Diss. ETH No. 16201

On Instruction-Set Generation for Specialized Processors

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
GERO DITTMANN
Dipl.-Ing.
born March 15, 1974
citizen of Germany

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Paolo Ienne, co-examiner
Prof. Dr. Andreas Herkersdorf, co-examiner

2005

Acknowledgments

First of all, I would like to thank my advisor, Lothar Thiele, for giving me the chance to pursue a Ph.D. at ETH Zurich while working at IBM, and for encouraging me to delve into this fascinating field of synthesis methods for embedded systems. The course on hardware/software co-design by Marco Platzner provided my with an invaluable basis for my research. At an advanced stage, my co-advisor, Paolo Ienne, was an important influence, shaping my work by asking all the right questions.

The IBM Zurich Research Laboratory (ZRL) is a tremendously challenging and inspiring work environment. I want to thank all the people who make this lab such a special place. I am most indebted to Andreas Herkersdorf and Ton Engbersen who were my managers and mentors at ZRL. They have been an enormous source of inspiration and motivation. I very much appreciate the support by the lab's technical and administrative staff. I am particularly grateful to Charlotte Bolliger for proofreading my papers and teaching me a lot about the subtleties of the English language.

Thanks to the technical community for receiving me so openly. Special thanks go to Gerhard Fohler and Peter Puschner for taking the time to discuss my ideas on real-time scheduling in great length. I also want to thank my co-authors on papers and patents, especially Paul Hurley for helping me cope with integer linear programs.

On a less technical note, I express my utmost gratitude to my office mates, Sonja Buchegger, Mark Verhappen, Silvio Dragone, as well as the plants Alice and Bob, and to my unfailing lunchtime companions for sedulously discussing—and solving—all major questions in politics, art, literature, philosophy, and bicycle repair. I thank all my friends, inside and outside ZRL, for being there with encouragement or constructive doubt—whichever was appropriate.

Finally, I am infinitely grateful to my parents and sister without whose moral support this book would never have come into existence.

Abstract

Owing to the ever-decreasing feature size of today's semiconductor processes, the cost of a mask set has already 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 gradual trade-off between the flexibility of general-purpose processor cores and the performance of hard-wired logic can be achieved with *application-specific instruction-set processors* (ASIPs).

Many automated ASIP design methods found in the literature today employ a library of patterns that represent potential specialized instructions. As the libraries tend to grow large their access times become a critical factor. However, no attempts have yet been made to speed up these searches in the pattern libraries. Furthermore, there are no methods available to recognize and exploit structural similarities between patterns.

Another deficiency in today's ASIP design methodologies is their exclusive focus on the data-dominated domain characterized by computation-intensive applications such as digital signal processing. This focus entails a lack of methods for control-dominated domains such as network processing. These domains are characterized by branch-intensive applications with fine-grained timing constraints imposed by frequent interactions with the ASIP environment. The major challenge here is not to speed up the over-all runtime of the applications, 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 thesis we propose a hierarchical organization for pattern libraries that removes the dependency of search times on the library size. In this way, much larger libraries can be handled which removes the need for heuristics to prune patterns from the library. Exact methods become possible. In our experiments we found that searches in our structure are orders of magnitude faster than in a linked-list library. Furthermore, we introduce a method that employs identity operands to find synergies between similar patterns. These similarities can be exploited to achieve leaner instruction sets and for data-path sharing.

On top of these library structures, we propose the first integrated ASIP design methodology for the control-dominated domain. We introduce novel methods to specify fine-grained timing constraints in ANSI C, to include them in an intermediate representation that facilitates compiler optimizations, and to derive an instruction set that enables the ASIP to meet the timing constraints. We present a case study that demonstrates the feasibility of our methods and the quality of the results.

Zusammenfassung

Die stetig schrumpfenden Strukturgrößen moderner Halbleiterprozesse haben dazu geführt, dass die Kosten für einen Belichtungssatz bereits eine Million Dollar überschreiten. Damit sich diese Investition auszahlt, muss ein Design vielfältig einsetzbar sein. Die erforderliche Flexibilität wird üblicherweise mit Hilfe von programmierbaren Bausteinen erreicht. Eine feinstufige Abwägung zwischen der Flexibilität von General-Purpose-Prozessoren auf der einen Seite und der Leistungsfähigkeit von fest verdrahteten Schaltungen auf der anderen Seite wird ermöglicht von *application-specific instruction-set processors* (ASIPs).

Viele der veröffentlichten automatisierten ASIP-Design-Methoden bedienen sich einer Bibliothek von Mustern, die Kandidaten für Spezialbefehle darstellen. Da diese Bibliotheken sehr groß werden können, spielt ihre Zugriffszeit eine entscheidende Rolle. Dennoch wurde bisher nicht untersucht, wie die Suche in einer Musterbibliothek beschleunigt werden kann. Darüberhinaus sind keine Methoden bekannt, um Gemeinsamkeiten in der Struktur zweier Muster zu erkennen und auszunutzen.

Eine weitere Einschränkung heutiger ASIP-Design-Methodologien ist ihre ausschließliche Beschäftigung mit der Klasse der rechenintensiven, datendominierten Anwendungen wie digitaler Signalverarbeitung. Es fehlt daher an Methoden für kontrolldominierte Anwendungsklassen wie der Verarbeitung von Netzwerkprotokollen. Diese Klassen sind gekennzeichnet durch Anwendungen mit vielen Verzweigungen und detaillierten Zeitanforderungen, die von häufigen Interaktionen mit der Umgebung herrühren. Unter diesen Bedingungen besteht die Herausforderung nicht darin, die Gesamtlaufzeit der Anwendungen zu verringern, sondern sämtliche Zeitanforderungen zu erfüllen. Dies kann erreicht werden mit Hilfe von Spezialbefehlen, die die Ausführung von zeitkritischen Pfaden beschleunigen.

Diese Dissertation führt eine hierarchische Organisation für Musterbibliotheken ein, die die Abhängigkeit der Suchgeschwindigkeit von der Größe der Bibliothek aufhebt. Diese Eigenschaft macht wesentlich größere Bibliotheken beherrschbar, wodurch Heuristiken zum Verwerfen von Mustern vermieden werden können. Exakte Methoden werden möglich. Experimente zeigen, dass das Suchen in dieser Struktur um Größenordnungen schneller ist als in einer Bibliothek, die als verkettete Liste aufgebaut ist. Weiterhin wird eine Methode vorgestellt, die mit Hilfe von Identitätsoperanden Synergien zwischen sich ähnelnden Mustern findet. Die Ähnlichkeiten bieten die Möglichkeit, schlankere Befehlssätze zu erreichen und Datenpfade mehrfach zu nutzen.

Aufbauend auf diesen Bibliotheksstrukturen wird die erste integrierte ASIP-Entwicklungsmethodologie für kontrolldominierte Anwendungsklassen vorgestellt. Es werden neuartige Methoden entwickelt, um detaillierte Zeitanforderungen in ANSI-C zu spezifizieren, sie in eine Graphendarstellung für Compileroptimierungen einzubinden und daraus schließlich einen Befehlssatz abzuleiten, der es dem ASIP ermöglicht, die Zeitanforderungen einzuhalten. Abschließend wird anhand einer Fallstudie die Machbarkeit der Methoden und die Qualität der Ergebnisse demonstriert.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter we motivate the design of specialized processors in general and the introduction of a new design methodology for control-dominated processors in particular. In Section 1.1 we describe the role of specialized processors in systems on a chip and how they compare to general purpose processors and to hardwired logic in this environment. Related work on design methods for specialized processors is presented in Section 1.2. In Section 1.3 we sketch our new methodology for the control-dominated domain and point out its challenges. Finally, Section 1.4 gives an overview of the remaining chapters of this thesis.

1.1 The Case for Specialized Processor Cores

1.1.1 Systems on a Chip

A major challenge the semiconductor industry is facing today is being discussed under the name *productivity gap*: Advances in process technology provide an ever increasing number of available transistors per design while, on the other hand, chip designers are struggling to handle the complexity of such systems. As a consequence, the designed application-specific integrated circuits (ASICs) fail to take advantage of what the technology offers.

A promising approach to bridge the gap is the *System-on-a-Chip (SoC)* design principle: A complex ASIC is composed of several small processing elements connected by a communication infrastructure. The processing elements can be taken from a library of predefined, optimized building blocks. If the communication infrastructure is standardized by a specified SoC *platform* the building blocks can be plugged together easily.

The SoC approach boosts the designer's productivity for several reasons:

- A new abstraction level is introduced which increases the level of complexity that can be handled.
- The use of component libraries fosters the reuse of design elements. Since the components in the libraries have been tested and used before, the approach reduces the probability of design errors.
- The regularity in an SoC architecture enables support by high-level design tools and has already given rise to pertinent specification languages such as SystemC [OSCI02].

The resulting productivity boost increases the chances for a first-time-right design and significantly reduces the time to market.

Another severe problem in chip design are non-recurring engineering costs (NRE). As the feature size decreases the cost of masks grows rapidly. For 90-nm processes Sematech expects mask-set costs to exceed one million dollars. The NRE of a design is better

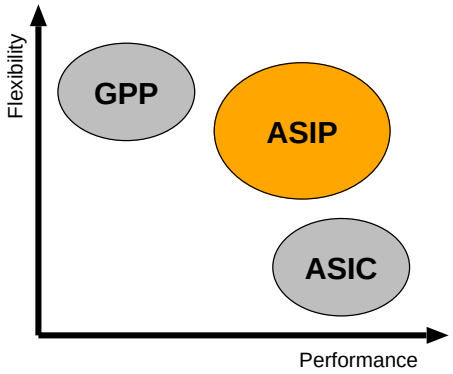


Figure 1.1: *Performance/flexibility trade-off.*

amortized if the chip can be reused for multiple purposes. This flexibility is achieved with the introduction of programmability, providing the means to adapt a design to one out of a class of applications. Furthermore, programmability alleviates some other issues:

- The complexity of SoCs makes the design process error-prone. With programmable elements bugs can be fixed without redesigning the hardware, avoiding another iteration of NRE and reducing the economic risk.
- The software part of a programmable SoC can be designed in parallel to the hardware which improves the time to market.
- The software can be adapted to changing requirements after the hardware has been produced. This increases the life-span of the design and the time in market.

Consequently, already in early 2004, 90% of new 130-nm ASIC designs included a processor core [Jer04] and there is a trend to increase the number of processors in a single SoC up to several hundreds [AKMN02].

Methods that are employed in the design of programmable SoCs are collected under the title *hardware/software co-design (HSCD)*. The main problems that these methods address are:

Hardware/software partitioning: The partitioning of applications into a first part that is implemented in software running on programmable processor cores, and a second part that is implemented in “hardware”, i.e., in hard-wired ASIC technology or in programmable logic.

Design-space exploration: Which and how many of the available building blocks to choose and how to arrange them on the chip.

Communication synthesis: How to organize communication between the parts of an application that are distributed across different components on the chip.

The larger the hardware partition of the SoC the higher is the performance but the lower the flexibility of the design. A more fine-grained and gradual trade-off between flexibility and performance, or programmability and efficiency, can be achieved with *application-specific instruction-set processors (ASIPs)*. The instruction set of an ASIP is specialized towards 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 (see Figure 1.1). Since most SoCs are specialized towards a particular application domain, the ASIP concept is a good match and ASIP cores are a valuable extension for the building-block libraries in SoC design [AKMN02].

Some vendors offer extensible processor platforms which consist of a standard processor core that can be augmented with specialized instructions, e.g., the Tensilica Xtensa [Ten] or the ARCtangent by ARC [Arc]. The platforms are supported by a tool environment, e.g., in Tensilica's case centered around the Tensilica instruction extension (TIE) language to describe the instruction-set extensions [Gon00]. The tools typically support synthesizing new instructions for the processor and adding support for new instructions to compilers by means of intrinsics, i.e., explicit calls to the special instructions in the high-level language.

1.1.2 Networking SoCs

In the networking field, protocol standards keep changing and evolving and new features are introduced at a high rate. Therefore, the time-to-market advantages and the flexibility that SoCs offer are in high demand in this domain.

Considerable research has been conducted on the architecture level of networking SoCs. An approach to quickly implement new communication protocols in a mixed hardware/ software system can be found in [SC98]. Although in principle programmable, the system is not supposed to run any other protocol after implementation. Flexibility is not a design goal here but the cost-performance trade-off is optimized.

Approaches for more flexible network processors (NPs) are described in [Ben01, SM01, VLW00]. The hardware/software partitioning of these designs is rather coarse-grained: Flexibility is achieved by integrating processor cores with general-purpose instruction-sets while performance critical parts of an application are offloaded to hardwired logic. A commercial example of such an NP architecture is the Intel IXP1200 which comprises

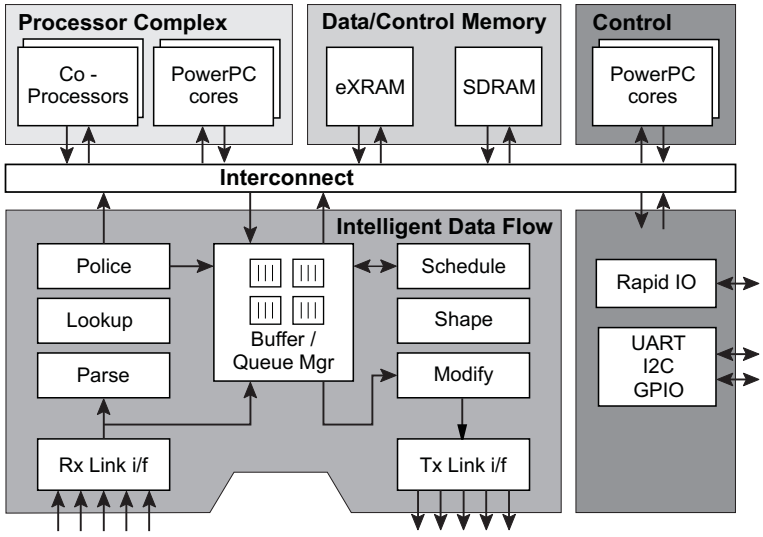


Figure 1.2: *NP with ASIP-based fast path.*

six GPP cores, called ‘micro-engines’, and hardware assists for hash functions, queuing, and bit operations. An exhaustive survey of commercially available NPs can be found in [Sha01].

In our NP architecture, depicted in Figure 1.2, we follow the fine-grained ASIP approach [GDD+03]. The core of the NP is a fast path (intelligent data flow; IDF) consisting of highly specialized processor cores for basic network processing tasks, such as header parsing, table look-up, or scheduling. These ASIPs are arranged in a pipelined fashion, passing the result of their computation on to the next processing element directly rather than through central memory. Only special packets that require complex handling are offloaded to the GPP cores in the slow path. In this way, the ASIPs provide wire-speed packet handling with programmability for future adaptations while the GPPs provide extra flexibility for tasks that are too complex for wire-speed processing. In either case, the ASIPs perform fast pre- and post-processing on which also the GPPs can rely.

Another research project that follows a similar approach is the PRO3 processor [VNO+03]. An industrial example of such an architecture are the EZchip NPs with their TOPcore ASIPs [EZ99]. The projects vary in the flexibility of the ASIPs, the on-chip interconnects, and the integration of the GPPs.

1.2 Current ASIP Design Methods

1.2.1 A Generic Design Flow

To derive an ASIP from applications in the target domain, a number of techniques is combined into a design methodology for ASIPs. Figure 1.3 shows a typical ASIP design flow. 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 translates 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.

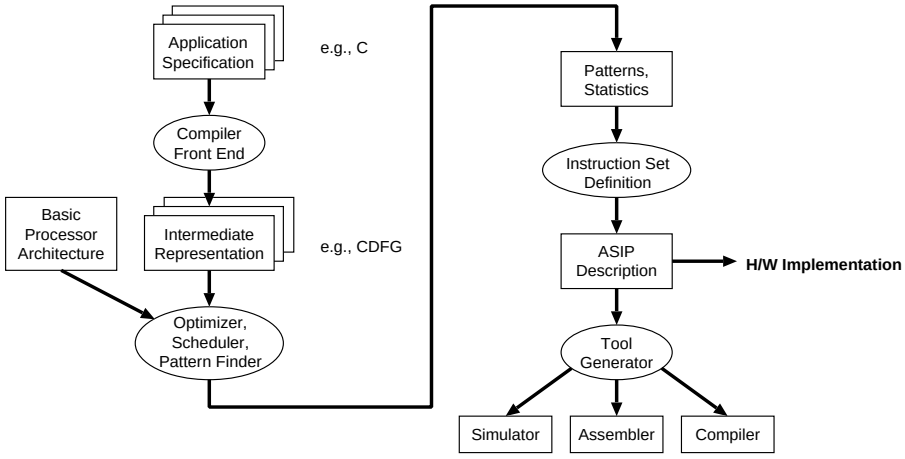


Figure 1.3: *A generic ASIP design methodology.*

Based on an architecture template, this graph is optimized, employing methods found in the compiler literature [ASU86, Muc97, Mor98], 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 processor description is implemented and retargetable tool suites are used to quickly build a development environment around the processor, including simulator, assembler, compiler, and debugger [PKB01]. A recent development is that an implementation in a hardware description language (HDL) as well as tools can even be generated automatically for the new processor from a formal processor definition in an architecture description language (ADL) [PHZM99, ENF00]. 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. The automatic generation of compilers from processor descriptions continues to be an active field of research.

In [WL01], the manual 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.

1.2.2 Instruction-Set Generation

A simple approach for ASIP instruction-set design proposed in [vPGLM94] is to analyze 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 [HD94] 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. [Gsc99], 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 [AC01], 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 enabling 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 a 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., combinations with other nodes that share at least one operand with it in the application graph. Patterns from this library are then selected to cover the application graph such that each operation is covered by exactly one pattern. This selection is called a *cover* of the application graph. A variation of dynamic programming is employed to minimize the implementation cost of the cover.

The patterns in the library are sorted by the number of times they occur in the application graphs and by the number of times they were selected for a cover. From this list, patterns are manually selected, grouped, and implemented.

A different method to cluster *parallel* operations to form new instructions is proposed in [BKKS02]. DFG nodes are scheduled as soon as possible and as late as possible to determine their mobility. From this information, a graph is derived in which two nodes are connected by an edge if they can be scheduled in the same schedule step. The edges are weighted with the number of times the nodes can be scheduled together. For instruction selection, a profiling function is employed to find the most frequently occurring operation pairs. This function must maintain a library of candidate pairs in order to collect profiling information, but it is not described in the paper.

Library based approaches suffer from the size of the libraries. In [Arn01], memory requirements of more than 200 MB and a running time of more than 24 hours are given for single benchmark applications. As a consequence, the author proposes a number of heuristics to keep the library size low by removing less promising patterns from the library. In this thesis we propose an exact solution to this problem by means of a new library structure that enables orders of magnitude faster access.

A completely different approach is introduced in [API03]. The authors propose to organize the patterns of a basic block in a virtual search-tree. This tree enables effective pruning of regions in the search space that violate design constraints. For each basic block, a search algorithm finds the pattern with the highest speed-up. Of these patterns from all basic blocks, the one with the highest speed-up is chosen as a new processor instruction.

The selection algorithm iterates, searching for an incremented number of non-overlapping patterns in the one basic block from which the new instruction was chosen. Again, the result from the basic block with the highest speed-up is selected. The iterations continue until a predefined number of new instructions is reached.

1.3 A New Methodology for Control-Dominated ASIPs

This section motivates the introduction of a special design methodology for control-dominated ASIPs. We give an overview of our approach and state our basic assumptions on the ASIP architecture.

1.3.1 Control-Dominated vs. Data-Dominated ASIPs

Most research publications on ASIPs concentrate on the design of digital signal processors (DSPs) or, more generally, on the data-dominated application domain. Data-dominated applications are characterized by long arithmetic sections between control-flow boundaries, i.e., between branches. Furthermore, they typically contain many computation-heavy loops. Processing often starts with receiving a sample of data and ends with sending out

a resulting sample [MBL+96]. 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. 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.

The properties of data-dominated applications are reflected in the ASIP design methods. The special instructions of an ASIP speed up segments of the application code. With rate constraints, any speed-up will improve the performance of the applications, irrespective of the code part in which the improvement is achieved. Code in loop bodies is executed many times and is therefore a preferred candidate for specialization efforts. Long arithmetic code sections provide an appropriate search space to find patterns that occur often in the applications to justify implementation as special instructions.

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 backbone NPs[1], 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.

For instance, in a network processor such as the one in Figure 1.2, memory bandwidth and bus contention are the major performance bottlenecks. Furthermore, accesses to such shared resources introduce unpredictability [TW04] which compromises the deterministic timing that control-dominated applications often require. One way to relieve this problem is to process packet headers on the fly as they come in from a link 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. On the other hand, different header words have completely different semantics and trigger different types of processing, e.g., a header-length field compared with a protocol number. Hence, we need to specify different timing constraints in many places in the application. Moreover, some timing constraints depend on run-time information such as the header length. In this case, the input data determines a number of cycles to wait for an event to occur, e.g., to wait for the beginning of the next header in protocols with variable header lengths.

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

Table 1.1: *Characteristics of application domains.*

[1] In the backbone of a network, the main task of an NP is header processing and routing—which is control-dominated. In contrast, NPs in access routers at the fringe of the network often perform payload processing—which is data-dominated. Throughout this thesis we only consider backbone NPs.

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 1.1 contrasts the properties of the two application domains.

