

RZ 3484 (# 93950) 02/24/03  
Electrical Engineering 8 pages

# Research Report

## Multi-Layer Intermediate Representation for ASIP Design and Critical-Path Optimization

Gero Dittmann and Andreas Herkersdorf

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

### LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at <http://domino.watson.ibm.com/library/Cyberdig.nsf/home>.

**IBM** Research  
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

# Multi-Layer Intermediate Representation for ASIP Design and Critical-Path Optimization

Gero Dittmann and Andreas Herkersdorf  
IBM Research, Zurich Research Laboratory  
8803 Rüschlikon, Switzerland  
{ged,anh}@zurich.ibm.com

## Abstract

Existing methods for the design of application-specific instruction set processors are tailored to the domain of *data*-dominated applications, which are characterized by extensive computations and few branches. In this paper we propose to combine a selection of current data-dominated design methods to form an integrated design methodology.

Design methods for the *control*-dominated area require an extended set of information on application characteristics in order to effectively handle many branches intersected with only small data-flow blocks. Therefore, we introduce a multi-layer application representation that captures control, data, and timing dependencies as well as further annotations, to form a basis for transferring the methodology to the control-dominated domain. Based on this representation, we propose a method to resolve scheduling conflicts between tight deadlines. This method demonstrates the potential of the new representation.

## 1 Introduction

Most research publications on application-specific instruction set processors (ASIPs) concentrate on the design of digital signal processors (DSPs). Some publications on micro-controllers and Prolog processors exist [9], but they tend to focus more on implementation details than on algorithms for automatic instruction-set construction.

DSP architectures are data-dominated in that their applications have long arithmetic sections between control-flow boundaries, i.e. between branches.

Examples of ASIPs in the control-dominated domain are the building blocks of network processors (NPs), such as a header parser [7], which extracts fields out of packet headers, or a *protocol engine*, which implements protocol FSMs. Both tasks consist mainly of branch decisions with only few computations. Implementation of such FSMs has been inves-

tigated for ASIC high-level synthesis (HLS), mainly for automotive applications [4], but not for programmable cores. The main difference between these two approaches is the fact that HLS optimizes the circuits for a single application whereas an ASIP must support a variety of applications, including future applications that have not been specified at design time, which introduces a flexibility factor that is hard to quantify.

On a higher abstraction level of a system-on-a-chip (SoC), considerable research has been conducted in the communications area. An approach to quickly implement new communication protocols in a mixed hardware/software system can be found in [15]. The system is not supposed to run any other protocol after implementation, but optimizes the cost-performance trade-off. Approaches for more flexible NPs, but also on a rather coarse-grained SoC level, are described in [3, 16, 21]. In contrast, we focus on the computer-assisted generation of a fine-grained instruction set for a specialized processing core as one building block of an SoC.

In this paper we contribute three concepts that form the basis for further research: First of all, in Section 2 we introduce an integrated approach for the design of ASIPs. Section 3 summarizes existing solutions for individual steps of this design methodology. Secondly, in Section 4 we present our application representation that is particularly suited for the control-dominated domain. Finally, the use of this representation is demonstrated in Section 5 by an optimization technique we call *branch postponing* that resolves scheduling conflicts between deadlines. Section 6 concludes the paper.

## 2 ASIP Design Methodology

To derive an ASIP from applications in the target domain, we combine a number of techniques into a design methodology for ASIPs as shown in Figure 1. The designer specifies a suite of applications or parts of applications that are characteristic for the target application-domain. This is done in a high-level language, such as C. A compiler front-end trans-

lates this specification to an intermediate representation (IR), which can usually be visualized as a graph, e.g. a control/data flow graph (CDFG), of basic instructions, such as add, subtract, shift, multiply, divide, etc.

