_h,\ v_i, v_k \in V_{\text{op}} \end{aligned} \tag{17}$$

$$\begin{aligned} &\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\ y_{h,j,k} \geq t_{\min}, \\ &\forall\, (i,h) : v_i \xrightarrow{t_{\min}} v_h,\ v_i, v_k \in V_{\text{op}}. \end{aligned} \tag{18}$$

This completes the formulation with objective function (11), and constraints (12) for assignment, (15) and (16) for precedence between operation nodes, and (17) and (18) for timing. The result of the optimization is the set of pattern bundles S_p.

In order to reduce the number of bundles in the set, we employ a method described in [15] that substitutes patterns by others. This method exploits the fact that a pattern can also implement simpler patterns by applying identity operands to operators in the pattern, thus bypassing the operators. The substitution relations between patterns are represented in an identity-operand graph (IOG).

An IOG can also be constructed for patterns in a pair. Therefore, a pair can implement combinations of simpler patterns that are part of the IOGs of the pair patterns. We use the IOGs of the patterns to find all pairs of simpler patterns that are dominated by a pair. Any pair in S_p that is dominated by another pair in S_p is removed from the instruction set.

We also construct the IOG library of the sequential patterns needed to cover the remaining operations that are not covered by any chosen parallel pair. From this IOG, we select all those patterns that are not dominated by any other pattern as instructions. The final instruction set for the ASIP consists of those sequential patterns and the chosen parallel pattern pairs. The entire design process is automated by feeding the ILP formulations to an ILP solver.

6 On-Chip Communication: Data-Push

The methods proposed in Sections 4 and 5 enable the automated design of highly-specialized small ASIP cores which are guaranteed to meet tight timing constraints. When several such ASIPs are integrated in a System-on-a-Chip, however, shared bus and memory accesses introduce an unpredictability that compromises the required deterministic performance. In this section we propose an on-chip communication scheme that solves this problem by again relying on the ASIP to meet tight timing constraints.

The scheme avoids data fetches from a central memory altogether. Instead, we use a model in which data is *pushed* to the ASIP's input registers as soon as it is available and data must be read before the register is overwritten with the next input. Similarly, the ASIP writes its results to the output registers in a write-and-forget fashion. The data in an output register is then pushed to the input of the consuming building block. We call this communication style a *data-push* model.

The requirement that the ASIP precisely times reads and writes to the I/O register. Is met by our design methodology for real-time ASIPs. The advantage of data-push communication is that it avoids any complexity associated with queuing, buffer management, and access to shared busses and memory.

In [16] a network processor (NP) is introduced that has functional units arranged in a pipeline so that one unit passes the results of its computations on to the next one by means of data-push communication. When a packet arrives from the network, the link interface writes the first word of the packet to the appropriate input register of a header parser and generates a signal to start header processing. The interface continues to write packet data to the same input register until the packet ends. The parser pushes each header field it extracts to an input register of the look-up processor, signaling the field type. In the same manner, results are passed from one unit to the next until the packet handling is completed.

Data-push communication is particularly well-suited for NPs: They require deterministic performance, and bus and memory bandwidth are the main performance bottlenecks in NPs. A data-push architecture addresses both problems by replacing the complexity of queue and buffer management between building blocks with a scheme that requires neither bus nor memory accesses. Furthermore, the data-push model provides full wire-speed processing and matches the streaming character of the network traffic.

Moving from a memory-centric to a data-push model involves a change in the way data is addressed. Figure 4 demonstrates this change for a network header structure. In the memory-centric model, an element in a data structure is accessed by adding an offset to the base address of the structure and reading from the resulting address. The offset corresponds to the position of the element in the structure and is therefore a *spacial address*. In the data-push model, by contrast, an element in the input data is accessed by waiting for it to occur in an input register. The number of cycles to wait from the signaled start of a data transmission corresponds to the position of the element in the input data and is therefore a *temporal address*.