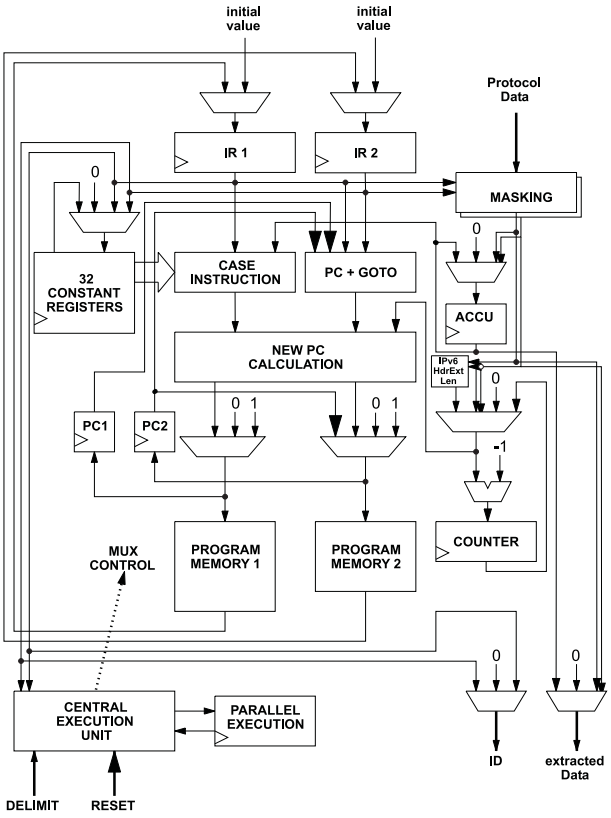


Figure 1.4: *A control-dominated ASIP.*

Some publications on control-dominated ASIPs exist, e.g. on a microcontroller [Küç99] and a Prolog processor [Gsc99], but they focus on implementation details rather than on algorithms for automatic instruction-set synthesis. Other examples of ASIPs in the control-dominated domain are the building blocks of network processors, such as the header parser shown in Figure 1.4 [Dit00], 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. This is illustrated in Figure 1.4 by the fact that only the right-hand quarter of the graph performs computations on input data while the other three quarters are occupied with the control flow.

Implementation of controller FSMs has been investigated for ASIC high-level synthesis (HLS) and HSCD, mainly for automotive applications as in [CGH+94], but not for instruction-set synthesis. The main difference between these two approaches is the fact that existing approaches optimize 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. This introduces a flexibility factor that is hard to quantify.

1.3.2 The Design Methodology and its Challenges

All methods for instruction-set generation surveyed in Section 1.2.2 focus on the data-dominated domain. Their metric for instruction selection is the speed-up achieved in the applications but none of them considers fine-grained timing constraints. In contrast, the research presented in this thesis is targeted at ASIPs for the control-dominated domain. Therefore, we consider fine-grained timing constraints throughout our design methodology which is depicted in Figure 1.5.

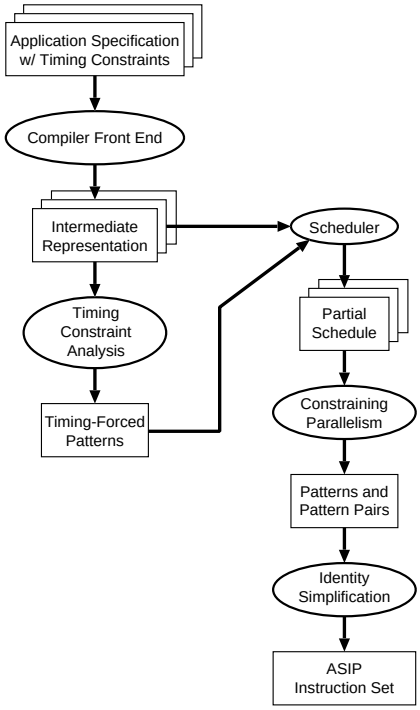


Figure 1.5: *Design methodology for control-dominated ASIPs.*

The first step in working with timing constraints is to specify them along with the applications in a high-level language (HLL). Many applications in the embedded area are programmed in C. However, the C language does not provide any constructs to specify timing. Therefore, we have to augment C with timing constructs. The constructs should be defined in a standard-compliant way to maintain compatibility with the existing tool infrastructure. Furthermore, to provide an appropriate resolution for the timing constraints they must be specified as accurately as possible. This is complicated by the fact that a single C statement often corresponds to multiple basic operations which access an even larger number of variables. The timing constructs must enable the programmer to associate constraints with one particular variable access. We define a set of pragma directives that meet all these requirements.

A compiler front-end translates this timing-annotated HLL to an IR on which compiler transformations and analyses can operate. The IR must be able to represent the timing constraints in addition to standard control and data flow. We address this requirement with a combination of appropriate IRs.

Once formally captured, we analyze the timing constraints to collect the *timing-forced patterns* that are necessary to meet the constraints. A scheduler employs these patterns

and additional basic operations in computing partial schedules of the applications. The scheduler assumes unbounded parallelism and is based on an analysis of the control, data, and timing dependencies between the operations. A partial schedule provides statistical information on how often pattern pairs occur in parallel in the same cycle.

Based on these statistics, the next method in the flow constrains the number of instructions issued in parallel. The method chooses the pattern pair that provides the highest reduction of parallel issues and bundles it to form a single instruction. With the new bundle in the instruction set, a new partial schedule is computed and the process is iterated until the maximum number of parallel issues that the designer specified is no longer exceeded.

In a last step, the resulting instruction set is simplified employing identity operands to merge related instructions for a leaner instruction set. This step is not specific to control-dominated ASIPs but is useful for any instruction set.

1.3.3 Data-Push Communication

In our research we focus on highly-specialized small ASIP cores with deterministic performance which guarantees to meet tight timing constraints. To avoid the unpredictability that shared bus and memory accesses introduce the ASIP does not fetch input data from a central memory. Instead, we rely on the communication model that we have sketched in Section 1.3.1: The 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 advantage of data-push communication is that it avoids any complexity associated with queuing, buffer management, and access to shared busses and memory. 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.

In the example case of our NP in Figure 1.2, the building blocks in the fast path (IDF) are 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 the 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. The header parser is described in more detail in Section 5.2.

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 1.6 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, in contrast, an element in the input data is accessed by waiting for it to

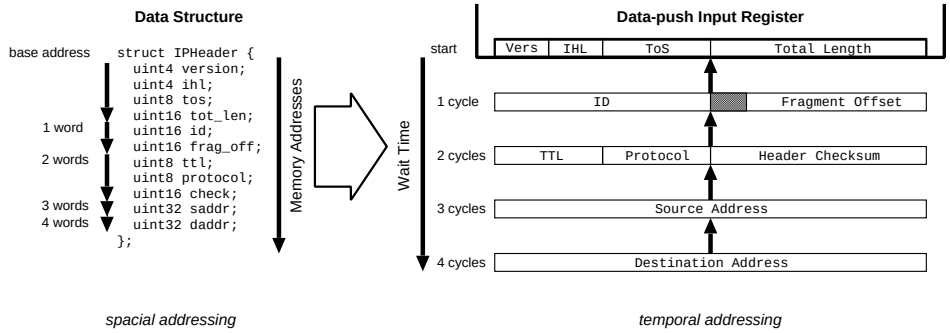


Figure 1.6: *From spacial to temporal addressing.*

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*.

1.3.4 Assumptions on Processor Architecture

The architectural decisions we have made are the following:

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 [SRM+94]. 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. VLIW architectures have been very successful in the DSP and multi-media domain.

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 [TW04]. 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.

No data memory. The reason to choose a data-push architecture was to avoid the unbounded delays of shared memory accesses in order to provide deterministic performance. Caches are a probabilistic technique and therefore do not solve the problem [TW04]. On the contrary, caches increase the worst-case access time. A viable solution is local scratch-pad memory [WM05]. To support memory, the data-dependency analysis on which some of our design methods rely requires pointer analysis to find

dependencies through accesses to the same memory location. This process is also called memory disambiguation. To avoid the complexity of this analysis we preliminarily exclude data memory elements from our considerations. Instead, we assume

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 significantly reduces the complexity of scheduling.

No interrupts. Thanks to the data-push communication and the absence of caches the ASIP does not need interrupts for I/O operations or the handling of memory page faults. We therefore do not consider interrupts which removes yet another hazard to predictable program behavior and deterministic timing.

Single-cycle instructions. Control-dominated applications typically do not use floating-point data-types but rely exclusively on integers. Therefore, 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, memory disambiguation algorithms, and register scheduling approaches to support pipelined architectures with scratch-pad memory and a limited number of registers.

1.4 This Thesis

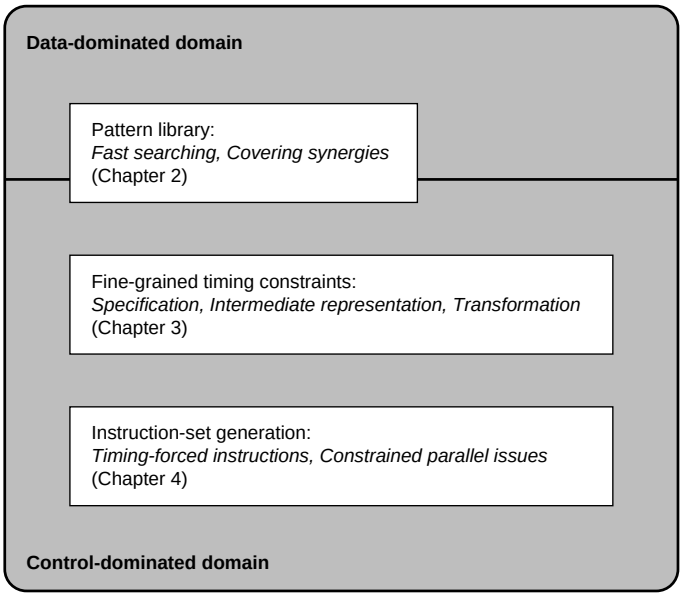


Figure 1.7: *Overview of the contributions.*

The contribution of this thesis, as shown in Figure 1.7 comprises the following concepts:

- Chapter 2 introduces a hierarchical organization of pattern libraries. Our method significantly improves the access times to the library which solves an important problem in existing ASIP design methods. Moreover, the hierarchical arrangement of

patterns reveals synergies between patterns that we exploit for leaner instruction sets and data-path sharing between instructions.

- In Chapter 3 we present our application representation that is particularly suited for the control-dominated domain. We introduce fine-grained timing constraints in ANSI C. These constraints are reflected in our intermediate representation. To demonstrate the benefit of the proposed representation we introduce an optimization technique we call *branch postponing* that resolves scheduling conflicts between deadlines.
- In Chapter 4 we demonstrate how to construct the operation patterns necessary to meet the given timing constraints. Furthermore, we introduce a method to merge parallel patterns in order to handle a constrained number of parallel instruction issues required for the ASIP. A special section is devoted to the handling of the data-dependent wait operations which are part of our application representation.

In Chapter 5 we give performance results of our organization method for pattern libraries. We also demonstrate the application of our ASIP design methodology in a case study with a real-life ASIP. Chapter 6 concludes the thesis.

2 Pattern Library for Fast Searches and Synergies

A crucial step in ASIP design is the instruction-set generation. Methods for automating this process, surveyed in Chapter 1, extract patterns from applications, usually in the form of data-flow graphs (DFGs), and insert them into a pattern library. Along with each pattern, statistical data is collected, such as the number of occurrences of a pattern in the applications. Based on this data, a subset of the patterns in the library is then selected for implementation as specialized instructions.

For each pattern which is found in the applications, a search in the library is performed to check whether the pattern is already present, and the pattern is then either added to the library or the statistics are updated. The complexity of this search has a significant impact on the running time of instruction-set synthesis tools. Current algorithms have a computational complexity of $O(n \cdot p)$, with n the total number of operation nodes of all patterns in the library and p the size of the pattern sought.

In this chapter, we introduce a novel organization for pattern libraries that enables a search algorithm with only $O(d)$, where d is the size of the pattern sought, up to the maximum pattern size in the library ($d \leq p$). Furthermore, the library organization reveals opportunities to substitute one pattern by another. This can be exploited for more efficient instruction selection and code generation.

The chapter is structured as follows: In Section 2.1 we refer to related work on the generation of pattern libraries for instruction-set generation, on search algorithms, and on methods based on identity operands. In Section 2.2 we introduce our concept of a novel graph-based library organization that facilitates fast pattern searches. We present search and insertion algorithms working on the graph. In Section 2.3 we extend this concept and show how identity operands can be used to substitute a group of patterns by a super-pattern. We summarize the chapter in Section 2.5.

2.1 Related Work

2.1.1 Pattern Libraries

While most approaches to instruction-set design for ASIPs involve the construction and evaluation of a pattern library, the organization of these libraries has not been described in the literature [vPGLM94, HD94, BKKS02].

The library-construction algorithm described in [AC01] tries to find pattern matches by iterating over the operation nodes of the patterns in the library and comparing them with the nodes in a subject pattern. We conclude that the library is an unordered collection of patterns. In the worst case, a search algorithm on such a library has to compare all operation nodes of the pattern in question with all operation nodes of all patterns in the library. Hence, the computational complexity of this search is $O(n \cdot p)$, with n the total

number of operation nodes of all patterns in the library and p the size of the pattern sought. In order to keep n low, the author introduces heuristics to limit the library size by excluding patterns that do not seem beneficial.

Because a search is conducted for each pattern found in the applications and because the pattern libraries tend to be large, the computational complexity of the search algorithm has a significant impact on the total running time of the instruction-set generation process. In [Arn01] for instance, memory requirements of more than 200 MB and a running time of more than 24 hours are given for single benchmark applications—with a number of heuristics already built in to keep library size low.

For technology mapping in logic synthesis, large cell libraries provide possible implementations of register-transfer level designs. The complexity of these libraries, however, is not due to the number of different pattern structures but due to the number of physical parameters, such as power levels, i.e., the large number of alternatives to implement the same pattern [BHSV90].

2.1.2 Searching and Pattern Matching

Search algorithms have been the subject of research for a long time. Most efficient algorithms that have been proposed apply, however, to one-dimensional data structures—in particular to string searches [CLR90]. DFGs, in contrast, are two-dimensional structures. While for strings it is clear that the next character will follow at the end of the string, in DFGs the next operation node to be added can be operand to one of many nodes. [HO82] proposes a method to transform trees into strings and then relies on string matching algorithms. In contrast, our approach does not require a transformation of the DFG but actually exploits its tree structure.

The covering of code sections with appropriate elements in a pattern library for implementation has traditionally been performed by graph-based pattern matching algorithms, e.g., for code generation in compilers [HO82, AG85] and for technology mapping in logic synthesis [Keu87, ZS01]. These algorithms assume a fixed set of patterns and they perform preprocessing on this set to speed up the actual matching. Methods for instruction-set generation, in contrast, construct their libraries on the fly, iteratively filling the library with new patterns. For such dynamic libraries the preprocessing of the entire pattern set would have to be repeated for each new pattern. This problem renders the algorithms very inefficient in our context.

More recently, symbolic algebra [PSM02] and combinational equivalence checking [CPHC04] have been employed for this mapping. Symbolic algebra, however, is constrained to patterns that represent linear functions. Non-linear functions can only be approximated. Both symbolic algebra and combinational equivalence checking are very complex and thus only feasible for small problems. They may be beneficial for canonicalization of patterns in combination with fast matching methods such as those presented in this chapter.

2.1.3 Identity Operands and Datapath Sharing

Identity operations, which we use to reduce patterns to simpler ones, have been exploited for basic algebraic transformations in compilers [Muc97] and in high-level synthesis [LM97]. The idea to use identity operands for pattern simplification is mentioned

in [CKG+96] but the authors provide no algorithms. In [PD94] and [HE99], identity operations are *inserted* into sequences of operation nodes in order to increase the number of identical patterns. In contrast, we propose to use identity elements to *eliminate* nodes from patterns. The library we construct in this way reveals the same opportunities to substitute one pattern by another—and more, because our approach is not constrained to small sequential patterns.

Another approach to merge patterns for datapath sharing is presented in [MAHM02]. The authors present heuristics to map edges in a pattern to edges that connect similar nodes in other patterns. The resulting combined pattern can then be configured to implement any of the original patterns by disconnecting different sets of edges.

The heuristic presented in [BKS04] decomposes the DFGs of complex instructions into a set of leaf-to-root paths. In this set the algorithm finds the common operation subsequence with the maximum area and merges instructions at this subsequence. The process is iterated until no more merging or no further area improvement can be achieved.

The latter two approaches rely on reconfigurable interconnection networks and multiplexers to configure the merged datapath. This approach entails more complexity than disabling operators by identity operands.

2.2 Hierarchical Library Organization

2.2.1 The Pattern Search Graph

Arranging patterns in a linked list results in a completely arbitrary order. There is no relation in the order of the patterns that could be exploited for directed and therefore faster searches. But patterns do have a structure that lends itself to ordering. In particular, a pattern can be a subgraph of another pattern, or two patterns can have common subgraphs. In the following, we present a method to order patterns according to their structure and the operation type of their nodes. First, we introduce the method for tree-shaped patterns and then generalize it to patterns that are directed acyclic graphs (DAGs) in Section 2.2.4. The resulting new structure supports significantly faster searches than conventional library structures.

We arrange the patterns in a tree. Each node in this tree represents a pattern. The root of the tree has edges to all patterns that consist of a single basic operation, forming a first level of patterns. The basic operations can be extended to form two-node patterns by connecting the output of another operation to one of the basic operation's operand inputs. Each of the first-level patterns has edges to all two-node patterns with the respective first-level pattern as their root node. The two-node patterns are the second level of the library tree. This process continues for more complex patterns, adding more levels to the library graph. We call this tree graph a *pattern search graph (PSG)* as it facilitates directed and fast searches as we will show in Section 2.2.2. Figure 2.1 shows an example PSG of a pattern from an application that parses headers of network packets [Dit00].

Compared with the linked list, the structure of a PSG is two-dimensional rather than one-dimensional. This entails the following redundancy. Take a pattern consisting of three operation nodes, two of which feed their result as operands to the root node. In this case, we could have two PSG paths to the pattern: first adding the left operand to the root operation and then right, or adding the right operand first and then the left. In order to have only one path from the library root to each pattern, we posit that appropriate operations be connected

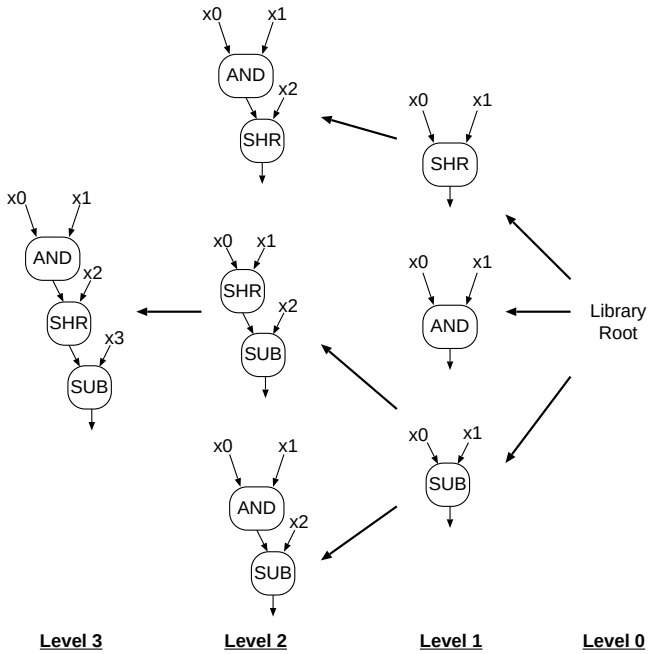


Figure 2.1: *Example of a pattern search graph.*

first to right operands and then to left operands, assuming basic operation nodes with two operands. The search algorithm presented in the next section requires only the remaining PSG paths with precedence for the right operand.

The PSG organization has the side-effect that all patterns on the path to a complex pattern must be present in the library. Hence, when inserting a pattern into the library, we sometimes must also insert additional patterns that are on the PSG path to the new pattern but that are not present in the library, yet. Our performance measurements in Chapter 5 show, however, that the overhead presented by this effect is marginal.

2.2.2 Searching a Pattern in a PSG

The access to the pattern library can be accelerated significantly by exploiting the order of the patterns in a PSG. When searching for a particular pattern in the library, we start with one of the primitive operation nodes it comprises, namely, the root node. We then add operation nodes in the pattern by following the edges in the search graph. In this way, we arrive at the complete pattern, provided it exists in the library.

The recursive algorithm in Figure 2.2 implements the proposed search strategy. It traverses the pattern sought depth-first and right-branch-first, corresponding to the rule for avoiding redundant paths we posited earlier. The procedure returns either the position of the pattern in the library or NULL.