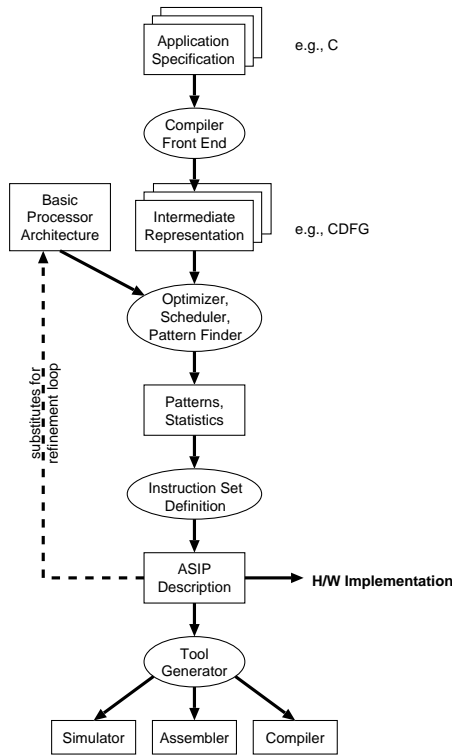


Figure 1: ASIP design methodology

Based on an architecture template, this graph is optimized, employing methods found in the compiler literature [1], and graph nodes are scheduled into time steps. Recurring instruction patterns are identified that are candidates for hardware implementation to render code execution more efficient. Optimizations, scheduling, and pattern finding have a significant impact on each other and are thus interwoven. The result of this process is a set of candidate patterns along with statistical information about their occurrence and their benefit. This is also the point where information from the individual applications in the set is merged because the value of a pattern is independent of the application in which it appears. Based on the statistics, patterns are selected to be implemented as instructions, and a processor description is generated.

The complete methodology can be iterated by feeding this processor description back into the pattern finder. Once the designer is satisfied with the outcome, the description is implemented. Retargetable tool suites are used to quickly build a development environment around the processor, including

simulator, assembler, compiler, and debugger [13]. A recent development is that tools can even be generated automatically for the new processor from a formal processor description [14, 8].

### 3 State-of-the-Art Ingredients

In this section we give an overview of existing methods for individual steps of our design methodology. A compiler front-end transforms programs from a high-level language to an IR. Possible types of IRs are the subject of Section 3.1. Innumerable ways to optimize programs and schedule operations can be found in the compiler literature [1]. Section 3.2 summarizes strategies for pattern finding in programs. Owing to the lack of automatic estimates of implementation complexity and tuning opportunities, the definition of a new instruction set is still a largely manual task. The design of compilers for new ASIPs is the subject of Section 3.3.

#### 3.1 Intermediate Representation

A crucial point for the design methodology is the intermediate representation (IR) of applications, which is analyzed to find optimizations and instruction patterns. Restrictions of the IR inadvertently result in deficiencies for the entire process because the effectiveness of optimizations depends on the set of available information.

The information that an IR for our target domain must convey is

- control flow as well as data flow;
- concurrency *and* sequentiality;
- timing constraints, and
- as much of the application developer’s expertise as possible.

An overview of the models commonly used in hardware/software co-design (HSC) at system level is given in [5]: FSMs, discrete-event systems, Petri nets, data-flow graphs, communicating processes, and synchronous/reactive models, as well as several derivatives thereof.

In [4], an FSM notation specialized for control-dominated models is introduced, called *co-design FSMs* (CFSMs). Furthermore, for the software part a directed acyclic graph (DAG) model is reduced to a *software graph*, which is supposed to allow some special optimizations impossible with control-flow graphs. The CFSM approach was then extended into *Function Flow Graph* (FFG) models [17]—an FSM with data manipulation code in each node, where the FSM represents the control flow and the code represents the data

flow. An FFG can be described in a *C-Like Interchange Format (CLIF)*. In [18], an FFG is annotated with designer-provided additional information—mainly visit probabilities of nodes—which makes it an *attributed FFG (AFFG)*. Several optimization strategies for those graphs are suggested in [17, 18].

### 3.2 Pattern Finding

A simple approach for ASIP instruction set design proposed in [20] is to analyse the data-flow graphs (DFGs) in a CDFG to find frequently recurring instruction *sequences*. Appropriate hardware resources that implement these sequences are then manually added to speed up program execution, and the code is modified to make use of the new resources. These two steps, sequence analysis and adding corresponding resources to the hardware, are iterated until the result is satisfactory for the designer.