7 Case study: Networking ASIP

In order to prove the concept of our design methodology in a real-life case, we applied our methodology to

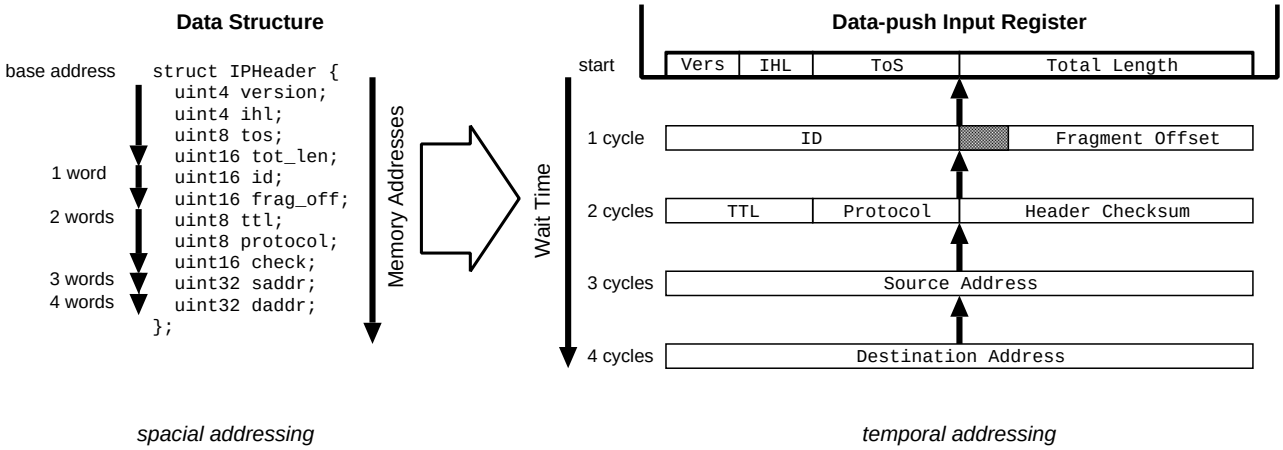

Figure 4: From spacial to temporal addressing.

a representative control-dominated ASIP: a parser for packet headers as a building block for an NP. The manual design and optimization of a header parser for network processing have been described in [4]. Figure 5 shows the interfaces of the parser. Protocol data is applied to the 32-bit input port, and a packet start is indicated by a delimiter flag. The flag starts the analysis of the packet header. Communication with the environment of the parser follows the data-push paradigm, i.e., the data words are expected to be available in an input register for only one cycle. The parser extracts the protocol fields that other building blocks in the NP need and writes the extracted data to the 32-bit output port together with a 4-bit ID that identifies the type of output data. The parser can issue two instructions in parallel in a VLIW fashion.

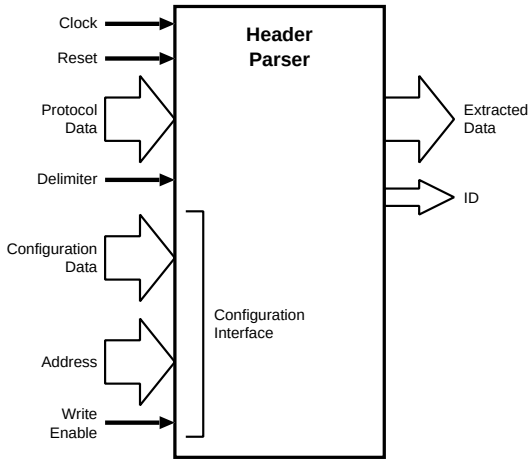

Figure 5: Header-parser interfaces.

The network protocols considered are versions 4 and 6 of the Internet protocol (IPv4, IPv6). As an example,

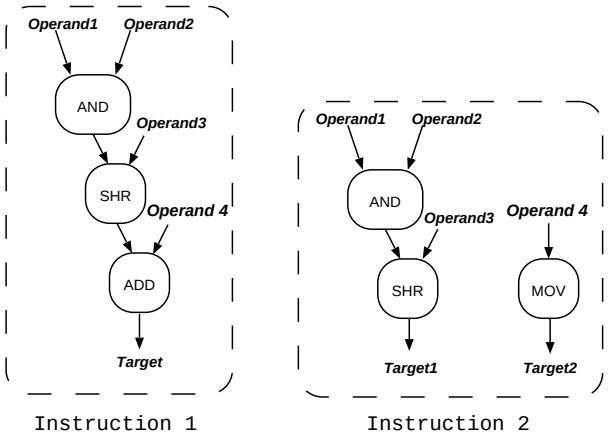

Figure 6: Automatically derived compound instructions.