Consider the example PSG in Figure 2.3. In order to search this graph for, e.g., the pattern in the upper right corner, the algorithm starts with the pattern root—in this case the subtraction. In the first line of the *find* function, `libNode` is set to the library entry corresponding to the pattern root by following the `next`-pointer indexed by the only operand of the library root and by a *SUB* operator. Then the right operand of the pattern root—labeled

```
activeOpnd = 0;     /* global variable */
patternInLibrary = find( patRoot, libRoot );


LibNode find( patNode, libNode ) {
 LibNode nextLibNode = libNode.next[activeOpnd][patNode.operator];

 activeOpnd = 0;    /* right operand is always 0 */
 if (patNode has rightOpnd AND nextLibNode exists)
  nextLibNode = find( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd AND nextLibNode exists)
  nextLibNode = find( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 2.2: *Pseudo code: Pattern search in a PSG.*

x2—is examined, which is NULL because it is an external pattern input. Therefore, it is skipped and the left operand is checked, which is not NULL because it is connected to the output of the shift operator. Consequently, the *`find`* function is called recursively with the shift operation as the next `patNode` and the subtraction as the `libNode`.

This time, the first line of the *`find`* function follows the `next`-pointer indexed by the second operand of the subtraction and by a *SHR* operator and hereby sets `libNode` to the library entry we have been seeking. Both, the left and right operand of `patNode` are NULL and therefore the library entry sought is returned, after unwinding the recursive calls, to be assigned to `patternInLibrary`.

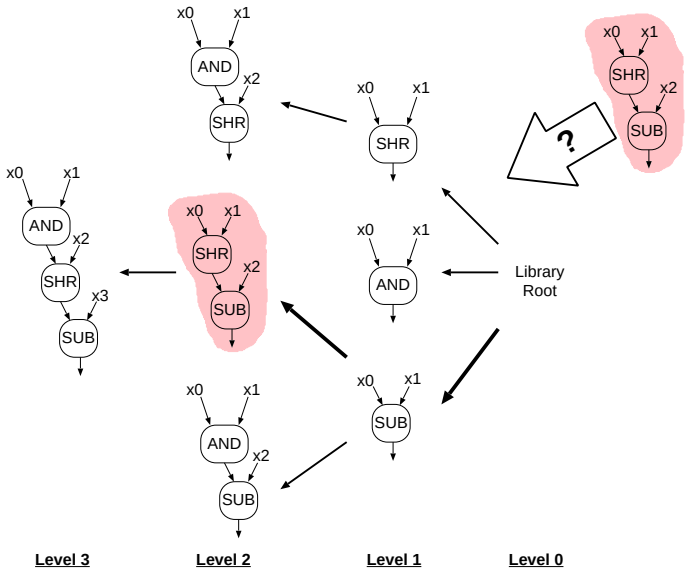


Figure 2.3: *Tree-pattern search in a PSG.*

Operand Numbering

The two-dimensional array `LibNode.next` holds references to all library entries that are derived from the current entry by attaching one more operation node to the associated pattern. The array is indexed by the operator of the attached node and by the operand of the current pattern to which the new node is attached.

To identify the operands they must be numbered. But not all operands are candidates for attaching another operation node. Figure 2.4 illustrates why. The search algorithm first traverses the right-most branch of the searched pattern until it has reached the end of the branch in step 3. Only then it considers the left operands, recursively backtracking the right-most branch. When the next operator node is attached in step 4, then all operands that have been visited before will not be used anymore in the future because if there would be an operation node to attach to any of them then this would have happened at the first visit to the operand. Hence, these operands are "dead" and need no entry in the `next`-array, nor do they need a number. The remaining numbered operands we call *live operands*.

The left-most right operand always is number 0. Left of operand number 0, the left operands are numbered along the fringe of the pattern. That is why the pattern in step 4 of Figure 2.4 has one operand column less in its `next`-array than the pattern in step 3 although it has one external operand more.

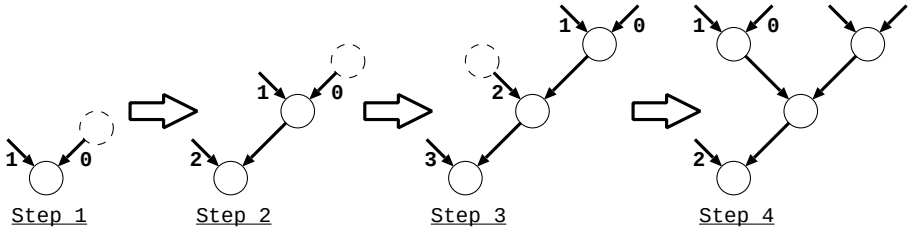


Figure 2.4: *Operand numbering.*

In the algorithm in Figure 2.2, `activeOpnd` is set to 0 when a right branch is explored because a right operand always has number 0. If there is no operator node attached at the right branch then the algorithm continues with the left operand of the same node which then has operand number $1 = 0 + 1$. If there is an operator attached and the recursion eventually returns from a right branch then the algorithm will have visited some left operands in the right branch and it will have set `activeOpnd` accordingly. The left operand of the current node is the next in line and its operand number is the value of `activeOpnd` plus one. Therefore, `activeOpnd` is increased by one before exploring the left branch of the current node.

This approach guarantees correct numbering of the operands, according to the numbers in Figure 2.4, and results in the minimum possible size of the `next`-array with each library node.

Computational Complexity

The *`find`* function is called at most once for each node in the pattern sought. The pointers to the next library nodes are stored in the array *`libNode.next`* with linear access time. Therefore, this search is $O(p)$, with p the number of operation nodes in the pattern sought. If the pattern sought is larger than the largest pattern in the library then the search stops

```
activeOpnd = 0;     /* global variable */
patternInLibrary = insert( patRoot, libRoot );


LibNode insert( patNode, libNode ) {
 LibNode nextLibNode = libNode.next[activeOpnd][patNode.operator];
 if (nextLibNode does not exist) {
  liveOpnds = libNode.liveOpnds - activeOpnd + 1;
  nextLibNode = new LibNode( liveOpnds );
  libNode.next[activeOpnd][patNode.operator] = nextLibNode;
 }

 activeOpnd = 0;    /* right operand is always 0 */
 if (patNode has rightOpnd)
  nextLibNode = insert( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd)
  nextLibNode = insert( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 2.5: *Pseudo code: Inserting a tree pattern into a PSG.*

even earlier. Hence, the worst-case computational complexity of a search is $O(d)$, with d the size of the pattern sought, up to the maximum number of operation nodes in any pattern in the library—which is equal to the maximum depth of the library search-graph. Note that $d \leq p$. Our experimental results presented in Chapter 5 show that indeed a search in a PSG is orders of magnitude faster than a search in a linked list.

2.2.3 Inserting Patterns into a PSG

We insert a pattern into a PSG by searching it and complementing the path if it ends before finding the pattern. The algorithm is given in Figure 2.5.

The difference to the search algorithm in Figure 2.2 is in creating a new `LibNode` if `nextLibNode` does not yet exist. The parameter passed to the new `LibNode` is the number of live operands that the associated pattern has. It is needed to dimension the `next`-array correctly. Compared with `libNode`, the number of live operands for the new `LibNode` is reduced by the number of the current active operand where the next pattern node is attached. This is because the number of an operand corresponds to the number of live operands to the right of it, and all operands that are right of the operand where a new node is attached are dead, as explained in Section 2.2.2. The resulting number is then increased by one because the new node occupies the current active operand but contributes two new operands—left and right. Hence, the number of live operands for the new `LibNode` computes to

$$libNode.liveOpnds - activeOpnd + 1. \tag{2.1}$$

The modification of the insert algorithm compared with the search algorithm does not affect the computational complexity. It is also $O(d)$, with d the size of the pattern sought, up to the maximum number of operation nodes in any pattern in the library, and $d \leq p$.

```
activeOpnd = 0;     /* global variable */
patternInLibrary = find( patRoot, libRoot );


LibNode find( patNode, libNode ) {
 LibNode nextLibNode;
 if ( patNode.nodeNumber == libNode.nextNodeNumber )
  nextLibNode = libNode.next[activeOpnd][patNode.operator];
 else
  nextLibNode = libNode.revisit[activeOpnd][patNode.nodeNumber];

 activeOpnd = 0;
 if (patNode has rightOpnd AND nextLibNode exists)
  nextLibNode = find( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd AND nextLibNode exists)
  nextLibNode = find( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 2.6: *Pseudo code: DAG pattern search in a PSG.*

2.2.4 Extension for DAG Patterns

In order to extend the PSG concept from trees to directed acyclic graphs, we need to cater for operator nodes whose result feeds more than one operand of other operator nodes. We support this by introducing a `revisit`-array with each PSG entry, similar to the `next`-array, through which operator nodes are revisited that have been encountered before via a different operand. The `revisit`-array is indexed by the operand number to which the next node is attached, and by the unique number of the operator node that is being revisited via this operand. Therefore, the operator nodes must be numbered which we do in the same order in which the search and insert algorithms traverse them. In this way we can detect that we revisit a pattern node because it will have a lower number than the current node while otherwise, the number of the next node is one higher.

The *search* algorithm in Figure 2.6 has been extended by the comparison of the number of the searched pattern node and the number of the next new node to be added to the library entry. If they are not equal the pattern node has been visited before. In this case, the PSG is traversed through a pointer in a `revisit`-array. For newly discovered nodes of the searched pattern, the `next`-array is employed for traversal as before.

Analogously, the *insert* algorithm for DAG patterns in Figure 2.7 first tests whether the next pattern node is being revisited or not. If it is being revisited the algorithm tries to find the according library entry in `revisit`-array, otherwise in the `next`-array. If the library entry does not exist it is created and entered in the appropriate array.

The `LibNode` constructor now has the next node number as an additional argument. This is the number of the next new node to be added to the pattern associated with the library entry. If the current pattern node has been visited before the next node number is the same as with the current library entry because there is no new node being attached—it is merely being revisited. In contrast, if the pattern node is new the next node number is one higher than of the current library entry because the current number will be taken by the new node.

We formulate the computational complexity of the DAG algorithms dependent on the number of edges in the pattern rather than the number of operator nodes because it is the

```
activeOpnd = 0;     /* global variable */
patternInLibrary = insert( patRoot, libRoot );


LibNode insert( patNode, libNode ) {
 bool revisit = ( patNode.nodeNumber == libNode.nextNodeNumber );
 LibNode nextLibNode;
 if (!revisit)
  nextLibNode = libNode.next[activeOpnd][patNode.operator];
 else
  nextLibNode = libNode.revisit[activeOpnd][patNode.nodeNumber];

 if (nextLibNode does not exist) {
  liveOpnds = libNode.liveOpnds - activeOpnd + 1;
  if (!revisit) {
   nextLibNode = new LibNode( liveOpnds, libNode.nextNodeNumber + 1 );
   libNode.next[activeOpnd][patNode.operator] = nextLibNode;
  } else {
   nextLibNode = new LibNode( liveOpnds, libNode.nextNodeNumber );
   libNode.revisit[activeOpnd][patNode.nodeNumber] = nextLibNode;
  }
 }

 activeOpnd = 0;
 if (patNode has rightOpnd)
  nextLibNode = insert( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd)
  nextLibNode = insert( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 2.7: *Pseudo code: Inserting a DAG pattern into a PSG.*

edges that the algorithms traverse and in a DAG there can be more than one edge per node. This, however, still results in a linear complexity: $O(e)$, with e the number of edges in the pattern sought, up to the maximum number of edges in any pattern in the library—which is again equal to the maximum depth of the library search-graph—plus one because the first step from the library root to the first library entry does not correspond to an edge in the pattern. This complexity is equal to the tree-pattern case if we also formulate it dependent on the number of edges in the searched pattern.

2.3 Exploiting Similarities between Patterns

The pattern search graph exploits relations between patterns for speeding up searches in a library. There are similar relations based on which a pattern can be used to substitute a class of simpler patterns by disabling different parts of the pattern. This can be exploited in ASIP design to speed up different application patterns with only one special instruction, in this way resulting in a leaner instruction set. The method also supports finding more opportunities to employ complex instructions during code generation. Furthermore, it provides a systematic approach to finding opportunities for datapath sharing in high-level synthesis. In our method we use identity operands to disable operation nodes in a pattern in order to mimic any of a group of simpler patterns.

2.3.1 Identity Operands

Most primitive operations that are found in the instruction sets of general-purpose processors can be used to map one input operand a to itself by applying an identity operand op_{id}, i.e. the algebraic identity element for that operator, to the other input such that

$$a \circ op_{id} = a, \quad or$$
$$op_{id} \circ a = a,$$

turning the primitive operation $\circ$ into an identity operation. Examples of identity operands are given in Table 2.1.

primitive operation	left operand	right operand
$+$	0	0
$-$	n/a	0
$\times$	1	1
$/$	n/a	1
$<<, >>$	n/a	0
AND	all 1's	all 1's
OR, XOR	0	0

Table 2.1: *Identity operands.*

An operand for an operation node in a DFG pattern is either generated by another node in the same pattern or is an external input to the pattern. Depending on their operands, we distinguish three types of nodes:

- A *leaf node* has two operands that are external inputs to the pattern.
- An *internal node* has two operands that are both generated by other nodes in the same pattern.
- A *cyclops node* has only one operand that is an external input to the pattern and the other operand is generated within the pattern. Depending on whether the external input is the right or left operand, we call the node a *right cyclops* or a *left cyclops*, respectively.

A complex pattern can be transformed into a simpler pattern by applying the identity operands of its operation nodes to the appropriate inputs, thereby effectively eliminating nodes from the pattern. Particular operands can be applied directly to leaf nodes and to cyclops nodes. The non-commutative operations in Table 2.1 have no left identity operand. Nodes of these operation types must be leafs or right-cyclops nodes to be removable, i.e., their right input must be accessible from outside the pattern.[1]

2.3.2 The Identity Operand Graph

By applying identity operands to one node at a time, a pattern of n nodes, of which m are removable can be transformed into m patterns of $n-1$ nodes. By recursively repeating

[1] Load and store operations have no identity operand at all, but because of their long latency they are less relevant for instruction-set generation than arithmetic operations.

this on each of the simpler patterns, the complex pattern can eventually be reduced to primitive operations. If all leaf nodes and all cyclops nodes at any stage of the recursion are removable then the set of primitive operations includes all operation types that occur in the pattern. The primitive operations finally all converge to a *move* operation.

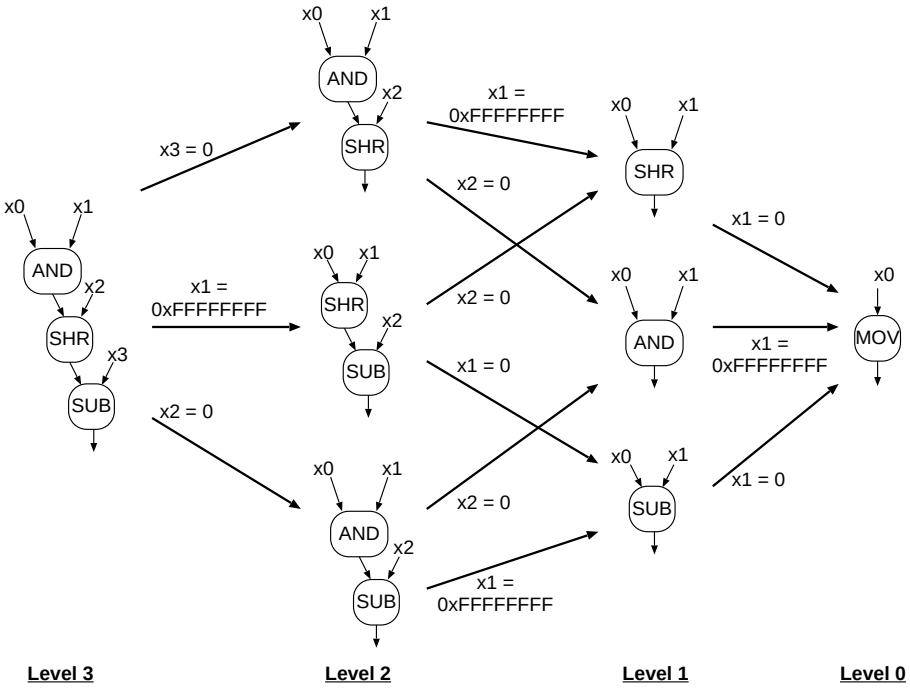


Figure 2.8: *IOG of a pattern.*

The sequence of applying the identity operands can be used to sort the patterns. We represent this sorting as a graph with the graph nodes being the patterns and the directed graph edges representing the application of an identity operand to one particular operation node in the pattern. The edges are directed from the more complex pattern to the derived smaller one. We call this type of graph an *identity-operand graph (IOG)* and say that a complex pattern *dominates* the simpler patterns in its IOG. Figure 2.8 shows the IOG for the level-3 pattern in the PSG in Figure 2.1. Evidently, both graphs contain the same patterns and only differ in the number and the orientation of the edges. This suggests that both graphs can be efficiently constructed simultaneously and be held in a unified data structure.

The library IOG shows which simpler patterns can be covered by a complex instruction during code generation, again by applying the appropriate identity operands to its input. Therefore, these simpler patterns need not be implemented as individual instructions if the complex pattern is chosen for implementation—provided that the possibly slower execution and the cost of applying the identity operands can be afforded. This cost may be, for instance, additional *move* instructions. If the cost is lower than the benefit then the IOG reveals opportunities to substitute patterns by more complex ones during instruction-set synthesis and code generation, leading to a reduced number of special instructions that provide the same benefit.

The power of a complex pattern to cover all derived simpler patterns seems to suggest that only the most complex patterns should be chosen for implementation. But in ASIP design methodologies there is an implementation cost function $C(pattern)$ associated with

the patterns that usually increases with pattern complexity, capturing for instance operand encoding effort, die area, or latency. This cost function balances the derived tendency towards more complex patterns for implementation.

In practice, nodes could also simply be deleted from the pattern for construction of the library graph, whether there is an appropriate ID operand or not. This approach would, however, eliminate the IOG property that a pattern can substitute all other patterns in its IOG by applying ID operands accordingly, a property that is needed for more efficient instruction selection.

2.3.3 Inserting Patterns into an IOG

In an early article on IOGs [Dit03] we outlined an algorithm for inserting the IOG of a pattern into a PSG library. First, we constructed the IOG of the new pattern and derived the corresponding search graph. Then, for each path in this search graph, we tried to find the corresponding path in the library. If a path did not exist in its entirety in the library then the part existing was connected to the remainder of the path in the search graph of the new pattern. Those parts of the new search graph that already existed in the library were deleted.

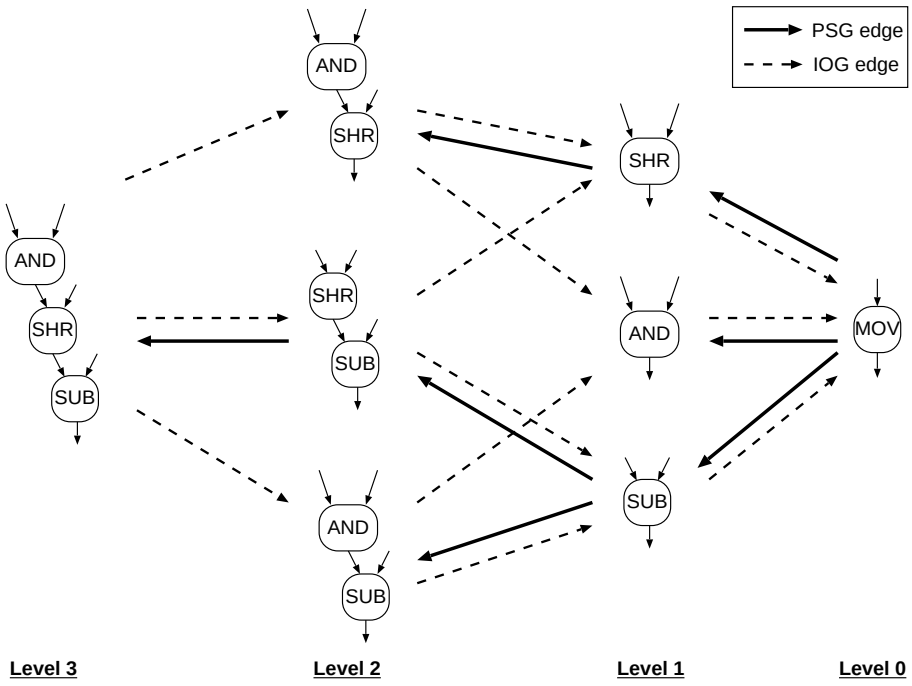


Figure 2.9: *Combined PSG/IOG.*

In the following we give a detailed description of a new algorithm that combines the construction of the IOG of a pattern with its insertion into a PSG library. The resulting combined pattern-search/identity-operand graph enables fast searches and shows synergies between patterns at the same time. Figure 2.9 shows the combined graph for the patterns in Figures 2.1 and 2.8. The search algorithm requires only the PSG edges and therefore the code in Figure 2.2 does not need to be modified. In contrast, the insertion algorithm has to create also the IOG edges.

The algorithm traverses the pattern to be inserted in the same way as the algorithms in Section 2.2.1, discovering right branches first. While doing so the algorithm maintains a list of patterns it has inserted, called *fragments*, that are consecutively combined with discovered pattern nodes to form all possible combinations of pattern nodes that occur in the IOG of the pattern. Each of these combinations is inserted into the PSG library and the IOG edges are created, provided that the according node is removable.

Discovered pattern nodes must be combined with fragments in such a way that only patterns are generated that can be modeled by disabling operation nodes from the original pattern to be inserted. In particular, a first node can only be operand to a second node on the same side—left or right—on which the pattern branch that contains the first node is attached to the second node.

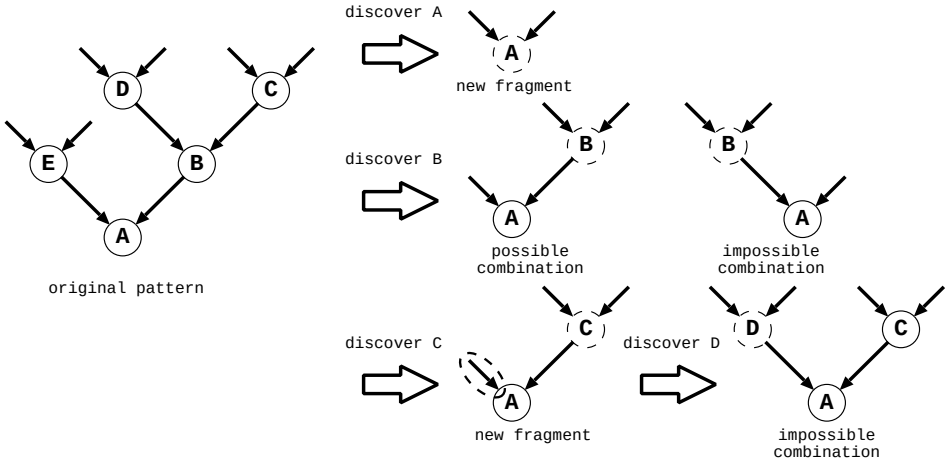


Figure 2.10: *Fragment combination.*

Consider, for instance, the pattern in Figure 2.10. The nodes are labeled in the order in which they are discovered. Node A is discovered first and inserted into the library as well as into the list of fragments. Node B is discovered next. Now, the only possible combination with node A in the fragments list is to make B the right operand of A. There is no set of nodes in the original pattern to be disabled such that node B would be the left operand of A.