The approach presented in [10] does the same considering *parallel* operations rather than sequences, and is targeted for pipelined processors. Parallel operations in DFGs are scheduled into time steps, and operations in the same time step form an instruction. A simulated annealing algorithm is then used to modify the original operation scheduling to find better instruction sets. Moreover, different operand encodings are tried out in order to meet a given instruction-size constraint.

Instead of starting from the most simple instruction set, other approaches are based on existing processor cores, as described in e.g. [9], in an attempt to keep design cost and time-to-market low. These cores are then manually extended with application-specific instructions to speed up critical code sections.

In [2], parts of the above approaches are combined: Existing processors are extended for an application domain by finding two-dimensional patterns (i.e. consisting of *sequential and parallel* operations) that share at least one operand and implementing them as special instructions. Applications are not represented by the compiler output directly but by execution traces, thus allowing the detection of patterns across control-flow boundaries, and a better estimate of their frequency of occurrence.

The pattern-matching algorithm that works on these traces develops its pattern library on the fly: It starts with a library of basic operations and then iteratively adds all possible combinations of each operation node with its neighbors, i.e. other nodes that share at least one operand with it. This library is then used to cover the application graph such that each operation is covered by exactly one pattern. A variation of dynamic programming is employed to minimize the implementation cost of the cover.

The patterns found are sorted by the number of times they occur in the application graphs and by their popularity for covering. From this list, patterns are manually selected, grouped, and implemented.

### 3.3 ASIP Compiler Design

The design of a compiler for an ASIP is tightly coupled with the design of the ASIP itself because the approaches used in instruction-set generation are similar to instruction selection in compilers. Furthermore, the automatic generation of compilers from processor descriptions is an active field of research [14, 8].

In [22], the implementation of a C compiler for a particular network processor is described. The focus is on operations on variable-length bit-vectors that are not aligned on register boundaries and may even span across two registers. Also, support for arrays of bit vectors is proposed.

## 4 Multi-Layer Intermediate Representation

In [2] it was found that the compiler output is not a good application representation to work on because it provides no information about the probability with which individual branches are taken. The consequence was to use execution traces instead. In order to reveal even more of the semantics lost en route from the developer’s expertise to the assembly code, we go to a higher abstraction level and introduce a program representation that allows programmers to express more of their application expertise, such as ranges for loop counters and timing constraints.

We start with the intermediate representation (IR), which is the data structure on which compilers work, because it determines the set of possible methods to derive information about the application, possible optimizations and ultimately about advantageous processor structures. From an IR that allows all desired methods it can then be concluded which annotations a high-level language has to provide.

IRs have a graph structure with nodes and directed edges that represent dependencies between nodes. Nodes and edges are annotated with information that is useful for the process. Dependencies that should be expressed for control-dominated applications are

- data dependencies for computations using results of other computations,
- control dependencies that determine the control-flow through an application, and

- time dependencies to express timing constraints and synchronization with the environment.

*Data dependencies* are expressed using data-flow graphs (DFGs), where nodes represent the operations, incoming edges the operands, and outgoing edges the results.

*Control dependencies* are represented in a Petri-net notation. The nodes (“places”) represent blocks of operations, i.e., they contain DFGs. The edges (“relations”) show where the control flow (“token”) leads, and can be unconditional or conditional. Conditional edges originate in a data node of a DFG internal to a control node. The false-edge is taken if the result of the data node is zero. The true-edge is taken if the result is not equal zero. Moreover, the model not only allows the expression of if-then constructs but also of case statements. For this purpose, the edges are annotated with the value for which they are taken. A default edge must be provided to prevent dead locks.

To express parallel threads of control in a Petri-net model, the control flow can split up at bars (“transitions”) with more than one edge leaving a bar. Threads are joined when their control edges enter the same bar. Control only proceeds past a bar when the control flow of a thread has arrived at each incoming edge. This provides synchronization between threads. The control flow of a program starts at a START bar and ends at an END bar.