Table 2 gives the relevant header fields to be extracted for IPv6, indicating the clock cycle in which a field occurs, its position in the 32-bit input word, and whether it is needed for processing within the parser or by an external building block. The clock cycles dictate the timing constraints that must be specified along with the application code. We have written this specification with the ANSI C timing constructs defined in [5].

The resulting optimal instruction set consists of only two compound instructions, shown in Figure 6. Combined with control instructions, this instruction set is sufficient to implement the benchmarks with the required timing.

The parser has been synthesized for an 18-μm technology, supporting data rates close to 10 Gb/s. The size of the parser, including a small instruction memory, is on the order of 0.45 mm^2, which demonstrates the area efficiency of the ASIP approach.

Table 2: Relevant header fields in IPv6.

Cycle #	Fields relevant internally	Fields relevant externally
1	–	Traffic Class [4–11], Flow Label [12–31]
2	Next Header [16–23]	–
3–6	–	Source Address [0–31]
7–10	–	Destination Address [0–31]
wait until NextHeader = layer-4 header	Next Header [0–7], HdrExtLen [8–15]	stored layer-4 NextHeader
wait for end of IP header ⇒ layer-4 header	–	TCP / UDP: Source Port [0–15] Destination Port [16-31]

8 Conclusions and Future Work

In this paper we have introduced the first ASIP design methodology for the control-dominated application domain. We have defined instruction-set synthesis for hard timing constraints as two consecutive optimization problems. The first step finds operation sequences that must be implemented as compound instructions to meet the timing constraints. The second reduces the number of required parallel instruction issues. We have formulated both problems as ILPs. The resulting ability to generate ASIPs that meet tight timing constraints has enabled us to propose the data-push paradigm, a low-complexity on-chip communication model. In a case study, we have demonstrated the relevance of the introduced concepts. Our next step will be to develop heuristics to solve both optimization problems more efficiently in order to enable the analysis of large sets of benchmark applications. Another area for future work will be to relax our assumptions on the processor architecture.

References

[1] Marnix Arnold and Henk Corporaal. Designing domain-specific processors. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES'01)*, pages 61–66, April 2001.

[2] Philip Brisk, Adam Kaplan, Ryan Kastner, and Majid Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of CASES 2002*, pages 262–269, October 2002.

[3] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Atomatic application-specific instruction-set extensions under microarchitecutral constraints. In *Proceedings of 40th DAC*, pages 256–261, June 2003.

[4] Gero Dittmann. Programmable finite state machines for high-speed communication components. Master's thesis, Darmstadt University of Technology, `http://www.zurich.ibm.com/~ged/`, 2000.

[5] Gero Dittmann and Andreas Herkersdorf. Fine-grained timing constraints for reactive systems in ANSI C. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004) – WIP Session*, pages 32–35, Lisbon, Portugal, December 2004.

[6] Barry Michael Pangrle and Daniel D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1098–1112, November 1987.

[7] T. C. Hu. Parallel sequencing and assembly line problems. *Journal of Operations Research*, 9(6):841–848, November 1961.

[8] Pierre G. Paulin and John P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–678, June 1989.

[9] In-Cheol Park and Chong-Min Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Proceedings of the 28th DAC*, pages 680–685, 1991.

[10] J. Lee, Y. Hsu, and Y. Lin. A new integer linear programming formulation for the scheduling problem in data-path synthesis. In *Proceedings of the IEEE ICCAD*, pages 20–23, November 1989.

[11] Robert A. Walker and Samit Chaudhuri. Introduction to the scheduling problem. *IEEE Design and Test of Computers*, 12(2):60–69, 1995.

[12] David C. Ku and Giovanni De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, Norwell, MA, USA, 1992.

[13] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1999.

[14] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[15] Gero Dittmann. Pattern libraries for fast searching and data-path sharing. In *Proceedings of the 3rd Workshop on Application Specific Processors (WASP)*, pages 76–83, Stockholm, Sweden, September 2004.

[16] Maria Gabrani, Gero Dittmann, Andreas Doering, Andreas Herkersdorf, Patricia Sagmeister, and Jan van Lunteren. Design methodology for a modular service-driven network processor architecture. *Computer Networks*, 41(5):623–640, April 2003.