In order to ensure that discovered nodes are combined with fragments only in possible ways, each fragment has one operand marked as *active operand*. When combining, a discovered pattern node is always attached to the current active operand of the fragment. Once the pattern branch attached to the active operand of a fragment has been completely discovered, the *active* mark is moved to the next possible operand of the fragment before nodes in the branch attached to this new active operand are being discovered. If there is no possible operand left to become active then all combinations with the fragment have been generated and the fragment is deleted from the list of fragments.

Attaching a discovered node to the active operand of a fragment, however, can still result in impossible combinations. This is the case if the pattern branch that is currently discovered is a left branch of a pattern node that is not contained in the fragment but the fragment does contain a node from the right branch. The node from the right branch then blocks the only operand where a node from the left branch could be attached to create a possible combination.

To illustrate this, consider again the pattern in Figure 2.10. When node C is discovered it is combined with node A from the fragment list as its right operand. As in the original pattern none of C's operands is fed by an operation node, the next possible operand of the new fragment is A's left operand which is therefore marked as the active operand, indicated by the dotted oval. If we now discover pattern node D and attach it to the active operand of the fragment we just created, as shown in the figure, then we arrive at an impossible combination again.

In order to prevent this type of impossible combination we temporarily disable fragments if the equivalent to their active operand in the original pattern is not on the path to the pattern node to be discovered next. In the example, A's left operand is not on the path to node D. Therefore, a fragment that has A's left operand marked active has to be disabled at this point. A disabled fragment is enabled as soon as the pattern node attached to its active operand is being discovered. The example fragment is enabled when node E is discovered.

In the following we present a pseudo-code implementation of the algorithm with subroutines for the fragment combination and the computation of the next active operand.

The Top-level Routine

Figure 2.11 shows the pseudo code of the top level routine of our insertion algorithm, starting with the root node of the pattern to be inserted. The discovered pattern node is combined with the fragments in `fragList` by calling `combineFrags()`. For each operand that exists, i.e., that is fed by another pattern node, `insert()` is called recursively, in this way traversing the pattern.

After the right branch has been traversed, the active mark is updated for all fragments that have the right operand of the current node as their active operand. This set of fragments coincides with the set that has been generated by the call to `combineFrags()`. All fragments generated later in the subsequent recursive call to `insert()` either do not include the current node, or in combining the fragments the right operand in question has been occupied.

For the fragment set, the active mark is moved from the right to the left operand—provided that the left operand exists. At the same time, all other fragments are enabled whose active operand is attached to the current node. This can be the case for any pattern generated while traversing the right branch of the current node. Then the nodes in the left branch are discovered.

After both branches have been traversed, the active mark for the fragments whose active operand is attached to the current node is advanced to the next active operand. Each of these fragments is disabled until the recursion unwinds back to the node to which their active operand is attached. If there was no valid next operand then all combinations with the fragment have been generated and it is deleted from the list. Consequently, the fragment list will be empty at the end of the insertion procedure.

Combining Fragments with a New Node

The algorithm `combineFrags()` in Figure 2.12 combines each enabled fragment in `fragList` with the operation node that is passed to it as an argument. If the resulting combination is not present in the library a new library entry is created and connected to a PSG edge. The number of live operands for the new entry follows Equation (2.1). The IOG edges from the new entry are held in the `eliminate`-array, indexed by the number of the

```
FragmentsList fragList = empty; // globally accessed
insert( patRoot );


insert( patNode ) {
 lastOldFrag = fragList.end();
 combineFrags( patNode );
 firstNewFrag = lastOldFrag.next;

 if (patNode.rightOpnd exists) {
  lastNewFrag = fragList.lastElement
  insert( patNode.rightOpnd );

  if (patNode.leftOpnd exists) {
   for each frag in fragList from firstNewFrag to lastNewFrag {
    frag.activeOpnd = FIRST_LEFT_OPND;
    frag.nextActiveOpnd = newNextActiveOpnd( frag );
   }
   for each frag from lastNewFrag.next to fragList.lastElement
    if (frag.activeOpnd is opnd to patNode)
     enable fragment;

   insert( patNode.leftOpnd );
  }

 } else if (patNode.leftOpnd exists) {
  insert( patNode.leftOpnd );
 } // else patNode has no opnds

 if (any of patNode's opnds exists) {
  for each frag in fragList from firstNewFrag to fragList.end() {
   if (frag.activeOpnd is opnd to patNode) {
    frag.activeOpnd = frag.nextActiveOpnd;
    frag.nextActiveOpnd = newNextActiveOpnd( frag );
    disable frag;
   }

   if (frag.activeOpnd is invalid)
    fragList.delete( frag );
  }
 }
}
```

Figure 2.11: *Pseudo code: Insert a pattern into an IOG.*

pattern node that is eliminated along the edge. For each node that can be eliminated from the combination using ID operands, the resulting pattern is searched in the library and an IOG edge to the according library entry is created.

In order to create a new entry for the fragments list, the active operand of the new fragment is determined. Depending on which operands of `patNode` exist, i.e. which of them are fed by other pattern nodes, the active operand is set to the right operand, the left operand, or to the result of Equation (2.1). The next active operand is computed accordingly. With these values, a new fragment is created and appended to the fragment list. According to the operand-numbering scheme described in Section 2.2.2 the active operand of the new fragment is not attached to `patNode` if `activeOpnd` is greater than 1. In this case, the creator routine disables the new fragment until the operation node to which the active operand is attached.

If none of `patNode`'s operands exist and the fragment with which it is combined has no valid `nextActiveOpnd` no new fragment is created as it would have no valid active operand to which an operation node could be attached for combination.

```
combineFrags( patNode ) {
 for each enabled frag in current fragList {
  libNode = frag.libraryLocation;
  nextLibNode = libNode.next[frag.activeOpnd][patNode.operator];

  if (nextLibNode does not exist) {
   liveOpnds = libNode.liveOpnds - frag.activeOpnd + 1;
   nextLibNode = new LibNode( liveOpnds );
   libNode.next[frag.activeOpnd][patNode.operator] = nextLibNode;

   for each eliminatable node in the nextLibNode pattern {
    iogChild = pattern w/o node;
    activeOpnd = 0; // global variable for find()
    edgeTarget = find( iogChild );
    nextLibNode.eliminate[patNode.nodeNumber] = edgeTarget;
   }
  }

  // create new fragment
  if (patNode's right or left opnd exists OR
      frag has valid nextActiveOpnd) {

   if (neither patNode's right nor left operand exists) {
    newActiveOpnd = frag.nextActiveOpnd - frag.activeOpnd + 1;
    newNextActiveOpnd = newNextActiveOpnd( frag );
    if ( newNextActiveOpnd is valid )
     newNextActiveOpnd = newNextActiveOpnd - frag.activeOpnd + 1;

   } else if (patNode's right and left opnd exist) {
    newActiveOpnd = RIGHT_OPND;
    newNextActiveOpnd = FIRST_LEFT_OPND;

   } else { // either left or right opnd exists
    if (patNodes's right opnd exists)
     newActiveOpnd = RIGHT_OPND;
    else
     newActiveOpnd = FIRST_LEFT_OPND;

    if (frag.nextActiveOpnd == INVALID)
     newNextActiveOpnd = INVALID;
    else
     newNextActiveOpnd = frag.nextActiveOpnd - frag.activeOpnd + 1;
   }

   newFrag = new Frag( newActiveOpnd, newNextActiveOpnd,
                       frag, newLibNode );
   // note that new Frag is disabled if active node is not the last added
   fragList.append( newFrag );
  }
 }
}
```

Figure 2.12: *Pseudo code: Combine fragments with next pattern node.*

Computing the Next Active Operand

Figure 2.13 shows the pseudo code for computing the next active operand of a fragment, assuming that the active operand has been advanced to `nextActiveOpnd` which therefore must be set to a new value. If `nextActiveOpnd` is invalid then all combinations with live operands of the fragment have been generated already and the next active operand remains invalid. If the current `nextActiveOpnd` is the left operand of the last added node then the `nextActiveOpnd` of the parent fragment can be used to compute the next active operand for `frag`. Otherwise, the `nextActiveOpnd` of `frag` and its parent refer to the same operand. Then we have to find the active operand after that by recursively calling `newNextActiveOpnd`. In both cases, the found `parentNAO` is transformed for the current fragment by Equation (2.1).

The recursion unwinds if either a fragment with an invalid `nextActiveOpnd` is encountered, or if a fragment has been found where the next active operand is attached to

```
int newNextActiveOpnd( frag ) {
 returnOpnd = INVALID;

 if (frag.nextActiveOpnd is valid) {
  if (frag.nextActiveOpnd == FIRST_LEFT_OPND)
   // found NAO => unwind recursion
   parentNAO = frag.parent.nextActiveOpnd;
  else
   parentNAO = newNextActiveOpnd( frag.parent );

  if (parentNAO is valid)
   returnOpnd = parentNAO - (frag.opndToParent) + 1;
 }

 return returnOpnd;
}
```

Figure 2.13: *Pseudo code: Compute next active operand.*

the last added node because then, the `nextActiveOpnd` of its parent fragment directly represents its own future `nextActiveOpnd`.

To understand why this is so, consider the family tree of fragments in Figure 2.14. Fragment 2 has been generated by combining fragment 1 with node B, and fragment 3 is the combination of fragment 2 with node C. From the way the active operands are marked we can conclude that the insertion algorithm is currently working on the left branch of node C. If we now set `activeOpnd` of fragment 3 to its next active operand and then try to find a new `nextActiveOpnd` we see that its parent's `nextActiveOpnd` is indeed identical with our old `nextActiveOpnd`: They both mark the left operand of node B. We therefore have to call the `newNextActiveOpnd()` procedure recursively with fragment 2.

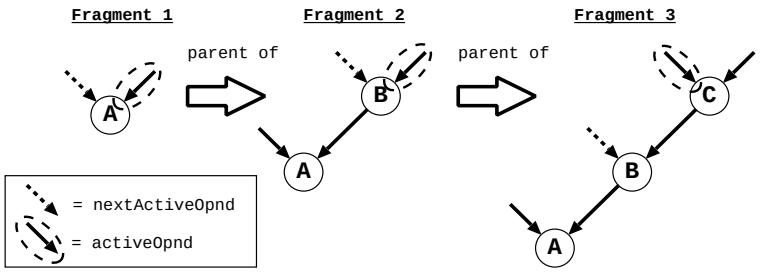


Figure 2.14: *Family tree of fragments.*

The active operand of fragment 2 is directly attached to the last node that has been added—compared with its parent. And indeed the parent's `nextActiveOpnd` is the future `nextActiveOpnd` we have been searching for. Therefore, the recursion unwinds at this point.

2.3.4 IOGs of DAG Patterns

Some steps in a DAG search in a PSG do not add another operation node to a pattern on the search path. Instead they attach another operand input to the output of a node that is already there. These steps cannot be reversed by applying ID operands. Therefore, the IOG for a DAG pattern does not necessarily contain the patterns on the search path in the PSG for the same pattern.

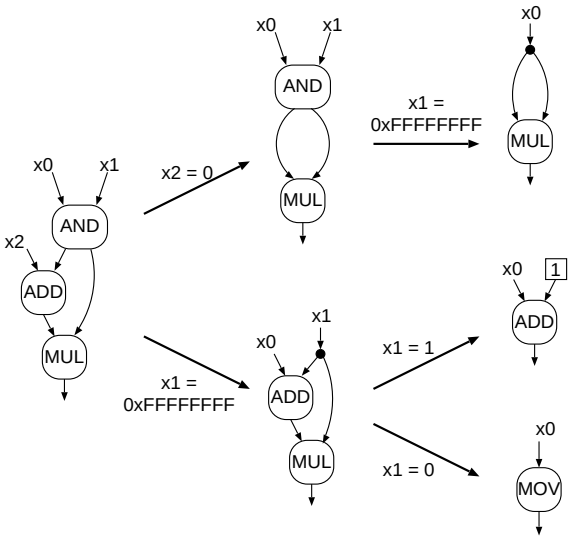


Figure 2.15: *IOG of a DAG pattern.*

The IOG of a DAG, however, is still useful to find substitution opportunities as in the tree case. For this purpose it can be constructed separately from the search graph. Figure 2.15 shows an example of the IOG of a DAG pattern. Unlike the tree-pattern IOG in Figure 2.8, the DAG-pattern IOG does not comprise the generic basic operations on level 1. The operations are more constrained, e.g., in two inputs being equal—in this case resulting in a squaring operation—or one input being a constant—in this case resulting in an increment operation. Furthermore, the IOG does not converge to a move operation at level 0.

The fragment technique used to construct the IOG for tree patterns is not easily adapted to DAGs. It appears more promising to construct it top-down by eliminating each node for which this is possible on each level of the IOG. In this work we will only build IOGs for tree patterns.

2.4 Library Size

2.4.1 Patterns in a PSG

As for the space per library entry, a linked-list library has to store a complete representation of the pattern with each entry. This amounts to a space of $n \cdot \mathrm{Space}_{\mathrm{patNode}}$ with n the total number of operation nodes of all patterns in the library and $\mathrm{Space}_{\mathrm{patNode}}$ the space per operation node. Moreover, there is some overhead for the list structure.

In contrast, the pattern represented by an entry in a PSG is given by the position of the entry in the library, i.e., by the path from the library root to the entry. Hence, the patterns do not have to be stored explicitly with each entry. What has to be stored with each entry are the arrays of `next`-pointers and, for DAG patterns, the `revisit`-arrays.

The number of possibilities to grow a pattern, i.e., the number of `next`-pointers stored with a pattern corresponds to the number of possible continuations of a search for larger patterns (see Section 2.2.2): the number of external left operands on the top left fringe of

the pattern, plus one right operand, times the number of primitive operators in the library.

$$\#_{\mathrm{NextPointers}} = (\mathrm{Operands}_{\mathrm{topleft}} + 1)\mathrm{Primitives}$$

Of the external right operands of a pattern, only the left-most has to be considered because the other right operands have been handled at lower levels of the search graph. The same is true for left operands outside of the top left fringe. In the worst case, all left operands in a pattern are on the top left fringe. Then the number of next-pointers for this pattern with p nodes is $(p+1) \cdot \mathrm{Primitives}$. Note that pointers that are NULL still have to be stored because they are a termination condition of the recursion in the search algorithm.

For each pattern in a PSG all patterns on the search path must also be present. In the worst case this could result in an overhead of as many additional library entries as operation nodes in a tree pattern. In practice, however, patterns on the search path will usually be present anyway as the pattern generator will find them as sub-graphs in the application graph, resulting in virtually no overhead entries at all. This reasoning is confirmed by our experimental findings in Chapter 5.

2.4.2 Analytical Bounds for IOGs

The patterns with the largest IOGs are those for which each operation node can be eliminated by its ID operand. This is the case for patterns that consist only of right-cyclops nodes with right identity operands, such as the pattern in Figure 2.8. These patterns form a sequence of nodes, each of which obtains its left operand from a previous node—except for the top leaf node—and that provides its result as the left operand to the following node—except for the root node. All right operands are pattern-external inputs through which the ID operands can be applied to eliminate any node at any level of the IOG. For the worst-case number of IOG entries, we consider patterns of this kind in the following.

The derivation of patterns from a parent pattern can be formulated as a combinatorial problem at each IOG level: On level k, how many different ways are there to choose k nodes out of the n nodes in the parent pattern? Hence, a pattern of n *different* right-cyclops nodes generates a graph of

$$\sum_{k=0}^{n} \binom{n}{k}$$

patterns through identity-operand transformations. This includes the parent pattern itself, the primitive operations, and the final *move* node. For the first pattern to be inserted into an empty library, this is also the number of new patterns for the library.

When a parent pattern that is being inserted has offspring patterns that are already present in the library then the number of new patterns that the parent introduces into the library is accordingly lower. Each sub-pattern that the library and the IOG of the new parent pattern have in common comprises the merge pattern where both graphs meet, and its complete cone of ID transformations down to the final *move* operation. For a merge pattern that comprises m right-cyclops nodes out of the n nodes of the parent, the number of additional patterns that are introduced to the library by the parent pattern is only

$$\sum_{k=0}^{n} \binom{n}{k} - \sum_{i=0}^{m} \binom{m}{i}.$$

If the library and the parent have more than one merge pattern then the IOGs of the merge patterns may overlap. In this case, the overlapping region must be subtracted only

once from the contribution of the parent to the library. For a parent that has two merge patterns with the library, comprising m_1 and m_2 nodes, respectively, and the IOGs of the two merge patterns merging at a pattern of m_3 nodes, the contribution of the parent to the library is computed by

$$\sum_{k=0}^{n}\binom{n}{k}-\left(\sum_{i=0}^{m_1}\binom{m_1}{i}+\sum_{j=0}^{m_2}\binom{m_2}{j}-\sum_{g=0}^{m_3}\binom{m_3}{g}\right).$$

The patterns with the smallest IOGs are those for which only one node can be removed at each transformation level. This is true for patterns that consist only of left-cyclops nodes of non-commutative operators. In these patterns, the only removable node is the leaf node because it is the only node with a pattern-external right operand. The IOG of such a parent has only one pattern at each level, namely, the pattern at one level higher without the leaf node. If the parent comprises n operation nodes its IOG will consists of n patterns and the *move* node. This minimum set of patterns resembles the search path of the parent pattern and is therefore equal to the parent's contribution to a PSG library.

Patterns that comprise multiple instances of the same operation will result in relatively small IOGs as redundant child patterns will occur only once in the IOG. Each duplicated operation node results in one primitive node fewer on level 1 of the IOG. Furthermore, duplicated operations will probably also result in redundant patterns on higher IOG levels. Each of these redundancies reduces the number of patterns in the IOG.

How many patterns a library will ultimately incorporate strongly depends on the composition of the parent patterns that have been inserted and therefore cannot be derived analytically.

2.4.3 Comparing IOGs with Unordered Libraries

An unordered pattern library only comprises parent patterns. Many offspring patterns in IOGs are also sub-graphs of the parent pattern. They would have been added to the library by a conventional library-construction algorithm as well. But there are other patterns in an IOG that are not sub-graphs of the parent and that therefore constitute an overhead compared with conventional libraries.

For each transformation step from a parent pattern to primitive patterns, a simpler pattern is a sub-graph of its parent if the operation node eliminated was a leaf or a root node, i.e., the eliminated node had only pattern-external inputs or its only output was a pattern-external output. Eliminating other nodes, i.e. cyclops nodes, always leads to connecting previously unconnected nodes. This new connection cannot occur in sub-graphs of the parent pattern. Hence, compared with an unordered pattern library, patterns that incorporate such a connection represent the overhead of an IOG.

The number of patterns in the IOG that are sub-graphs of the parent is equal to the sum of leaf nodes and removable root nodes of the parent pattern and those of all its generated sub-graphs. If a leaf that is being eliminated from a pattern feeds into a cyclops node then the child pattern generated has the same number of leaves. If the leaf feeds into an internal node then the child has one leaf less than its parent.

The patterns with the highest number of children that are not sub-graphs of the parent, i.e. the patterns with the largest IOG overhead compared with unordered pattern libraries, are again patterns of only right-cyclops nodes and a single leaf node which all represent different operations, such as the pattern in Figure 2.8. Each IOG level from the parent

pattern to level 1 has one real sub-graph more than the previous level—starting with one in the highest level. The other patterns are not sub-graphs of the parent and constitute the overhead. Subtracting the total number of sub-graphs from the total number of patterns in the IOG of a pattern results in the following formula for the worst-case IOG overhead for a parent pattern of n nodes:

$$\sum_{k=0}^{n} \binom{n}{k} - \sum_{i=1}^{n} i = 1 + \sum_{k=1}^{n} \binom{n}{k} - k$$

In practice, inserted patterns will have a significant number of internal nodes that cannot be eliminated. Moreover, they will comprise redundant sub-graphs that are inserted into the library only once. Consequently, the overhead of such patterns is significantly lower than in the worst case given here. This is confirmed by our experimental results in Chapter 5.

2.5 Summary of Pattern-Library Organization

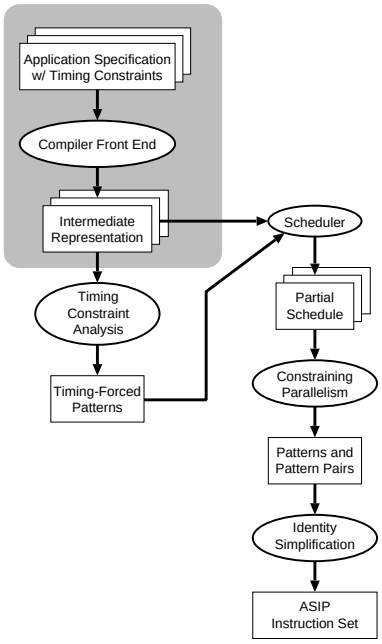
In this chapter we have presented a novel method to organize libraries of tree and DAG patterns by means of our pattern search graphs. Compared with conventional unordered libraries, PSGs enable more efficient searches with a computational complexity of $O(d)$ instead of $O(n \cdot p)$, with $d \leq p$. Our experiments presented in Chapter 5 confirm that our PSG libraries can be searched orders of magnitude faster than state-of-the-art libraries, with virtually no overhead in library size.

PSGs eliminate the dependency between computational complexity and library size and therefore can handle large pattern libraries. Current methods for library organization rely on heuristics that exclude less promising patterns in order to keep the library access time low. Given the memory size of today's workstations, our approach eliminates the need for such heuristics. Therefore, exact methods are now possible that do not risk to miss beneficial patterns that heuristics might eliminate. This is of particular importance for control-dominated applications where patterns that occur only rarely may be indispensable to meet fine-grained timing constraints.