As the DFGs in control nodes (Petri places) rely on computation results of other control nodes, data dependencies also exist between control nodes, forming a second level of DFG. This *meta-DFG* overcomes the imperative to store *all* results of computations at *every* control-flow boundary in either registers or memory, and allows optimization runs to move computation nodes across control-flow boundaries. This is particularly useful for control-dominated applications in which the size of DFGs in control nodes is often very small and only their extension across control-flow boundaries will allow an effective optimization.

A control node, however, may be reached by more than one control edge and each of these control edges may require a different set of meta-DFG edges to be used for the computation in the control node. Hence, sources must be selectable by the control edges. This is represented by a multiplexer consisting of one box per arriving control edge. Each box joins a control edge with the meta-DFG edges it requires.

Finally, a way of representing *time dependencies* are output transition graphs (OTGs) as introduced for controller FSMs in ASICs [12], where edges are annotated with the minimum and maximum time between nodes and scheduled nodes are annotated with the determined time step, given for instance in processor cycles.

We combine DFG, Petri net, meta-DFG, and OTG into a *multi-layer IR* with a single start node and a single end node.

Existing optimization runs that have been proposed for one of the original graphs can still be used by working only on the corresponding layer. Figure 2 shows the different layers in a simple example graph.

As graph operations need more information, nodes and edges can be further annotated, e.g., scheduled data nodes will have an associated time step, conditional control edges may be annotated with the minimum and maximum number of times they are taken in one run, or DFG edges may have ranges of legal values imposed on the variables they represent.

Sequentiality is expressed with data-dependency and timing edges. Concurrency can be found on several levels:

- Nodes in a DFG that cannot reach the other (obeying the direction of edges) can be executed in parallel.
- DFGs in the same control node represent parallelism.
- An control edge can split up at a bar and lead to multiple control nodes.

Hence, the multi-layer IR fulfills all requirements postulated at the beginning of this section.

#### 4.1 Nodes for Data-Dependent Delay

In control-dominated applications timing constraints are often data-dependent, i.e., the required time between two nodes is determined by a system input. One such problem in a network processor is the task of finding the beginning of a TCP packet header after a variable length IP header. The length is encoded in a header field and this value corresponds to the number of input words to bide before the TCP header appears at the network interface.

In order to provide an expression for this dependency, we further extend the multi-layer IR by a type of operation node that connects the DFG layer with the timing layer. We call this node a *delay node*. It has one DFG edge as an input whose value determines the delay that the node represents, given in the same time unit as the edges in the timing layer of the multi-layer IR. Furthermore, to be meaningful, a delay node must have an incoming and an outgoing timing edge because its purpose is to provide a particular delay between two other nodes.

The delay node is a virtual node in that it does not translate directly to a primitive processor instruction. Instead, it is transformed into one of two possible implementation types. It can be implemented

1. entirely in software by moving a start value into a register and then explicitly decrementing this register in appropriate intervals and branching when it reaches zero, or