Furthermore, we have introduced the concept of identity-operand graphs which reveal opportunities to substitute patterns by others. This can be exploited for instruction-set generation, resulting in a leaner instruction set with the same speed-up. Moreover, IOGs can be used during code generation to increase the number of opportunities for the use of specialized instructions, resulting in faster code. We also see an application of IOGs to find opportunities for datapath sharing in synthesis systems. In this area, IOGs significantly increase the application space for datapaths that result from pattern-merging methods such as [MAHM02, BKS04]. While the merging methods construct super-patterns our IOG shows all sub-patterns that can be implemented by the super-pattern employing identity operands. Moreover, configuring a merged datapath with identity operands does not require additional chip real-estate as opposed to reconfigurable interconnection networks or sets of multiplexers which are commonly used in the literature.

3 Compiler Methods for Fine-Grained Timing Constraints

This chapter discusses how to specify, represent, and transform the benchmark applications that represent the domain for which the ASIP is to be designed. It is these applications that will later be analyzed in order to derive the instruction set for the ASIP.



The control-dominated domain is characterized by fine-grained timing constraints. If an input register is not read in the right cycle it may be overwritten with a new value, and if an output is not generated on time, it may not have the intended effect. Moreover, control-dominated systems often have timing constraints that depend on input data at run-time, e.g., to process a stream of network-packet data depending on a header-length field.

For the automated design of an ASIP that meets these constraints, a designer must annotate the constraints in the benchmark applications. Common approaches to timing specification in behavioral HLLs provide only coarse-grained resolutions and data-dependent constraints cannot be expressed with current language constructs. The control-dominated domain, however, requires timing annotations with a precision of a single register access. This calls for a novel system of annotations. In order to facilitate the reuse of existing HLL code in the design process it is desirable that the constraint annotations do not require a redesign of the code. For the application analysis, the code with the annotations must then be transformed to a graph representation known as *intermediate representation (IR)* in compilers. The IR must represent all constructs of the HLL and it must support optimizations that help meet the timing constraints.

In this chapter we propose solutions to these problems. After a survey of related work in Section 3.1, we introduce a new method to express fine-grained timing constraints in ANSI C in a standard-compliant way in Section 3.2. The method includes a novel construct for data-dependent waiting. Section 3.3 presents our multi-layer IR which combines representations for data and control flow with a new layer of timing constructs, including a data-dependent wait. Section 3.4 describes the transformation from the timing annotations in C to the timing graph in the multi-layer IR. Section 3.5 demonstrates the expressiveness of our multi-layer IR by the example of an optimization technique that employs information from each layer in the IR to resolve scheduling conflicts. Section 3.6 summarizes this chapter.

3.1 Related Work

3.1.1 Specifying Timing Constraints

The fundamentals of the classification, specification and verification of timing constraints have been studied in [Das85]. Most methods found in the literature express minimum and maximum timing constraints as proposed there.

Interestingly, the classical hardware description languages (HDLs), such as VHDL [VHDL02] or SystemC [OSCI02], only have a basic notion of time to specify strict simulation timing. They do not provide constructs to specify minimum, maximum, or range constraints that allow optimizations for synthesis [EKPD95].

Various HLLs include constructs to specify timing constraints, e.g., an annotated version of Esterel [CPP+01] and Real-Time for Java [RTJ]. While ANSI C [KR88] does not provide any means to express timing information, there have been attempts to use C derivatives as HDLs, e.g., C^x [EB94] and HardwareC [KM90]. The programming style of these derivatives, however, significantly differs from ANSI C [KR88], e.g., in constructs to model parallel processes. Therefore, these languages require a fundamental rewrite of existing applications. Moreover, the derivatives introduce extensions that are not standard-compliant. Hence, the code can no longer be processed by common ANSI C tools. The extensions are introduced to make C a suitable HLL for ASIC synthesis, but they make the derivatives much more powerful and complex than necessary for ASIP design.

We take a different approach that is closer to ANSI C and also more suitable for an algorithmic coding style as opposed to the hardware focus of the HDL derivatives of C. Our approach is presented in Section 3.2.

3.1.2 Intermediate Representations

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.

In the data-dominated domain, the main optimization objective is to reduce the overall running time of an algorithm. In the pursuit of this goal it makes no difference in which section of the algorithm time is saved. Hence, loops are a promising optimization target because each cycle saved in a loop is rewarded multiple times if the loop is executed more

than once. In estimating the leverage factor of a loop, branching probabilities play an important role.

In [AC01] it was found that in the data-dominated domain, the compiler perspective 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 our control-dominated domain, we must meet hard timing constraints under all circumstances. In this environment, branch probabilities do not help because we must always assume the worst case. We propose to capture more information on the applications by going to a higher abstraction level and introducing a program representation that enables programmers to express more of their application expertise. For the loop-intensive data-dominated domain, however, the annotation method introduced in Section 3.2 could easily be extended to express branching probabilities or value ranges for variables.

The most commonly used models for hardware/software co-design, namely FSMs, discrete-event systems, Petri nets, communicating processes, and synchronous/reactive models [CEP99], as well as several derivatives [CGH+94, TTSV00a, TTSV00b], have been inherited from the hardware domain. Consequently, they assume an independent type of concurrency with reactive processes starting and running independently of each other. Scheduling such concurrent specifications for single-thread processors is non-trivial [SLWSV99, WBC+00, CPP+02].

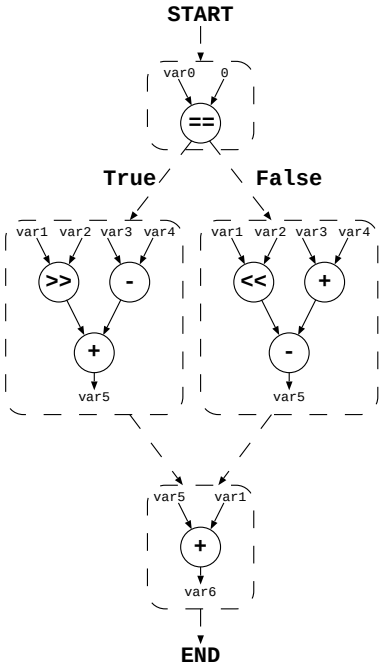


Figure 3.1: *A simple CDFG.*

On the instruction level of an ASIP, in contrast, we do not have this independent type of concurrency. Assuming a VLIW architecture, instructions are *statically* scheduled in parallel. Therefore, a standard compiler IR, such as a control/data flow graph (CDFG), is better suited for an ASIP design system, providing well known transformations from C to the IR and a plethora of available optimizations. A CDFG is a combination of a control-flow graph (CFG, a.k.a. flow graph) and data-flow graphs (DFGs). Figure 3.1 shows a

simple example of a CDFG with DFGs inside the dotted CFG nodes. For an overview of other compiler IRs the reader is referred to [Muc97, Mor98, Wea95].

A CDFG can be transformed into the static single assignment (SSA) form [CFR+91] in which each variable use has exactly one definition. Furthermore, the SSA form makes data dependencies across control flow boundaries explicit and expands the scope of transformations.

An example of a graph representation of timing constraints is the output transition graph (OTG) as introduced for controller FSMs in ASICs [NT92]. The constraints are represented by edges between output events and the edges are annotated with the minimum and maximum time between the operations. Scheduled nodes are annotated with their associated control step.

A CDFG must be extended to capture fine-grained timing constraints to be suitable for the control-dominated domain. The extensions we propose are presented in Section 3.3.

3.2 Integrating Timing Constraints into ANSI C

Most existing software for embedded systems is available in C only, tested and well understood. But C does not provide means to express any kind of timing information [KR88]. Re-implementation of all applications in another language for the ASIP design process is not a viable option. Therefore, we need an extension to C that enables the ASIP designer to supplement existing software with timing constraints without requiring a major rewrite of the code. In this section we describe a method to integrate timing information into C code in an ANSI-C-compliant way.

In data-dominated systems, such as DSPs, processing often starts with receiving a sample of data, called a *frame*, and ends with sending out a resulting frame [MBL+96]. 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. Each of these I/O interactions handles a different type of information that, hence, needs to be processed by a different part of the software, resulting in many different fine-grained deadlines at many different points in the application.

In particular with a data-push architecture, every relevant value in an input register 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 value. Furthermore, different input data may have completely different semantics and trigger different types of processing, e.g., a header-length field vs. a protocol number. Hence, we need to specify fine-grained timing constraints in many places in the application.

A timing constraint is defined by the following set of information [Das85]:

- The two points in the code, namely two instructions, between which the constraint applies.
- The minimum time that must elapse between the execution of the two instructions.

- The maximum time that must not be exceeded between the execution of the two instructions.

We found that the time to pass between two operations can also depend on input data, i.e., the value in an input register may encode a time that must pass between two events.

The unit in which the time is given can be seconds or clock cycles of the ASIP to be designed. Time values in seconds will have to be rounded to a multiple of the cycle time of the ASIP once this has been determined. Time values in clock cycles require that the cycle time of the ASIP be determined already when specifying the constraints.

In the following we assume time values in clock cycles because in a data-push architecture the cycle time corresponds to the communication rate with the environment, i.e., the rate by which the input registers are written and the output registers are read by the environment. Therefore, the cycle time will be part of the systems requirements and is known at the beginning of the ASIP design process.

All timing constraints are positive numbers. They are always specified in the same direction as the control/data flow. A timing constraint that spans a loop is treated as if the loop is not taken because the execution time of a loop is unbounded and therefore cannot be covered by timing analysis. The only exception is a timing constraint that spans exactly one iteration of a loop, i.e., a constraint that starts and ends at the same operation. Such a constraint represents a rate constraint on the loop.

3.2.1 Fixed Timing Constraints between Operations

To mark the points in the code that are hooks for constraints, we use standard C labels. As their only purpose in C is to mark jump targets, they do not alter the behavior of the code. A complete *labeled statement* in C comprises an identifier with a colon followed by the statement it marks. We define reserved "`START`" and "`END`" labels as hooks for timing constraints relative to the start or the end of a program.

To convey minimum and maximum time between labels we need to pass values to the compiler. ANSI C provides `#pragma` statements to pass more information to a compiler than has been defined in the standard. The compiler designer can freely define the syntax of what follows the `#pragma` token. If a compiler can parse this syntax it can use the extra information for the compilation process. If a compiler does not understand a `#pragma` it encounters, the standard requires it to ignore the statement. Accordingly, `#pragma`-annotated code can still be processed by any ANSI C compiler.

We define a `#pragma` syntax to express timing constraints. The first statements specify a minimum or maximum time:

```
#pragma mintime <src_label> <dest_label> = <time>
#pragma maxtime <src_label> <dest_label> = <time>
```

In our `#pragma` syntax, `src_label` and `dest_label` are the names of the C labels between which the constraints apply. The amount of time is given by `time`. To conveniently specify both minimum and maximum time at the same time, we define:

```
#pragma time <src_label> <dst_label> = <mintime> <maxtime>
#pragma time <src_label> <dst_label> = <time>
```

The second statement sets the minimum and maximum time to the same value. Both statements are shorthand for combinations of the two `#pragma` statements defined before.

The combination of C labels and `#pragma` statements enables a programmer to specify timing constraints between two C statements. A statement in C, however, can comprise multiple basic operations. Therefore, we need to improve the resolution of the labeling.

The actual time-critical part of an application is the communication with the environment. In a data-push architecture, this communication has the form of read and write accesses to I/O registers. In order to attach timing constraints to these accesses, we introduce another `#pragma` statement to declare that a particular variable name represents an I/O register:[1]

```
#pragma io <procedure>::<variable>
```

The name of the variable is given by `variable`, and `procedure` gives the name of the procedure in which the variable is declared. Within a C statement identified by a label, a timing constraint will now be attached to that basic operation which accesses an I/O variable, as identified by a `#pragma io`. We have thus devised a method to provide a coarse-grained HLL with fine-grained timing constraints.

The programmers must ensure that only one I/O variable is accessed in such a C statement. They can achieve this by splitting statements with more than one I/O variable and introducing new variables for intermediate results. For instance, if `in` and `out` are I/O variables, the statement

```
out = in + 5;
```

with two I/O variables can be split into

```
temp = in + 5;
out = temp;
```

resulting in two statements with only one I/O variable each, as required.

With a resolution of a single IR operation we can now specify where exactly the timing constraints apply. For $mintime = maxtime = 0$, the identified operations must be scheduled in the same cycle. For $mintime = maxtime = 1$, the second operation has to be scheduled in the cycle following the first operation. With $mintime = 0$ and $maxtime = 1$, the operations are scheduled either in the same cycle or one cycle apart. Timing constraints with larger values are interpreted in the same fashion.

Note that the timing-critical action is only the access to I/O variables, i.e., I/O registers. In contrast, algorithmic operations are not directly observable from the outside of the ASIP system and their timing is therefore only relevant where they feed I/O operations that have a timing constraint. An algorithmic operation derives a latest possible execution time from such an I/O operation if

- the I/O operation has a data dependency on the algorithmic operation, or

[1] It might be more elegant to use the `register` keyword in C to declare I/O variables. However, we use the SUIF2/Machine-SUIF compiler framework [SUIF, MS] for our implementation and the only available C front-end for SUIF2 does not transform `register` statements correctly.

- the result of the algorithmic operation is needed to compute an execution condition of the I/O operation.

Hence, timing constraints do not necessarily include any algorithmic operations on the values of these registers. To meet the timing constraint of a read access to an I/O register it is sufficient to save the register value to an internal register before the I/O register is overwritten by the environment. This fact can be exploited in operation scheduling.

3.2.2 Data-Dependent Wait

In control-dominated applications timing constraints are often data-dependent, i.e., a system input determines the time required between two events. Data-dependent delay operations have been proposed for high-level synthesis to model communication with the environment or conditional blocks whose total execution time depends on a runtime condition because one branch takes longer than the other [KM92]. We extend this concept by an explicit wait operation. This operation has one operand that is computed at runtime and specifies a time to wait in clock cycles.

An example scenario requiring a data-dependent wait is the task of finding the beginning of a TCP packet header after a variable-length IP header in a network processor [Dit00]. The length is encoded in a header field and its value corresponds to the number of input words to bide before the TCP header appears at the network interface.

To express such a dependency between input data and timing, we introduce another `#pragma` statement:

```
#pragma wait <src_label> <dest_label> <variable> <min_val>
```

Here, `variable` is the variable that determines the number of cycles to wait. The labels `src_label` and `dest_label` mark the points in the code between which the wait time must elapse. The minimum value that `variable` can possibly have, by the programmers expertise, is provided by `min_val`. This value provides the freedom to schedule the beginning of the wait anytime between *immediately* and `min_val`. We will use this freedom in Chapter 4.

Note that the `variable` must be valid in the scope of the `dest_label` so that its value is accessible to the wait operation. There must be only one assignment to the `variable` to avoid ambiguities. Moreover, the operation at `dest_label` is not executed in the same cycle when the wait triggers, but will be scheduled in the *next* cycle. Therefore, `min_val` must not be less than 1 to allow for this one cycle delay.

3.2.3 Code Example

For an example of C code with timing annotations, consider the program in Figure 3.2. The variables `data_in` and `data_out` are marked as I/O variables by the first two `#pragma` statements. The next two statements specify a timing constraint between `label1` and `label2`, and introduce a data-dependent wait between `label1` and `label3`, respectively, with a minimum wait input of 5.

The timing constraint between `label1` and `label2` is zero cycles. This means that the I/O variables in each statement must be accessed in the same cycle. The accesses in

```
#pragma io main::data_in
#pragma io main::data_out

#pragma time label1 label2 = 0
#pragma wait label1 label3 counter 6

int main(int argc, char argv[]) {
  int counter;   /* wait register   */
  int data_in;   /* input register  */
  int data_out;  /* output register */
  int temp;

 label1:
  counter = ( (data_in & 0x0f000000) >> 24) + 5;
 label2:
  temp = (data_in & 0x00ff0000) >> 16;

 /* wait on counter */

 label3:
  data_out = temp;
}
```

Figure 3.2: *Example of C code with timing constraints.*

both cases are reading from `data_in`. The timing critical operation is to read the I/O variable before it is changed by the environment. The computations that have `data_in` as an operand are not affected by this timing constraint and can be scheduled in a different cycle.

The value that is assigned to `counter` determines the number of cycles to pass between reading `data_in` at `label1` and writing the resulting value of `temp` to `data_out` at `label3`. The minimum value for `counter` is given as 5 because even if `data_in` is 0 the addition of 5 in the computation of `counter` makes 5 the minimum value and the programmer here thinks that the parantheses will always yield at least 1.

3.3 Multi-Layer Intermediate Representation

In this section we propose a novel intermediate representation (IR) that carries more information than the IRs do that are commonly used in ASIP design. The additional information includes the timing constraints, expressed in C as described in Section 3.2.

IRs have a graph structure with nodes and directed edges that represent dependencies between nodes. Nodes and edges are annotated with necessary information. For control-dominated applications we need to express the following information:

- data dependencies for computations using results of other computations in the same basic block;
- control dependencies that determine the control-flow through an application;
- data dependencies across control-flow boundaries to support optimization, and

- time dependencies to express timing constraints and synchronization with the environment, including our data-dependent wait construct.

The IR commonly used in the literature on ASIP design is a CDFG. The employed pattern-finding algorithms are constrained to the data-flow graphs (DFGs) within a basic block of the CDFG. In control-dominated applications, the basic blocks are very small which significantly constrains the effectiveness of such pattern finding. Furthermore, timing information is not part of a CDFG at all. To overcome these restrictions, we combine graph notations that have the required properties, forming a new IR with several layers, and new concepts, such as data-dependent wait operations. We call it the *multi-layer Intermediate Representation (mlIR)*.

3.3.1 Data-Flow Layer

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

3.3.2 Control Layer

FSM-based IRs require an FSM-based language for application specification [CGH+94, TTSV00b]. In contrast, the benchmark applications that are used in ASIP design will mostly be specified in procedural languages, such as C. Common compiler front-ends transform the control flow of these languages into a CFG representation. Hence, the control layer of our IR is also based on this graph type. Moreover, a CFG representation gives access to standard compiler transformations and optimization runs.

The nodes in a CFG represent basic blocks of operations. In a sequential program a new basic block begins after each branch instruction and before each branch target. The computations within a basic block are represented by DFGs in the data-flow layer that are associated with the according node in the control layer. In practice, there will be dummy nodes representing the start and end of a control node that will connect to all leaves and roots of the enclosed DFGs. In the interest of clarity, however, we will omit these dummy nodes in our figures.

The edges of the control layer show where the control flow 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-else 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 always be provided to prevent deadlocks. The control graph is delimited by two empty nodes, the start node and the end node.

3.3.3 Meta-DFG Layer

As the DFGs in control nodes 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 their extension across control-flow boundaries will allow a more 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.

The meta-DFG is essentially a graphical representation of the static single assignment (SSA) form [CFR+91]. The multiplexers correspond to the SSA ϕ-functions and the meta-DFG edges represent the connection between the definition and the use of variables—the def-use chain.

3.3.4 Timing Layer

In Section 3.2 we described a method to specify fine-grained timing constraints in ANSI C. In order to capture these constraints in the multi-layer IR, we introduce a timing layer that can be compared to output transition graphs for FSMs [NT92]. Graph edges in the timing layer are annotated with the minimum and maximum time between nodes. Scheduled nodes are annotated with the determined time step.

The nodes in the timing layer between which timing constraints exist are start and end nodes, I/O nodes, and wait nodes. I/O nodes represent communication with the environment of the ASIP, i.e., read or write accesses to variables that have been declared as I/O registers in the C code by a `#pragma io`. I/O nodes act as operands to operation nodes in the data-flow layer.

In order to provide a representation of the wait statements introduced in Section 3.2.2, our multi-layer IR features a new type of operation node that connects the DFG layer with the timing layer. We call this node a *wait node*. It has one DFG edge as an input whose value determines the delay the node represents, given in clock cycles. Furthermore, to be meaningful, a wait node must have at least one incoming and one outgoing timing edge because it provides a delay between two other nodes. Finally, a wait node also represents a control construct in that it blocks the control flow until its timer triggers. Therefore, just like a branch instruction, a wait node ends a basic block, and a control edge connects it to the next basic block.

3.3.5 Putting it all Together

We combine DFG, CFG, meta-DFG, and our timing layer 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 on the corresponding layer. Modifying adjacent edges of a node in one layer does not affect the edges of another layer. Figure 3.3 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 are annotated with the minimum and maximum number of times they are taken in one run, or DFG edges have constrained value ranges imposed on the variables they represent. The

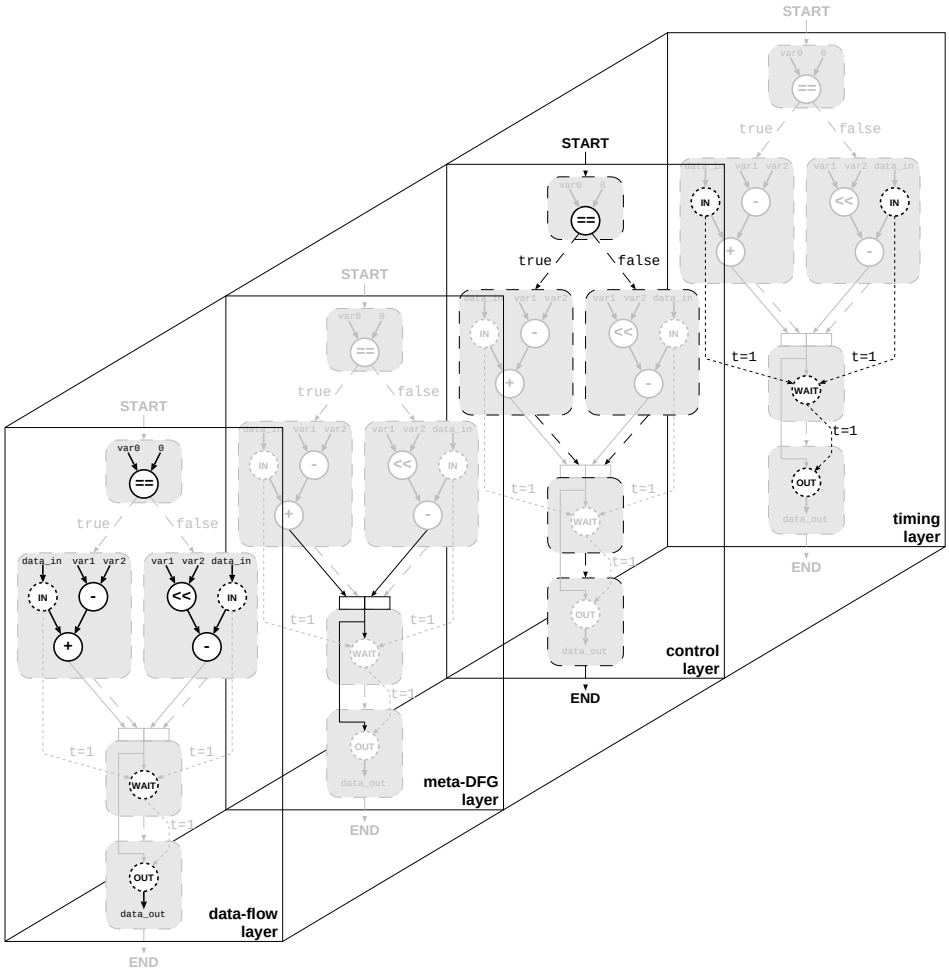


Figure 3.3: *Example of multi-layer IR graph.*

information is given either explicitly by a programmer, using further `#pragma` statements, or is derived from program analysis, e.g., by back-annotation of profiling results.