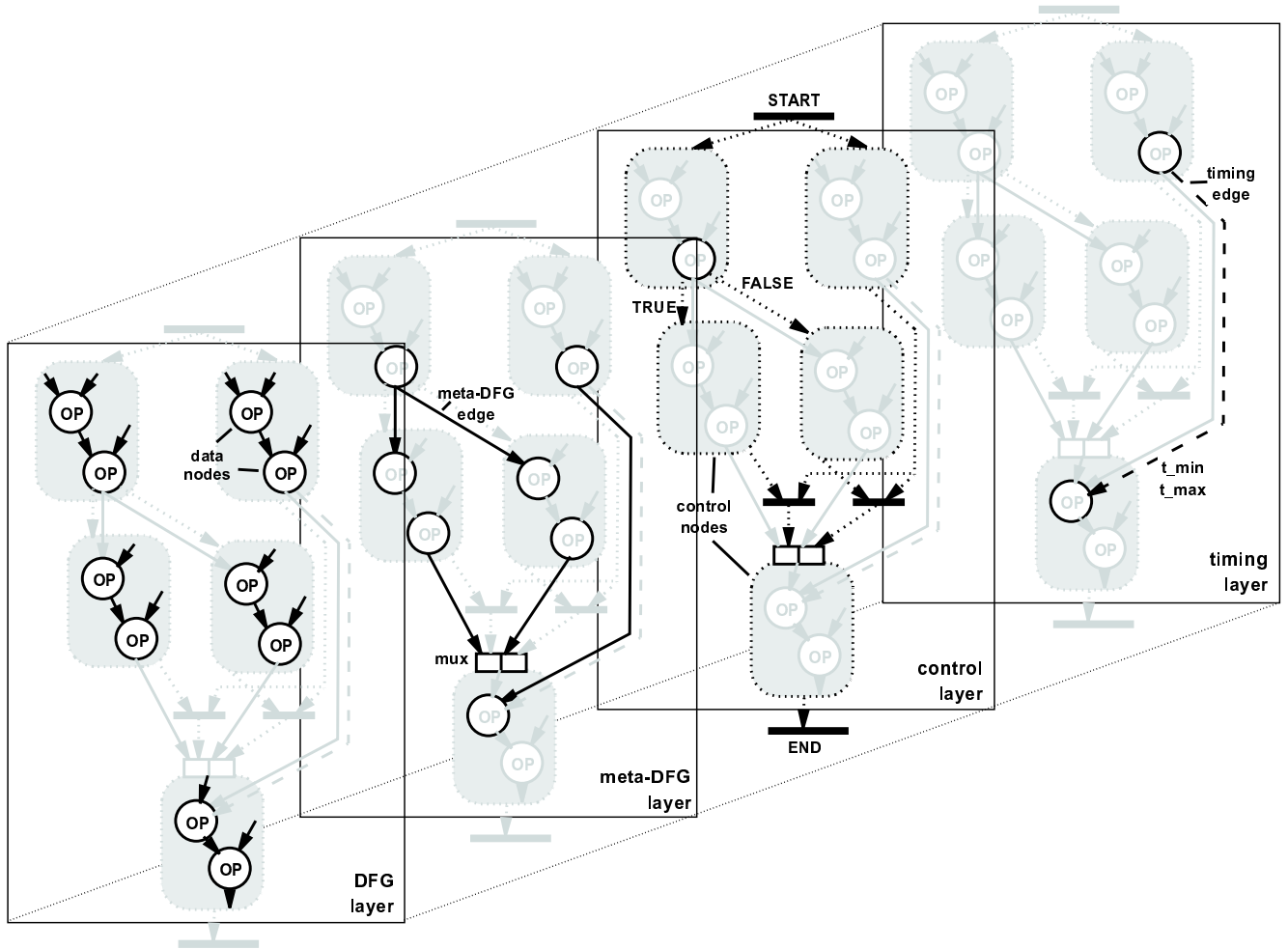


Figure 2: Example of multi-layer IR graph

2. partially in hardware by providing a counter register that is implicitly decremented by a constant value—typically by one—and compared to zero in each clock cycle. When the counter reaches zero then the program counter of the processor is set to an address that is stored in a second special register.

The software implementation may require quite a number of instructions in the application code for the repeated subtraction and test for zero. Each additional instruction complicates the instruction scheduling process.

In comparison, the hardware solution relies on additional infrastructure. Moreover, a counter can be used for only one delay node at a time. Starting the counter in the application code requires at most two *move* instructions—one to set the start value of the counter and one to set the jump address in the second register. Writing the start value to the

counter, however, needs to be scheduled in precisely the cycle required by the timing edges that lead to the delay node. Otherwise, the counter would not go off at the intended point in time.

We can achieve the freedom to schedule the counter start earlier or later by introducing another *add* or *subtract* node, respectively, to adjust the start value accordingly. This additional node may be arithmetically merged with other nodes in the delay computation by appropriate optimization methods.

Note that the adjustment value depends on the final scheduling of the instruction that starts the counter. Hence, the value can only be determined after the final instruction scheduling and might then even be zero. In order to perform the correct adjustment the scheduler needs to be aware of operations that implement delay nodes in the application. It is

the scheduler which decides whether an adjustment node is introduced or not.

The scheduling freedom of the counter start furthermore depends on the minimum possible start value of the counter. The minimum value determines the time after which the counter must be tested for zero for the first time and corresponds to the latest possible start time of the counter—even with adjustments. This calls for the afore-mentioned value ranges to be annotated with DFG edges.

For a software implementation of a delay node, an enumeration of the legal delay values offers an optimization opportunity. Gaps between the values correspond to scheduling slots in which the register used for the count-down does not have to be decremented nor tested for zero. To compensate, it only has to be decremented by a higher value later on.

In conclusion, the delay node offers the application programmer an additional abstract expression and enables the instruction scheduler to select an optimal implementation strategy for the expression.

## 5 Branch Postponing

Once an application suite of the target domain has been captured in the multi-layer IR it can be optimized and scheduled to meet timing requirements. By means of a novel optimization algorithm we now demonstrate how the combined information in the multi-layer IR can be used to resolve scheduling conflicts that would otherwise inhibit the timely execution of an algorithm.

In data-dominated systems, such as DSPs, processing often starts with receiving a sample of data and ends with sending out a resulting sample [6]. Between 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.

In control-dominated real-time systems, such as NPs, often there is not only one deadline at the end of a run but there are many I/O interactions with the environment and many of them have a deadline associated with them.

In a high-speed network processor, for instance, memory bandwidth is a major bottleneck. One way to relieve this problem is to process packet headers on-the-fly as they come in from a link (*data-push processing*) instead of retrieving them from memory for each processing step. But this means that every header word that contains fields to be processed has a deadline associated with it because it has to be processed—or at least saved to a *stable* register—before being overwritten by the next incoming header word.

With multiple deadlines in short sections of code the need for fine-granular timing optimization arises. An example of a problem that can occur is given in Figure 3.

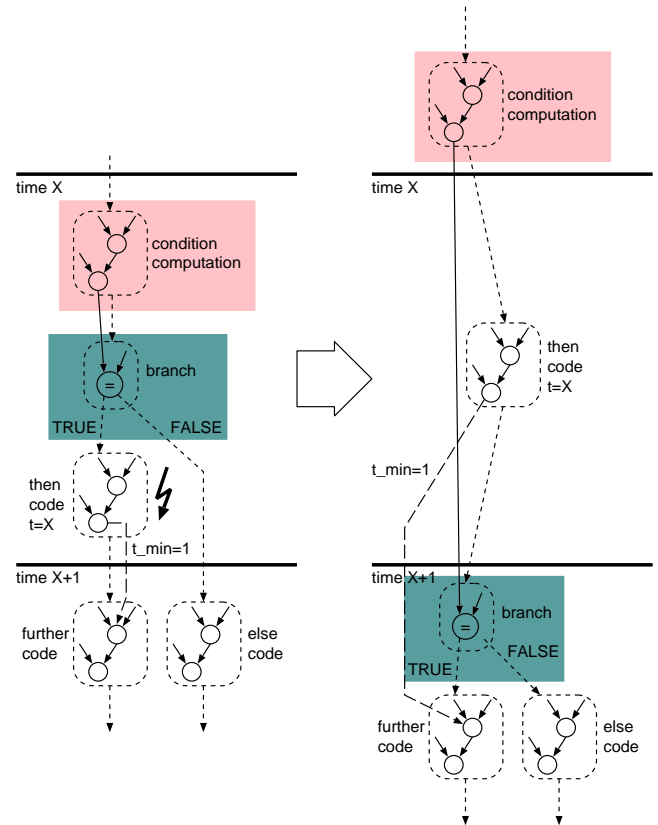


Figure 3: Branch postponing

On the left, condition computation, branch, and then-code are all scheduled in the same time step  $X$ . Assume that the then-code alone needs a full time step to be computed. As it has the annotated requirement to be scheduled in time step  $X$ , e.g. because of input data that only occurs in this particular cycle, the other control nodes must be moved to another time step.