The proposed multi-layer IR fulfills all requirements postulated at the beginning of Section 3.3 and it captures all information that the `#pragma`-annotated C code provides.

The transformation of C source code to CFGs and DFGs, i.e. to a CDFG, is standard technique performed by many C compilers [ASU86]. Algorithms have been proposed to derive the SSA form for the meta-DFG layer from a CDFG [BCHS98]. The following section describes the transformation of timing annotations in the C source code to the timing layer in the multi-layer IR. This completes the construction of the multi-layer IR from the C source code.

3.4 Timing Layer Transformations

In this section we show how to parse the `#pragma` timing-annotations to construct the timing layer, and how to generate code from wait nodes.

```
IoPragma → #pragma io id:procedure :: id: variable
 { symbolTable. findVariable ( procedure , variable ). isTimingHook = true; }

Label → id :
{
 while true {
  proceed to next statement ;
  for each operand in statement
   variable = symbolTable. findVariable ( currentProcedure , operand.name )
   if ( variable .isTimingHook )
    break;
 }
 symbolTable. findLabel ( id ). timingHook = operand;
}

TimePragma → #pragma time id:srcLabel id : dstLabel number:time1 number:time2
 { new TimingEdge( srcLable.timingHook, destLable .timingHook, time1, time2 ); }
             | #pragma time id:srcLabel id : dstLabel number:time1
 { new TimingEdge( srcLable.timingHook, destLable .timingHook, time1, time1 ); }
             | #pragma mintime id:srcLabel id : dstLabel number:time1
 { new TimingEdge( srcLable.timingHook, destLable .timingHook, time1, null ); }
             | #pragma maxtime id:srcLabel id: dstLabel number:time1
 { new TimingEdge( srcLable.timingHook, destLable .timingHook, null , time1 ); }
```

Figure 3.4: *Parser productions for the basic timing constructs.*

3.4.1 Parsing Timing Annotations in C

The `#pragma` statements defined in Section 3.2 have to be transformed to elements of the multi-layer IR by the parser of a compiler front-end. Figures 3.4 and Figure 3.5 give the productions that have to be included in the context-free grammar of a C-language parser [ASU86] in order to create the timing layer of the multi-layer IR.

Figure 3.4 shows the productions for the basic timing constructs. The first two productions generate the hooks to which timing edges can be attached. These hooks are points where an I/O variable is accessed, i.e., where it is used as an operand. An I/O variable is declared by a `#pragma io` and we mark each of these variables in the symbol table as a potential timing hook. Subsequently, for each label in the program we search the first access to an I/O variable in the code following the label. The operand that constitutes this access is then stored in the symbol table as the timing hook with that label.

After the timing hooks have been generated we create a timing edge for each `#pragma time`, `#pragma mintime`, and `#pragma maxtime` between the timing hooks associated with the given labels. We annotate the edge with the minimum and maximum timing values given by the `#pragma`.

In the production for a `#pragma wait` in Figure 3.5 we first locate the only assignment to the input variable that determines the time to wait. We start the search from the beginning of the current scope. Once found, we create a DFG edge from this assignment to the operand input of a new wait node. Furthermore, we create timing edges between the wait node and the timing hooks associated with the `src_label` and the `dest_label`. The instruction following the wait node cannot be fetched before the next cycle. Therefore, the timing edge *from* the wait node to `dest_label` has $mintime_{out} = maxtime_{out} = 1$. The other time values must be adjusted as follows.

```
WaitPragma → #pragma wait id:srcLabel id:dstLabel
                          id: variable number:minTime
{
 go to scope.start;
 while true {
  proceed to next statement;
  if ( statement contains assignment to variable ) {
   waitInput = statement.targetOperand;
   break;
  }
 }
 newWait = new Wait( waitInput );
 new SubtractionNode( waitInput, minVal, newWait.input );
 offset = minVal − 1;
 new TimingEdge( srcLabel, newWait, offset, offset );
 new TimingEdge( newWait, destLabel, 1, 1 );
}
```

Figure 3.5: *Parser production for the wait pragma.*

The constraint on the timing edge from `src_label` *to* the wait node as well as the wait-time input are dependent on the minimum value of the wait time as provided by `min_val` in the `#pragma wait`. Subtracting the one cycle for the outgoing edge, we get $mintime_{in} = maxtime_{in} = min\_val - 1$. This timing edge, however, requires the wait to be started only after the minimum wait time has elapsed already. We therefore need to adjust the wait-time computation accordingly and insert a subtraction node between the wait input variable and the wait node. We subtract the minimum value from the wait input t_{wait}, yielding t'_{wait}. With the subtraction of min_val the minimum input value of the wait node t'_{wait} is zero. The time between `src_label` and `dest_label` then is

$$t = mintime_{in} + (t_{wait} - min\_val) + mintime_{out} = (min\_val - 1) + t'_{wait} + 1.$$

The scheduler can trade off time before and after the wait by subtracting time from one edge and adding it to the other. Furthermore, it can trade off time at the incoming timing edge against the wait-input offset: the lower the time of the incoming edge the larger the wait input.

As an example of the timing-layer transformation, Figure 3.6 shows the mlIR result of the example code in Figure 3.2. Note how the `min_val` specified in the code has been distributed across the two timing edges connected with the wait node. The offset at the wait input has been adjusted accordingly.

Following the above scheme, we have extended the existing C front-end of the SUIF2/ Machine-SUIF compiler framework [SUIF, MS] to translate the `#pragma` statements to the corresponding graph structures in our multi-layer IR [Lae03].

3.4.2 Implementing a Data-Dependent Wait

The wait node does not translate directly into a primitive processor instruction. Instead, it is transformed into one of two possible implementation types as the example in Figure 3.7 demonstrates. The wait node, depicted on the left-hand side, can be implemented

1. entirely in software, as shown on the right-hand side of Figure 3.7, by moving a start value `data_in` into a register `wait`, decrementing this register in appropriate

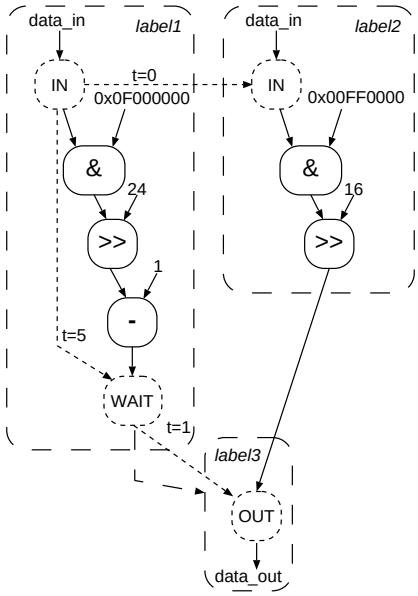
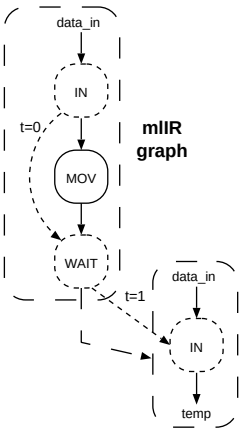


Figure 3.6: *Example code transformed to mlIR.*

intervals with an explicit subtraction, and branching when the register reaches zero, or

2. partially in hardware, as shown in the middle column of Figure 3.7, by providing a counter register that is decremented implicitly by a constant value—typically by one—and compared with zero in each clock cycle. The counter is set by writing a value into the counter register. Then the processor is stalled by a special control instruction, which we call *wait-for-counter (WFC)*, until the counter reaches zero. The WFC instruction stops the execution of all instructions at the end of the cycle in which it has been issued. When the counter triggers, the processor resumes execution in the following cycle.



mlIR graph

with hardware counter	*control step*	software count-down
`counter := data_in` `WFC`	*1*	`wait := data_in` `if wait = 0 goto CONT`

`temp := data_in`	*2*	`LOOP:`
		`wait := wait - 1`
		`if wait > 0 goto LOOP`

	3	`CONT:` `temp := data_in`

Figure 3.7: *Wait-node implementation.*

The software implementation requires more instructions in the application code for the repeated subtractions and tests for zero. Each additional instruction complicates the instruction-scheduling process. Note also that updating the wait variable and testing it for zero have to be scheduled in the same cycle.

The hardware solution, on the other hand, relies on additional infrastructure. Moreover, a counter can be used for only one wait node at a time. This, however, is not a severe constraint as a wait stalls the entire processor for an unbounded time, and hence there cannot be two waits in parallel.

Using the hardware counter in the application code requires two instructions: a *move* to set the start value of the counter, and the WFC instruction. Writing the start value to the counter, however, needs to be scheduled exactly in the cycle required by the timing edges that lead to the wait node. Otherwise, the counter would not go off at the intended point in time.

For both implementations there is 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.

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. The scheduler needs to be aware of operations that implement wait nodes in the application so that it can decide whether it should introduce an adjustment node.

Moreover, the scheduling freedom of the counter start depends on the minimum possible start value of the counter. This value determines the time after which the counter must be tested for zero for the first time. It is the latest possible start time of the counter—even with adjustments. The designer must specify the minimum value of the wait input, using the `min_val` of our wait-pragma. The larger this value, the larger the scheduling freedom. In Section 4.4.2 we will employ this method to schedule waits.

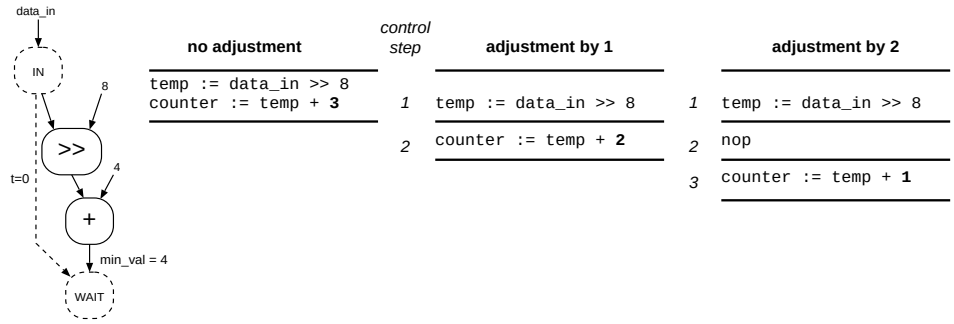


Figure 3.8: *Wait-node adjustment for scheduling.*

Figure 3.8 shows an example of a wait node with three possible schedules. The `min_val`, after subtraction of 1 for the transition to the operation following the wait, allows an adjustment of up to 3. The adjustment value is subtracted from the operand of the addition node. In the first implementation with no adjustments, both instructions must be executed in the same cycle to start the counter correctly. Using adjustments this situation can be relaxed. With an adjustment value of 2, there is even an idle slot, marked by the `nop`, that can be

filled with a productive instruction. The WFC instruction is not shown here because it does not have to occupy an instruction slot as explained in Section 4.4.3.

For a software implementation of a wait node, an enumeration of the possible delay values by the programmer offers another optimization opportunity. Gaps between the values correspond to scheduling slots in which the register used for the countdown does not have to be decremented or tested for zero. To compensate the gaps, the counter merely has to be decremented by a higher value later.

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

3.5 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 Section 3.2 we motivated the introduction of a timing layer with the fine-grained timing constraints that are characteristic of control-dominated applications. With several deadlines in short sections of code, the need for fine-grained timing optimization arises. An example of a problem that can occur is given in Figure 3.9.

On the left-hand side, condition computation, branch, and then-clause are all scheduled in the same time step X. Assume that the then-clause 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 is similar to speculative execution in that it changes the execution order of a conditional branch and subsequent 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. Reverse speculation [GSK+01] does the opposite by moving unconditional operations into conditional basic blocks. The objective is, again, to minimize the over-all critical path.

In contrast, branch postponing improves the schedulability rather than the average or maximum execution time. It might even grow the critical path through the else-clause. But it allows code to be scheduled that otherwise could not meet its timing constraints. We do this independently of what the average case is and hence, we make no assumptions on branch probabilities.

3.5.1 The Algorithm

The first step to solve the problem in Figure 3.9 is to move the condition computation to the preceding time step, as shown on the right-hand side. 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

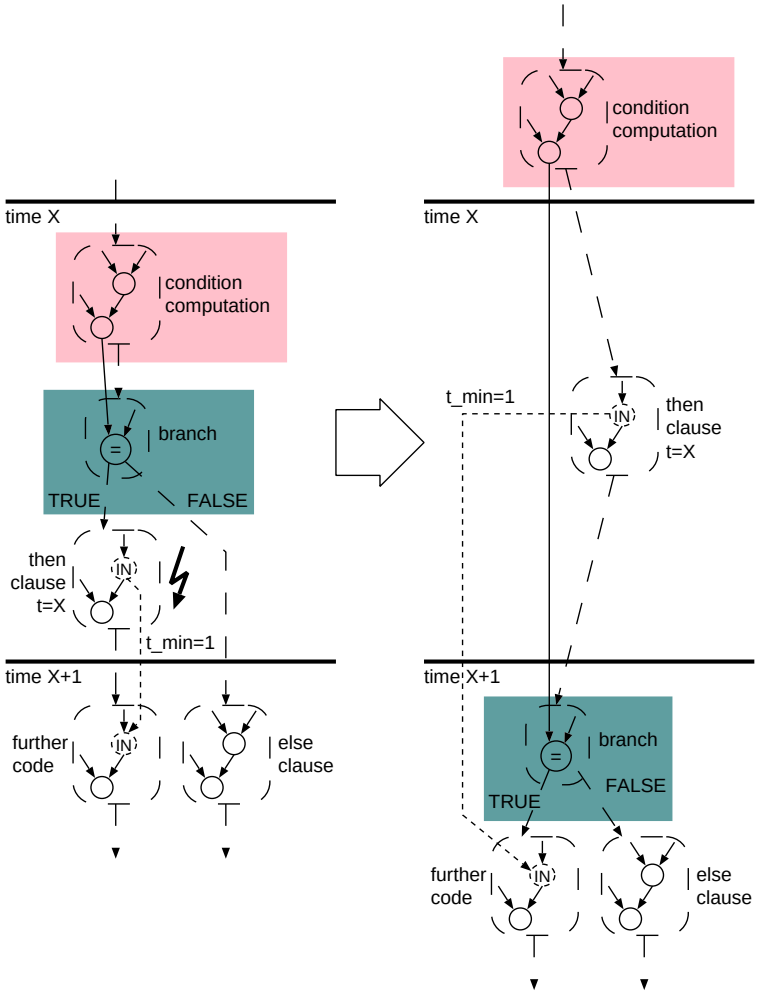


Figure 3.9: *Branch postponing.*

that would mean to move the branch after a code section that should only be executed if the branch has actually been taken, as also shown on the right-hand side.

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

- no output to the ASIP environment occurs in the DFG nodes of the then-clause because this communication is part of the program result that must not be altered by the transformation;

- no memory writes occur in the DFG nodes of the then-clause 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-clause with read addresses in the else-branch. This can, however, be a complex task because of the memory alias problem of two different expressions that denote the same memory location.

```
while ( (latency of BBs) > (time between deadlines) ) {
  while ( time slots available before first deadline )
    push mobile BBs beyond first deadline;
  push mobile BBs beyond second deadline;
  for each remaining BB (in bottom-up order)
           with a conditional branch {
    if ( then-clause must stay before second deadline
       AND then-clause is harmless to else-clause )
      move then-clause before branch;
    if ( else-clause must stay before second deadline
       AND else-clause is harmless to then-clause )
      move else-clause before branch;
    if BB is now mobile
      push BB beyond second deadline;
  }
}
```

Figure 3.10: *Pseudo code: Branch postponing.*

In control-dominated applications this situation occurs frequently, for instance, when 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.

A basic block is *mobile* if it can be pushed beyond a deadline without violating given timing, data, or control dependencies. A generic pseudo-code representation of the branch-postponing algorithm for a set of basic blocks (BBs) between two deadlines is shown in Figure 3.10.

Note that branch postponing adds only little to the critical path in the else-branch, because the else-clause in Figure 3.9 would in any case have to wait for time $X + 1$ to arrive owing to the given minimum time distance to the then-clause of 1. The time added by moving the branch to the same time step in many cases is not critical, 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-clause to be scheduled, which otherwise could not be accommodated.

3.5.2 Applicability and Relevance

To illustrate the relevance of branch postponing in a real-world example, we compiled the header-compression code in [Jac90] 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 closer examination, we implemented the compress routine in the multi-layer IR. The routine handles only common-case packets and delegates error handling to another processing entity. The target ASIP is a protocol engine with a data-push architecture as introduced in Section 1.3.3 as part of a network processor. Therefore, the deadlines between two reads of header fields are very tight; for instance, with a 32-bit input register and a network data-rate of 10 Gb/s the time between two header words is only 3 ns. Under such tight constraints, any gain of scheduling freedom is highly valuable.

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, also be applied to the remaining conditional branches.

We examined each conditional branch in the compress routine that is locked between deadlines of one or two input cycles. According to the example above, this equals deadlines of 3 to 6 ns. Table 3.1 shows the number of operation nodes that branch postponing will move out of the critical path at these branches. In this way, the operation count in the examined sections is reduced by 36 to 93%. The remaining nodes will at least move header words from the input register to another register before the input register is overwritten with the next header word. With the scheduling freedom gained, moved operation nodes can then be scheduled in less timing-critical sections.

Branch no.	1	2,3	4	5	6,7	8
Movable nodes	13	8	3	7	10	3
Remaining nodes	1	3	2	2	2	2
Deadline in cycles	1	2	1	1	1	1
Improvement per cycle	93%	36%	60%	78%	83%	60%

Table 3.1: *Scheduling freedom through branch postponing.*

In consequence, the required deadlines can be met. Thanks to the better balancing of the number of operation nodes per cycle, the circuit may even be clocked faster than projected.

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 analyze whether the then-clause block is harmful.
- The meta-DFG layer makes data dependencies between control nodes obvious. Therefore, possible conflicts when moving the branch are found on this layer.

The branch-postponing algorithm demonstrates the potential of combining information in the multi-layer IR.

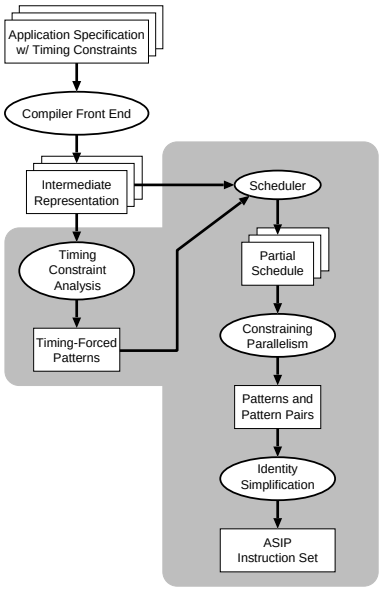
3.6 Summary of Compiler Methods

In this chapter we have introduced a new method to annotate existing C code with fine-grained timing constraints and data-dependent waits, both of which are typical for control-dominated applications. Unlike other known methods, our approach does not require a significant rewrite of the code. Hence, existing application code for which an ASIP is to be designed can be used as-is and only needs to be annotated with the timing requirements. Moreover, our method increases the resolution of the timing constraints to a single basic IR operation on the level of a coarse-grained HLL, compared to a resolution of an entire C statement in other approaches.

Our multi-layer IR captures the timing constraints provided by the C annotations and we provided parser productions to transform the annotations to the timing layer. In the timing layer we introduce the new concept of a wait operation that delays the program execution for a number of cycles computed at run time. The wait operation simplifies the timing specification of I/O operations for the communication with the environment and enables automated scheduling support for the implementation of the specified timing.

Based on the multi-layer IR we introduced branch postponing, a novel variation on speculative execution. The method resolves a type of scheduling conflict that occurs frequently in control-dominated applications due to tight timing constraints. Branch postponing combines information from all layers of our IR and demonstrates the potential of the multi-layer IR.

4 Instruction-Set Generation for Precise Timing



In this chapter we introduce our algorithms to choose patterns for implementation as special instructions. The resulting instruction set has to enable an implementation of the benchmark applications such that two kinds of constraints are met:

- the timing constraints given by the timing layer of the multi-layer 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 goal 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 goal minimizes the number of bits needed for the instruction encoding.

We formulate these optimizations as two consecutive scheduling problems. The first problem is to segment each path that is 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 problem 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.