The technique we use to achieve this has similarities with speculative execution in that it changes the execution order of a conditional branch and following code. Speculative execution does this to fill processing slots before the branch in order to minimize the execution time of the average case and the critical path through the program. For choosing the right code to speculate, branch prediction is employed.

In contrast, branch postponing improves the schedulability, not the average execution-time. It might even grow the critical path through the else-code. But it allows to schedule code that otherwise could not meet its timing constraints. This is done independently of what the average case is and hence, no assumptions are made on branch probabilities.

The first step to solve the problem in Figure 3 is to move the condition computation to the preceding time step, as depicted on the right. Assume that time step  $X - 1$  is now fully occupied. This means that the branch cannot be moved to the preceding time step as well. Then the only remaining solution is to move the branch to time  $X + 1$ . But that would mean to move the branch after a code section that should only be executed if the branch is actually taken, as also shown on the right.

This transformation does not change the result of the program if the then-code is not “harmful”, i.e., it does not change any data that is used in the else-branch. This condition is met if

- no output to the ASIP environment occurs in the DFG nodes of the then-code because this communication is part of the program result that should not be altered by the transformation;
- no memory writes occur in the DFG nodes of the then-code because any data written might be read in the else-branch. This criterion can be further relaxed by examining memory accesses more closely and comparing write addresses in the then-code with read addresses in the else-branch. This can, however, be a complex task because of the memory alias problem of two different expressions denoting the same memory location.

In control-dominated applications this situation occurs frequently, for instance, if the branch tests a termination condition and the else-branch starts an alternative algorithm that does not use any result from the first algorithm because it handles a special case for which the first algorithm is not suitable.

## 5.1 Applicability and Relevance

For an estimation of the relevance of branch postponing in a real world example, we compiled the header-compression code in [11] with the gcc compiler for IA-32 processors and isolated the compress and uncompress routines in the assembly code. Header compression is a typical control-dominated application. We found that 9% of all assembly instructions are conditional branches, each of which represents a potential scheduling problem that branch postponing can solve.

For a closer examination, we implemented the compress routine in the multi-layer IR. The target ASIP is a protocol engine with a data-push architecture as part of a network processor. The compress routine handles only common-case packets and delegates error handling to another processing entity. We found that 33% of the conditional branches in the program are of the above mentioned termination condition type that branch out of the algorithm between tight deadlines.

This is a typical situation where branch postponing ensures schedulability within the timing constraints. It can, however, be applied also to the remaining conditional branches.

Note that branch postponing adds only little to the critical path in the else-branch, because the else-code would in any case have to wait for time  $X + 1$  to arrive owing to the given minimum time distance to the then-code of 1. The time added by moving the branch to the same time step is not critical in many cases, such as in the above-mentioned case when it terminates the algorithm. The gain, on the other hand, is significant as it allows the then-code to be scheduled, which otherwise could not be accommodated.

Branch postponing makes use of all four layers of the multi-layer IR:

- The control layer represents the branch.
- The timing layer expresses the deadline problem.
- The DFG layer is used to analyse whether the then-code block is harmful or not.
- Instead of assigning a variable name to each computed value in a control node when leaving the node, like in a standard CDFG, the meta-DFG layer makes data dependencies between control nodes obvious. Therefore, no possible conflicts have to be examined when moving the branch.

This demonstrates the potential of combining information in the annotated multi-layer IR.

## 6 Conclusions and Future Work

In this paper we have proposed a way to link ASIP design methods to form an integrated design methodology. As part of the methodology we have combined several approaches for application representation and extended them with new expressions to arrive at our multi-layer IR that allows us to use optimization techniques defined for one of several graph types. Finally, we presented a novel optimization for branches between short-term deadlines that demonstrates the power of the multi-layer IR by exploiting additional scheduling freedom thanks to timing information.

The next step will be to implement the IR and methods working on it, including branch postponing, in a compiler framework, such as the Stanford University Intermediate Format (SUIF) [19], in order to further quantify the relevance for real-life cases.