This chapter is organized as follows: Related work on operation scheduling is presented in Section 4.1. In Section 4.2 we introduce a method to find the timing-forced patterns that must be implemented as instructions in order to meet the specified timing constraints. A path between the start and end point of a timing edge in the application graph is segmented such that the timing constraint will be met if each of the segments is implemented as a one-cycle instruction. In Section 4.3 the number of parallel instruction issues is constrained to the number specified by the designer. Our method generates a partial schedule with unbounded parallelism and then bundles patterns that are frequently used in parallel to compound instructions. The process takes into account the effect that the bundling has on scheduling freedom. How wait operations and other control constructs are handled in the methodology is the subject of Section 4.4. Section 4.5 summarizes the chapter. To our knowledge, this is the first complete instruction-set design flow for control-dominated applications.

4.1 Related Work

4.1.1 Operation Scheduling

The simplest scheduling algorithms used in high-level synthesis 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 [PG87]. ASAP and ALAP schedules do not take resource constraints into consideration.

An early algorithm for resource-constrained scheduling is *list scheduling (LS)* [Hu61]. 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 [CSS98, ACD74].

Another popular algorithm is *force-directed scheduling (FDS)* [PK89]. 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 [PK91] and the formulation of scheduling as an integer linear program (ILP) [LHL89], which has been extended to include parallel scheduling of instructions for VLIW processors [KW01]. Solving an ILP provides optimum schedules. However, it is generally an $\mathcal{NP}$-complete problem and therefore it is practical only for small problems.

A comprehensive introduction to the scheduling problem followed by a survey of the most popular scheduling algorithms for high-level synthesis can be found in [WC95].

4.1.2 Scheduling with Timing Constraints

For scheduling applications with timing constraints it is assumed that the specification of the timing constraints is feasible, consistent, and complete, i.e., minimum values are not larger than maximum values, there are no contradictory constraints, and all timing-critical paths in an application graph have associated timing constraints. Methods to ensure these properties have been presented in [KM92, GM97].

Most HLS systems do only allow for the specification of static timing constraints. They do not consider minimum, maximum, or 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 [KM92].

4.2 Timing-Forced Patterns

4.2.1 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 4.1 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 have to traverse the control and data-flow layers of their mlIR. To facilitate the traversal, we combine these layers in a single 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 of the mlIR, 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 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_c and E_d, respectively, in the mlIR: $E = E_{\text{dmy}} \cup E_c \cup E_d$. An edge in the CDFG is denoted $v_i \rightarrow v_k$, where v_i is an immediate predecessor of v_k.

The edges in the timing layer of the mlIR impose maximum or minimum constraints on the time between the operands at their ends. This represents an optimization problem on the CDFG, namely, how to schedule the operantion nodes along each timing edge. In the following sections we will first develop a formal definition of the problem and then give a heuristic to solve it with limited computational complexity.

4.2.2 ILP Formulation

We formalize the optimization problem in the form of an integer linear program (ILP) [NW99]. 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:

$$\min\{\ \max(|I_s| : 1 \leq s \leq m)\ \}. \tag{4.1}$$

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 4.1). $|V_{\text{op}}|$ is the number of operation nodes in G and $j_{\max}$ is a maximum length of the schedule in time steps. Note that the matrix considers only operation nodes. The dummy nodes, in contrast, are not to be scheduled as they do not represent any operations.

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

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

Figure 4.1: *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. Component-wise multiplication of the vector $(x_{1,j}, \ldots, x_{|V_{\text{op}}|,j})$ for one time step j in X 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:

$$|p(j)| = p_1 x_{1,j} + \ldots + p_{|V_{\text{op}}|} x_{|V_{\text{op}}|,j} = p \cdot x_j. \tag{4.2}$$

The path segment with the largest latency corresponds to the critical path in the slowest pattern. Therefore, the objective in Eq. (4.1) can be recast as a function of all path vectors of all timing edges, which are combined in the set P:

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

$$= \min\{\ \max(p \cdot x_j,\ \forall\, p \in P,\ j \in \{1, \ldots, j_{\max}\})\ \}. \tag{4.3}$$

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:

$$\sum_{j=1}^{j_{\max}} x_{i,j} = 1, \quad \forall\, i : v_i \in V_{\text{op}}. \tag{4.4}$$

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

$$g_i \leq g_k, \quad \forall\, (i,k) : v_i \rightarrow v_k. \tag{4.5}$$

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

$$g_l = \sum_{j=1}^{j_{\max}} j\ x_{l,j}. \tag{4.6}$$

Transforming the inequality to $g_i - g_k \leq 0$ and substituting with Eq. (4.6), we get the precedence constraint:

$$\sum_{j=1}^{j_{\max}} j\ x_{i,j} - \sum_{j=1}^{j_{\max}} j\ x_{k,j} \leq 0, \quad \forall\, (i,k) : v_i \rightarrow v_k,\ v_i, v_k \in V_{\text{op}}. \tag{4.7}$$

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:

$$\sum_{j=1}^{j_{\max}} j\ x_{i,j} - \sum_{j=1}^{j_{\max}} j\ x_{k,j} \leq -1, \quad \forall\, (i,k) : v_i \rightarrow v_{\text{exit}} \rightarrow v_{\text{entry}} \rightarrow v_k, \quad v_i, v_k \in V_{\text{op}},\ v_{\text{exit}}, v_{\text{entry}} \in V_{\text{dmy}}. \tag{4.8}$$

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 [AKPW83, AHM97] 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. (4.6) results in:

$$\sum_{j=1}^{j_{\max}} j\ x_{i,j} - \sum_{j=1}^{j_{\max}} j\ x_{k,j} \leq t_{\max}, \quad \forall\, (i,k) : v_i \xrightarrow{t_{\max}} v_k,\ v_i, v_k \in V_{\text{op}} \tag{4.9}$$

$$\sum_{j=1}^{j_{\max}} j\ x_{i,j} - \sum_{j=1}^{j_{\max}} j\ x_{k,j} \geq t_{\min}, \quad \forall\, (i,k) : v_i \xrightarrow{t_{\min}} v_k,\ v_i, v_k \in V_{\text{op}}. \tag{4.10}$$

This completes our formulation with the objective function (4.3) and the constraints (4.4) for assignment, (4.7) and (4.8) for precedence between operation nodes, and (4.9) and

(4.10) 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. However, solving an ILP quickly becomes intractable with increasing problem size. The search space for the ILP solver can be somewhat reduced by techniques such as constraint propagation. This, however, does not reduce the exponential worst-case complexity of the problem. To enable the handling of large application graphs we present a heuristic for the optimization problem in the following section. A heuristic represents a trade-off between optimality and time complexity.

4.2.3 Heuristic

Interactions between Timing Constraints

For scheduling an operation node in the CDFG, multiple timing constraints may have to be considered at once. Figure 4.2 gives an example why considering only one constraint at a time can lead to an incorrect schedule. The left-hand side of the figure shows a data-flow graph with two timing constraints. Considering only the constraint between nodes A and E and attempting to balance the size of the resulting patterns, we would cut the path from A to E in half as shown on the right-hand side of the figure. This partitioning, however, makes it impossible to meet the constraint between node B and D: The constraint requires nodes B, C, and D to be executed in the same cycle but the cut pushes node D to a different cycle than the others.

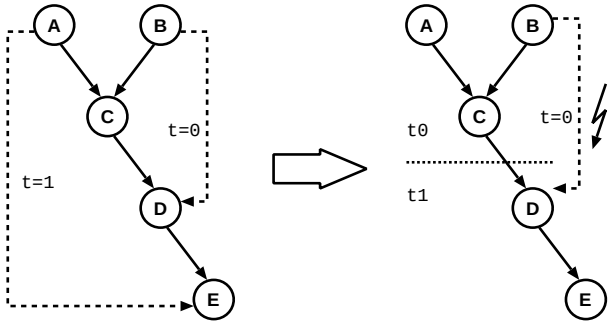


Figure 4.2: *Considering one constraint at a time is not sufficient.*

We solve this problem by annotating each operation node with the set of cycles in which it can be scheduled without violating any constraint. This set, D, is called a *domain*. We start by annotating nodes that are hooks to timing edges with the constraints or domains that follow from these edges. This is shown on the left-hand side of Figure 4.3. We then propagate the constraints to neighboring nodes until each node has its consistent domain, as shown on the right-hand side of the figure. Note that data dependencies in the graph represent precedence constraints as expressed by Eq. (4.5):

$$g_i \leq g_k, \quad \forall\,(i,k) : v_i \rightarrow v_k,\ v_i, v_k \in V_{\text{op}} \tag{4.11}$$

Control dependencies represent a precedence constraint according to Eq. (4.8):

$$g_i < g_k, \quad \forall\,(i,k) : v_i \rightarrow v_{\text{exit}} \rightarrow v_{\text{entry}} \rightarrow v_k,\ v_i, v_k \in V_{\text{op}},\ v_{\text{exit}}, v_{\text{entry}} \in V_{\text{dmy}} \tag{4.12}$$

As for the ILP in Section 4.2.2, a path that consists of more control nodes than the timing edge allows cycles represents an inconsistency in the constraint specification. Again,

transformations such as if-conversion [AKPW83, AHM97] must be used to decrease the number of control nodes and resolve the situation.

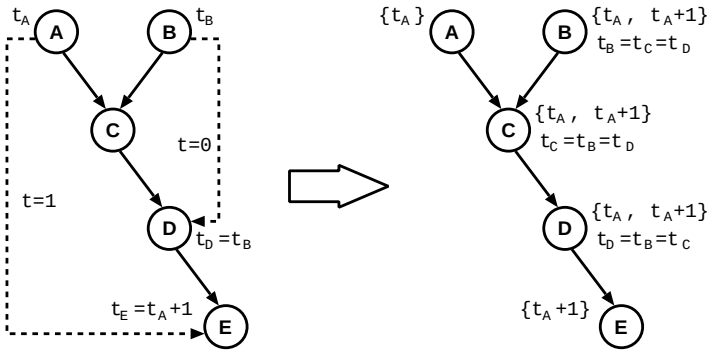


Figure 4.3: *Constraint propagation.*

Constraint propagation has been investigated for a long time in the field of constraint programming [Tsa93]. A fundamental and widely used class of propagation algorithms can be found in [Mac77], each with polynomial time complexity [MF85]. These algorithms provide an efficient means to automate the transformation in Figure 4.3. In the following we will use the AC-3 algorithm. It has a time complexity of $O(a^3n^2)$ for a number of nodes n and a maximum domain size a.

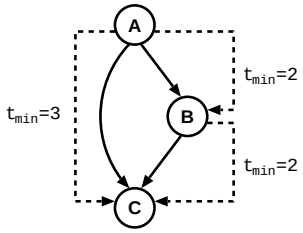


Figure 4.4: *Combined timing edges overruling* $\mathtt{t_{min}}$.

The common propagation algorithms require that the domains be finite. The natural lower bound for the domains in a scheduling problem is cycle 0. For the upper bound, t_∞, we must find a value that does not constrain the scheduling, i.e., a value that does not inhibit the longest conceivable optimal schedule. In a scenario as shown in Figure 4.4, minimum timing constraints can force two adjacent nodes to be scheduled further apart than the largest t_{min}. Therefore, to be on the safe side and not prevent an optimal schedule, we choose t_∞ such that it allows each pair of nodes in the graph to be as far apart as the sum of all minimum timing constraints,

$$t_\infty = n \cdot \sum \mathtt{t_{min}}.$$

The domains of all operation nodes before constraint propagation are then initialized to

$$D_i = \{0, 1, \ldots, t_\infty\} \quad \forall\, i : v_i \in V_{\text{op}}.$$

Scheduling Heuristic

After constraint propagation the schedule may still be underdetermined, i.e., there are still node domains with more than one cycle in which the node can be scheduled. In Figure 4.3

```
generateForcedPatterns( graph ) {
 propagateConstraints( graph.startNode );
 dfsVisit( graph.startNode, {} );
}

dfsVisist( node, path ) {
 path.append( node );
 node.visited = true;
 lastInPath = true;
 for each sucessor of node
  if successor.visited == false {
   lastInPath = false;
   dfsVisist( sucessor, path )
   path = {node};
  }
 if lastInPath == true
  balance( path );
}

balance( path ) {
 earliest = path.startNode.earliest();
 tMax = path.endNode.latest() - earliest;
 midTime = earliest + ceiling( tMax / 2 );
 fixNode = path.middle(); // if middle is between two nodes pick second one
 assignedCycle = fixNode.domain.closestMember( midTime );
 if fixNode.domain.size > 1 {
  fixNode.domain = {assignedCycle};
  propagateConstraints( fixNode );
 }
 if path.successor( startNode ) != fixNode
  balance( path.subPath( startNode, fixNode ) );
 if path.successor( fixNode ) != endNode
  balance( path.subPath( fixNode, path.endNode ) );
}
```

Figure 4.5: *Pseudo code: Generate timing-forced patterns.*

this is true for nodes B, C, and D which can be scheduled in cycle t_A or t_A+1. We therefore introduce a heuristic to choose a cycle from each node domain.

The basic idea of the heuristic is this:

- Traverse the graph in depth-first search (DFS) order;
- on the way, cut each taken path in half,
- schedule the middle node such that the maximum time allowed for the path is evenly distributed between both halves, and
- recurse over the halves until all nodes are scheduled.

Distributing the available time evenly across the graph works towards the optimization goal of having evenly sized patterns.

Figure 4.5 shows the algorithm, beginning with the top-level function `generateForcedPatterns`. After the initial constraint propagation, we start the graph traversal with a call to `dfsVisit`. This procedure implements the DFS [CLR90] with additional functionality to record the paths and to call the `balance` procedure on each complete path for scheduling. First, we append the discovered node to the current path and mark it as visited. The following loop examines if the successor nodes have been visited before. If an unvisited node is discovered we visit it by a recursive call to `dfsVisit`. Upon return from

the call all nodes in the path have been scheduled. To construct the next path of successors we need only the current node as a start node. Therefore, we reset the path to contain only the current node before continuing with the next successor. If, on the other hand, all of the successor nodes had been visited before then the current node is at the end of a path. In this case, we call the `balance` procedure to schedule all nodes on the complete path.

The `balance` procedure first computes the available time for the given path as the difference between the earliest possible time for the start node and the latest possible time for the end node, according to their domains. As we will schedule the middle node of the path we then compute `midTime`, the absolute time in the middle of the available time range, rounded up to the next integer number. This rounding up corresponds to picking the node just after the middle of the path as `path.middle()` does for `fixNode` if the middle is between two nodes. In this manner, the available time is evenly distributed between two halves of the path.

The cycle then assigned to `fixNode` is the member of its domain closest to `midTime`. Fixing a node to a cycle effectively means eliminating all but one element from the node domain. If the domain contained more than one element before then this may further constrain the legal cycles of other nodes. Hence, we have to propagate constraints again to restore consistency with the surrounding node domains. For efficiency, we pass `fixNode` to the propagation algorithm which enables it to start at the modified domain rather than traversing once again the entire graph. With consistency reestablished the `balance` procedure recurses on both halves of the path. The recursion terminates when there are no more nodes between the start and end node of the passed path.

Just as with the ILP, after scheduling, each set of nodes that have been scheduled in the same time step and are connected by data-dependencies in the CDFG represents a pattern in the pattern set S_t. The patterns generated in this way are inserted into the pattern library, employing the PSG structure from Chapter 2. As a result, the library contains all patterns necessary to implement the application graph.

Computational Complexity

The computational complexity of the heuristic is dominated by the constraint propagation. The AC-3 algorithm has been shown to have a complexity of $O(a^3n^2)$ with n the number of nodes and a the maximum size of their domains [MF85]. We call constraint propagation once at the beginning and then at most once in the `balance` procedure, which in turn is called at most once for each node in the graph except for the start node. This yields a worst-case complexity of $O(n \cdot (a^3n^2)) = O(a^3n^3)$.

The remaining `dfsVisist` procedure implements the DFS. For a bipolar graph with all `visited`-flags initialized to *false*, DFS is known to be $O(e)$, with e the number of edges in the graph, because it traverses each edge in the graph exactly once [CLR90]. Expressing e as a function of n we get $O(e) = O(n^2)$ which is dominated by the n^3 in the complexity we already have. The running time of the entire algorithm is therefore $O(a^3n^3)$. We have achieved a polynomial time complexity as opposed to the exponential complexity of an ILP solver.

4.3 Constraining Parallel Instruction Issues

4.3.1 Problem Statement

The methods to find timing-forced patterns described in Section 4.2 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 first develop a formal definition of the problem and then introduce a heuristic to limit the computational complexity of the process.

4.3.2 ILP Formulation

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

$$\min\{\ |S_p|\ \}. \tag{4.13}$$

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 $v_i \in V_{\text{pat}}$, $j \in \{1, \ldots, j_{\max}\}$, $k \in \{1, \ldots, k_{\max}\}$, and $y_{i,j,k} = 1$ iff v_i is scheduled in time step j and parallel-issue slot k (see Figure 4.6). $|V_{\text{pat}}|$ is the number of pattern nodes in G'. The dummy nodes are not represented in the array.

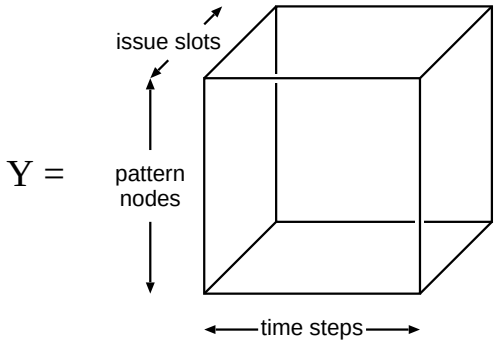


Figure 4.6: *Three-dimensional scheduling array.*

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 pattern be scheduled in exactly one time step and exactly one issue slot:

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} y_{i,j,k} = 1, \quad \forall\, i : v_i \in V_{\text{pat}}. \tag{4.14}$$

The precedence constraint requires that each node be scheduled later than its predecessors. Unlike the problem in Section 4.2.2, it is not possible to schedule dependent nodes in the same cycle. If g_l is the time step for v_l we get

$$g_i < g_h, \quad \forall\, (i,h) : v_i \rightarrow v_h. \tag{4.15}$$

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

$$g_l = \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{l,j,k}. \tag{4.16}$$

Tranforming Eq. (4.15) to $g_i - g_h \leq -1$ and substituting with Eq. (4.16) yields the precedence constraint:

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{h,j,k} \leq -1, \quad \forall\, (i,h) : v_i \rightarrow v_h,\; v_i, v_h \in V_{\text{op}}. \tag{4.17}$$

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

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{h,j,k} \leq -1, \quad \forall\, (i,h) : v_i \rightarrow v_{\text{exit}} \rightarrow v_{\text{entry}} \rightarrow v_h, \; v_{\text{exit}}, v_{\text{entry}} \in V_{\text{dmy}}. \tag{4.18}$$

Finally, the timing constraints are taken into account. We derive the constraint similarly to Eqs. (4.9) and (4.10), with Eq. (4.16) for the scheduled time steps:

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{h,j,k} \leq t_{\max}, \quad \forall\, (i,h) : v_i \xrightarrow{t_{\max}} v_h,\; v_i, v_k \in V_{\text{op}} \tag{4.19}$$

$$\sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{i,j,k} - \sum_{j=1}^{j_{\max}} \sum_{k=1}^{k_{\max}} j\; y_{h,j,k} \geq t_{\min}, \quad \forall\, (i,h) : v_i \xrightarrow{t_{\min}} v_h,\; v_i, v_k \in V_{\text{op}}. \tag{4.20}$$

This completes the formulation with objective function (4.13), and constraints (4.14) for assignment, (4.17) and (4.18) for precedence between operation nodes, and (4.19) and (4.20) for timing. The result of the optimization is the set of pattern bundles S_p. To overcome the intractability of ILP optimization for large problems we introduce a heuristic in the next section.

4.3.3 Heuristic

The basic idea of our heuristic is as follows:

- Compute partial schedule to analyze which pairs of patterns could be scheduled in parallel.
- For each pair, compute a value that captures the frequency of occurrences of the pair as well as the constraining effect it has on scheduling freedom.
- Sum up these values per control step as a metric for the parallelism demand in a control step.
- From the control steps with the largest parallelism demand, choose the pair with the largest value as a new compound instruction.
- Iterate until the constraint on parallelism is met: $k \leq k_{\text{max}}$, where k is the maximum number of patterns scheduled in parallel.

We again start by propagating the constraints across the graph to determine the domains of possible control steps for each pattern. The difference between the earliest and the latest possible control step in which a pattern can be scheduled, i.e., the difference between the smallest and the largest value in its domain is called the *mobility* [PG87] of a pattern. The mobility is a metric for scheduling freedom. We represent this in a *partial schedule* in which patterns that have no mobility are assigned to a particular control step while mobile patterns are assigned to their range of possible control steps. Figure 4.7 shows an example of a partial schedule. The table shows which patterns may be scheduled in parallel in the same cycle. Patterns that cannot be scheduled in parallel due to dependencies are placed in the same column. The tables of partial schedules can be constructed separately for each control node because only patterns in the same control node can be scheduled in parallel.

Step	Patterns			par_{step}
1		A	D	2
2	B			$1/3$
3	|	C	E	$1/3+1$
4	$\veebar$	|	F	$1/3+1$
5		$\veebar$	G	$1/3+1$
6		H	I	2

Figure 4.7: *Partial schedule with parallelism values.*

In order to measure the benefit we would gain from bundling two patterns in a compound instruction we define a *parallel value*, inspired by a method for regularity extraction that was sketched in [BKKS02]. The parallel value v_{par} of a *pattern* we define as the inverse of the number of control steps m in which it can be scheduled, according to its mobility. This is similar to the probabilities used in FDS [PK89]. We define the parallel value of a *pattern pair* to be the product of the values of its patterns:

$$v_{\text{par}}(\text{pair}_{12}) = v_{\text{par}}(\text{pattern}_1) \cdot v_{\text{par}}(\text{pattern}_2) \quad : \quad v_{\text{par}}(\text{pattern}) = \frac{1}{m}$$

This definition of the parallel value corresponds to the probability of the patterns in a pair being scheduled in the same control step, assuming equal scheduling probability for each step in the mobility range of a pattern. The value is 1 if both patterns in the pair have no

mobility. Otherwise, the value is a fraction of 1. This mechanism assigns a lower value to pairs that constrain the scheduling freedom more: Pairing binds one pattern to another, hence constraining the mobility of the pair to the intersection of the patterns' mobility.