Future use of the multi-layer IR will include pattern finding across control-flow boundaries and patterns that include branches, which will make the approach even more valuable

for the control-dominated domain, which features a large number of branches.

For pattern finding in general it would be helpful to find a better figure of merit than the sum of matching and covering contributions—preferably one that considers implementation cost and the latency of custom operations. This might lead to the inclusion of two-cycle operations for possible pattern implementation.

Once the set of annotations required in the multi-layer IR has been consolidated, the application specifier must be enabled to express this information, including timing constraints, in a high-level language that can then be compiled to the annotated multi-layer IR.

The goal is to develop a system that relieves the ASIP designer of tedious and complex tasks while still providing sufficient control of the process to optimize the result.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proc. of CODES'01*, pages 61–66, April 2001.
- [3] M. Benz. An architecture and prototype implementation for TCP/IP support. In *Proc. of the TERENA Networking Conf. 2001*, May 2001.
- [4] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware/software co-design of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.
- [5] L. A. Cortés, P. Eles, and Z. Peng. A survey on hardware/software codesign representation models. Technical report, Dept. of Computer and Information Science, Linköping University, June 1999.
- [6] H. De Man, I. Bolsens, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest. Co-design of DSP systems. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 75–104. Kluwer Academic Publishers, Dordrecht, 1996.
- [7] G. Dittmann. Programmable finite state machines for high-speed communication components. Master's thesis, Darmstadt University of Technology, [http://www.zurich.ibm.com/~ged/HeaderParser\\_Dittmann.pdf](http://www.zurich.ibm.com/~ged/HeaderParser_Dittmann.pdf), 2000.
- [8] F. Engel, J. Nuhrenberg, and G. P. Fettweis. A generic tool set for application specific processor architectures. In *Proc. of CODES 2000*, pages 126–130, May 2000.
- [9] M. Gschwind. Instruction set selection for ASIP design. In *Proc. of CODES'99*, pages 7–11, May 1999.
- [10] I.-J. Huang and A. M. Despain. Generating instruction sets and microarchitectures from applications. In *Proc. of ICCAD-94*, pages 391–396, November 1994.
- [11] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. IETF RFC 1144, February 1990.
- [12] J. A. Nestor and V. Tamas. Exploiting scheduling freedom in controller synthesis. In *Proc. of the Int'l Workshop on High-Level Synthesis*, pages 74–86, November 1992.
- [13] P. G. Paulin, F. Karim, and P. Bromley. Network processors: A perspective on market requirements, processor architectures and embedded s/w tools. In *Proc. of DATE 2001*, pages 420–429, March 2001.
- [14] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA: Machine description language for cycle-accurate models of programmable DSP architectures. In *Proc. of DAC'99*, pages 933–938, June 1999.
- [15] J. H. Schiller and G. J. Carle. Semi-automated design of high-performance communication subsystems. In *Proc. of HICCS'98*, pages 273–282, 1998.
- [16] H. Shimonishi and T. Murase. A network processor architecture for very high speed line interfaces. *Journal of Communications and Networks*, 3(1), March 2001.
- [17] B. Tabbara, A. Tabbara, and A. Sangiovanni-Vincentelli. Hardware and software representation, optimization, and co-synthesis for embedded systems. Technical Report UCB/ERL M00/7, University of California at Berkeley, Electronics Research Laboratory, January 2000.
- [18] B. Tabbara, A. Tabbara, and A. Sangiovanni-Vincentelli. Task response time optimization using cost based operation motion. In *Proc. of CODES 2000*, pages 110–114, May 2000.
- [19] The Stanford SUIF Compiler Group. <http://suif.stanford.edu/>.
- [20] J. van Praet, G. Goossens, D. Lanner, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proc. of the IEEE/ACM Int'l Symp. on High-Level Synthesis*, pages 11–16, May 1994.
- [21] S. Virtanen, J. Lilius, and T. Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proc. of the 18th IEEE NorChip Conf.*, November 2000.
- [22] J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *Proc. of the ACM LCTES'2001*, pages 155–164, June 2001.