For each control step in the partial schedule we generate all possible pairs of patterns. For each pair we insert a *pairing edge* between the according pattern entries in the PSG library, annotated with the *parallel value* of the pair. Each time a pair is generated for which the according pairing edge already exists we add the parallel value of the occurrence to the value counter in the library. If a pair occurs more than once in the same control step then we count only the highest parallel value.

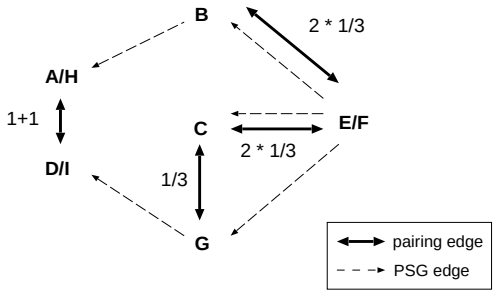


Figure 4.8: *Pairing edges with parallel values in a PSG.*

Figure 4.8 shows a fragment of a PSG with pairing edges and annotated parallel values for the schedule in Figure 4.7, assuming that patterns A/H, D/I, and E/F are pairwise identical. Pattern C, for instance, can be scheduled in three time steps. Hence, it has a parallel value of $v_{\text{par}} = 1/3$. As it shares time steps 3 and 4 with the identical patters E and F, the parallel value for the pair is $v_{\text{par}}(C \leftrightarrow E/F) = 2 \cdot 1/3$.

The parallel values provide a ranking by the benefit that the implementation of the pairs as an instruction would provide. We use this ranking to iteratively choose from the pairs in control steps with the highest parallelism. We define the parallelism par_{step} per control step as the sum over the maximum parallel value of all i operations in each column c:

$$par_{\text{step}} = \sum_{c} \max_i \{\, v_{\text{par}}(\text{pattern}_{c,i}) \,\}.$$

This again takes into account the probability for a pattern to be scheduled in the particular control step, given its mobility. Patterns B and C in Figure 4.7 each have a parallel value of $v_{\text{par}} = 1/3$. All other patterns have no mobility and have a parallel value of one. Adding up the parallel values, for instance, in step 4 is $1/3$ for either B or C, as they are in the same column, plus one for node F and yields $1/3 + 1 = 4/3$.

From the control steps with the highest parallelism we choose the pattern pair with the highest parallel value. In this way we give precedence to pairs that occur often which also tend to be simpler pairs, composed of fewer operation nodes. We replace all occurrences of the pair in the applications with the new bundle. If the patterns in the pair do not not occur anywhere else in the applications their entries in the sequential pattern library are removed. Then we start another iteration of the process by computing the new partial schedule. We iterate until the given constraint on parallel instruction-issues is not violated anymore.

Figure 4.9 shows the pseudo code for the entire procedure. The first `for`-loop builds the schedule table, and the second loop computes the parallel value for each pattern pair and the parallelism for each cycle. The third loop finds the pattern pair to be chosen for

```
while k > k_max {
 propagateConstraints();
 for each cycle in each pattern.domain {
  scheduleTable.insert(pattern, cycle, cntrlNode);
  // considers pattern dependencies by column assignment
 }
 for each cycle in scheduleTable {
  cycle.parallelism = sum of all column.maxPatternValue() in cycle;
  for each pair of patterns in different columns in cycle
   psg.pairingEdge( pair ).value += pair.value;
 }
 for each pair in each cycle with cycle.parallelism = maxParallism
  if psg.pairingEdge( pair ).value > maxValue {
   candidate = pair;
   maxValue = psg.pairingEdge( pair ).value;
  }
 psg.insert( candidate );
 for each occurrence of candidate in graph {
  replace occurrence by candidate;
 }
 for each pattern in candiate
  if no more occurrence of pattern in graph
   psg.remove( pattern );
}
```

Figure 4.9: *Pseudo code: Constrain parallel instruction issues.*

implementation which is then inserted into the PSG and the application graph. If a pattern in the pair does not occur in the graph individually any longer it is removed from the library.

The worst-case computational complexity of the constraint propagation is $O(a^3n^2)$. For table construction, each domain member of each node is visited exactly once and each dependency in the graph is analyzed. With e dependencies, this yields a complexity of $O(an+e) = O(an+n^2)$ for table construction. The next two `for`-loops analyze all pairs of patterns which cannot have higher complexity than visiting all possible pairs of patterns in each cycle which is $O(an^2)$. The entire procedure iterates at most n times. This results in a total worst-case complexity of $O(\ n \cdot (a^3n^2 + an + n^2 + an^2)\ ) = O(a^3n^3)$. Again we have achieved a polynomial time complexity as opposed to the exponential complexity of an ILP solver.

4.3.4 Using IOGs to Eliminate Instructions

In Chapter 2.2 we described the IOG method based on ID operands to substitute patterns by others in order to simplify an instruction set. This method can also be applied to 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 chosen pair in S_p is removed from the instruction set and its occurrences in the graph are covered by the dominating pair.

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.

4.4 Handling Control Constructs

Some control constructs in the mlIR need to be converted in a preprocessing step before they can be handled by the methods described earlier in this chapter. In the following sections we show how we cope with loops and wait nodes, as well as branch, nop, and WFC operations.

4.4.1 Loop Ripping

The constraints we formulated in the previous sections, e.g. Eq. (4.8), require that a control node be scheduled later than its predecessors. In a loop this is impossible to achieve because here each node is its own predecessor and successor at the same time. Therefore, we must cut open each loop, transforming the application graph into a *directed acyclic graph (DAG)*, while maintaining the timing and data dependencies between nodes, inside and outside the loop.

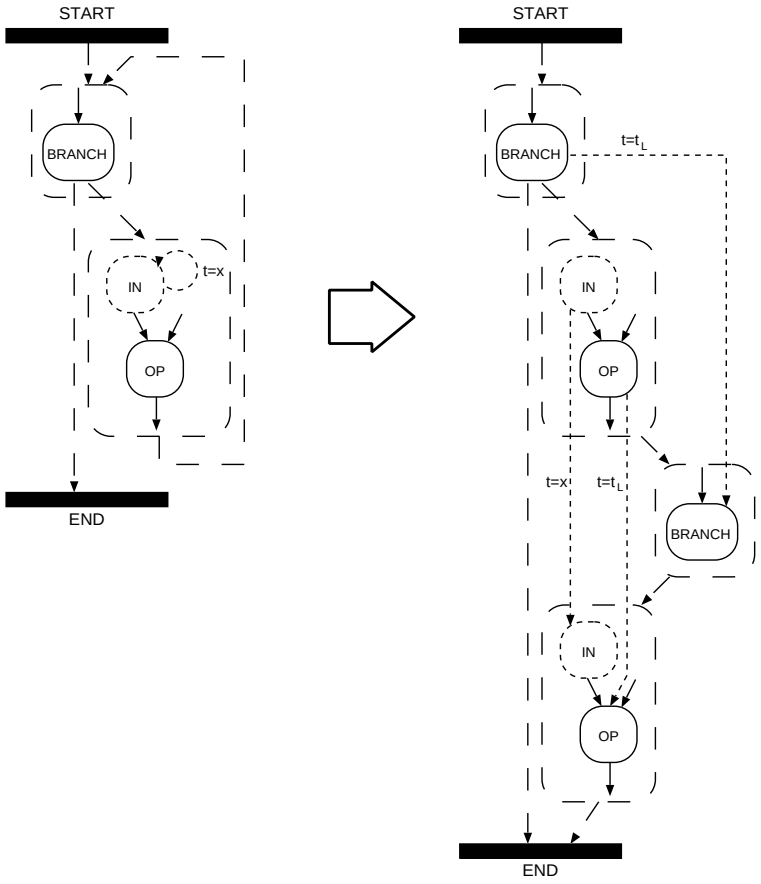


Figure 4.10: *Loop ripping: concatenating two iterations.*

We solve this problem by concatenating two iterations of the loop with all their dependencies between each other. Instead of closing the loop again, we assign the end node as the successor of the last node in the loop. In this manner, the loop has been *ripped* open while any dependencies between loop iterations are represented by dependencies between

the two concatenated iterations. To maintain the same timing between nodes across iterations we pose the additional constraint that all nodes must have the same time distance to their clones in the next iteration.

Figure 4.10 gives an example. On the left-hand side, there is a graph with a loop in which an I/O node has a timing constraint on itself. This constraint represents a rate constraint on the loop. On the right-hand side, the loop has been ripped open, two iterations are concatenated, and the rate constraint now spans from one iteration to the next. Furthermore, there is a constraint between each node and its clone with an identical loop constraint t_L to ensure equal spacing between nodes across iterations.

This approach also enables us to maintain dependencies between nodes in the first loop iteration and predecessors outside the loop. Moreover, to also cater for constraints between nodes in the last iteration and successors outside the loop, we can copy an iteration *before* the original branch. The first node of this copy would have the start node as its predecessor. The constraints and the spacing between iterations are maintained in the same manner as before.

The result of this *loop ripping* conversion is a DAG for which the precedence constraints between control nodes can be met. We have arrived at a structure on which our methods for instruction-set generation can work. Note that in our methodology the cloned iterations are only necessary for constraint propagation. The other algorithms in the pattern construction process ignore the clones.

4.4.2 Wait-Node Scheduling

In Section 3.3.4 we have suggested hardware and software implementations of wait nodes. Following the considerations presented there we implement wait nodes by a dedicated counter register in order to control the complexity of scheduling. Consequently, a wait node is composed of two instructions: one to write the start value to the counter register—a simple move operation that we call *counter start*—and one to block the control flow until the counter reaches zero—the WFC instruction. The WFC instruction stops the fetching of instructions and the incrementation of the program counter (PC) until the wait counter triggers. Therefore, the next instruction to be executed after the wait is the one at $PC_{\text{wait}}+1$.

The counter start can be scheduled in the same cycle as the WFC or earlier. When shifting the counter start to earlier cycles the waiting time must be increased in order to maintain the original trigger time. Hence, we adjust the input value to the wait node accordingly by adding the number of cycles by which the counter start has been shifted. In the mlIR, we represent this shifting to an earlier slot by subtracting cycles from each incoming timing edge and adding them to each outgoing timing edge. In this way the wait node represents setting the start value of the counter register.

The WFC either remains in the cycle before the wait target or it is also shifted to an earlier slot, filling the resulting gap with other instructions. If the WFC is shifted the start value must be adjusted accordingly. Moving the WFC closer to the counter start we have to subtract the number of cycles by which the WFC has been shifted. Note that the total offset incurred by this scheduling and the `min_val` provided by the wait `#pragma` must not be negative in order to prevent negative wait times:

$$\text{offset} = \text{min\_val} + \text{offset}_{\text{schedule}} \geq 0$$

All these adjustments must be performed consistently on all incoming or outgoing edges of a wait node.

For pattern construction we require each wait node to be preceded by an adder to allow for scheduling adjustments. If there is no addition or subtraction before a wait node in an application then an addition node is inserted before pattern construction commences. The adjustment adder assumes the task to move the start value into the counter register, making a dedicated move operation obsolete.

The incoming and outgoing timing edges of each wait `#pragma` are joined for pattern construction to form a single timing edge between the source of the incoming and the destination of the outgoing edge. The timing values of the two edges are summed up. The dynamic run-time delay that the wait node represents is considered to be zero for pattern generation because the timing constraints must also be met if at run time the input to the wait node turns out to be zero. After the construction of the timing-forced patterns, the values of the original timing edges are adjusted according to the resulting distribution of cycles before and after the wait.

4.4.3 Branches, Nop, WFC

Conditional branches can form a pattern with the arithmetic operations that compute the branching condition. Their scheduling, however, is constrained by the fact that they transfer the control to another basic block and therefore must always be scheduled in the last cycle of their basic block. We integrate this requirement into our constraint framework by means of timing edges from the other DFGs in the same basic block to the conditional branch with $t \geq 0$.

In contrast, unconditional branches have no dynamic data input and can therefore not be part of a DFG pattern. We implement these branches by a goto-offset that is available in every cycle in parallel to other instructions. The offset requires only few bits as control-dominated applications are characterized by small basic blocks and skipping these basic blocks requires only short jumps. On the other hand, small basic blocks result in frequent branches which means that the goto-offsets are used frequently. These considerations justify having dedicated bits in the ASIP's instruction format.

Similarly, the WFC operation can be activated by a single bit in an instruction word rather than occupying an entire instruction slot. Furthermore, we assume *nop* to be part of any instruction set. This approach coincides with the fact that nop, goto, and WFC operations are not represented by operation nodes in a data-flow graph because they are only introduced by the scheduler. The approach enables the instruction generator to concentrate on the operations that are connected to the data-flow layer, thereby simplifying the employed scheduling methods.

4.5 Summary of Instruction-Set Generation

In this chapter we have completed our methodology to generate an instruction set for control-dominated applications. In a first step, we devised an algorithm to derive timing-forced patterns from fine-grained timing constraints, specified by the ASIP designer in an HLL as suggested in Section 3.2. The resulting patterns guarantee that the timing constrains of the applications can be met with the final instruction set.

In a second step, we suggested a method to constrain parallel instruction issues to a number requested by the ASIP designer. The algorithm bundles patterns that occur in parallel, taking into account their mobility and the overall contribution of a pair to reducing the

number of parallel issues. We furthermore elaborated on how to handle control constructs such as wait nodes and loops in our pattern construction flow.

The final instruction set consists of timing-forced instructions, pattern pairs, and the individual operations needed for a complete covering of the application graphs. The timing-forced patterns guarantee that the given applications can be implemented with this instruction set in a way that meets the timing constraints which the designer specified. Thanks to our bundling technique the instruction set complies with the required maximum number of instructions issued in parallel. The use of IOGs in the process exploits synergies between patterns, leading to a leaner instruction set.

5 Experimental Results

In this chapter, we first examine the performance of PSG and PSG/IOG libraries compared with traditional linked-list implementations in Section 5.1. In Section 5.2 we demonstrate the feasibility of our methodology by generating an instruction set in a network-processing application domain with our algorithms. We assess the quality of the result by comparing it with a manually designed instruction set for the same domain. In Section 5.3 we summarize our findings.

5.1 Pattern-Library Performance: Speed and Size

In this section we present the performance measurements we have conducted on our C++ implementation of the PSG and IOG data structures and algorithms.

5.1.1 Workload

We have implemented a pattern generator within the Machine-SUIF compiler framework [MS]. To grow the scope for the pattern generator beyond basic-block boundaries, the generator works on a static single assignment (SSA) representation which extends the DFGs to operand definitions in other basic blocks. We include operations in other DFGs in a pattern if these operations are reached directly, not through a ϕ-function, because these operations can easily be moved across control-flow boundaries.

We feed the patterns to one of four types of pattern libraries:

- A PSG for DAG patterns, constructing only the search path to each pattern.
- A combined PSG/IOG for tree patterns, constructing the entire IOG for each pattern.
- Unordered linked-list libraries for trees and DAGs, respectively, for comparison with the state of the art.

We tested the libraries on a subset of the MediaBench benchmark suite [LPMS97]. We measured the search times while the library was being constructed, i.e., from the first inserted pattern to the last. Furthermore, we measured the number of entries in each library after the last pattern had been inserted. The pattern generator passed between 60 and 41233 patterns per run to the library.

5.1.2 Performance in PSGs

Figure 5.1 compares the search times for DAG patterns, giving the speed-up of PSG libraries over linked lists. The size of both library types is basically identical because we

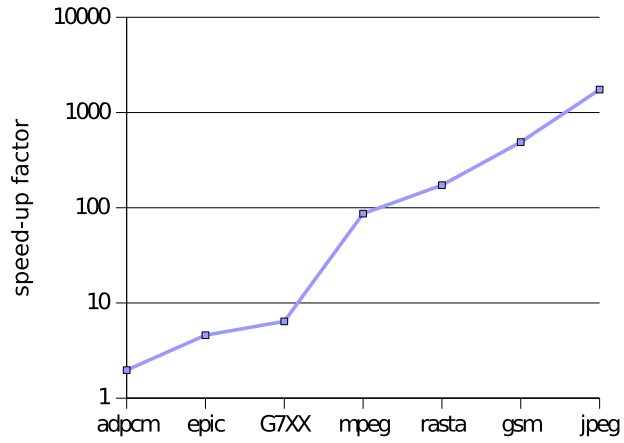


Figure 5.1: *Search speed-up for DAG patterns in a PSG.*

only supplement patterns on the search path to each pattern rather than constructing the entire IOG for each inserted pattern. The benchmarks are ordered by the number of different patterns that the pattern generator passed to the library.

The more patterns in the library the longer the linked list grows and the longer the worst-case search time in the list. In contrast, the search time on a PSG is independent of the library size. Hence, the larger the library the more significant becomes the advantage in search speed of the PSG. In the JPEG library with 41233 DAG patterns we measured a PSG speed-up factor compared with the linked list of 1743.

5.1.3 Performance in Combined PSG/IOGs

In a second set of experiments, we constructed the entire IOG for each inserted pattern. As we showed in Section 2.3.4, a PSG/IOG library can only be constructed for tree-shaped patterns. Because of the increase in size due to constructing an IOG for each pattern a PSG/IOG organization is viable only for medium-sized libraries. The largest library in our experiments was the rasta benchmark with 11371 patterns in the PSG/IOG. The upper graph in Figure 5.2 shows the resulting speed-up for searches for tree patterns, comparing a PSG/IOG with a linked list. The lower graph shows the increase in size of the PSG/IOG over the linked list for the same workload.

Our results show that the overhead due to constructing the IOG for each entered pattern grows the library to up to nine-fold size compared with a linked list which holds only the entered patterns—which is the minimal set. However, the hierarchical organization of the PSG/IOG still reduces the search times as dramatically as the pure PSG—in spite of the larger size. Moreover, the size overhead of constructing an IOG for real workload is far below the theoretical worst case indicated in Chapter 2. Note that only the IOG enables the use of complex patterns to substitute simpler ones. The linked list does not provide this advantage.

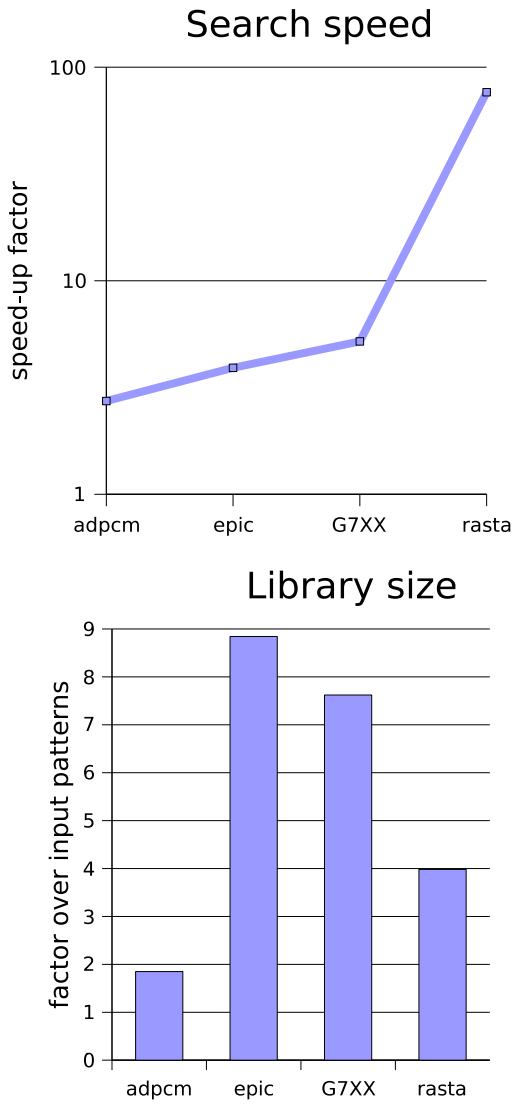


Figure 5.2: *Search speed-up and library size for tree patterns in a combined PSG/IOG.*

5.2 Example of a Control-Dominated ASIP Design

In order to prove the concept of our design methodology in a real-life case, we apply our methodology to a representative control-dominated ASIP: a parser for packet headers as a building block for a network processor (NP). We compare the result with a manually designed header parser to assess the quality of our automatically generated instruction set.

In the following, we first describe the manually designed header parser. Then we illustrate each step taken in our methodology to arrive at an instruction set for the same application domain. Finally, we compare the two results.

5.2.1 A Parser for Protocol Headers in Network Packets

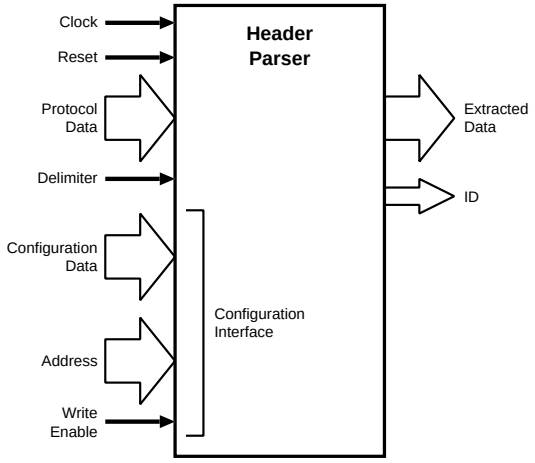


Figure 5.3: *Header-parser interfaces.*

The manual design and optimization of a header parser for network processing has been described in [Dit00]. Figure 5.3 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 defined in Section 1.3.3, 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 network protocols considered are versions 4 and 6 of the Internet protocol (IPv4, IPv6). The relevant header fields to be extracted are given in Tables 5.1 and 5.2, 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 resulting instruction set is given in Table 5.3, roughly ordered into four data-only and five control-related instructions. The patterns that the compound data-only instructions implement are shown in Figure 5.4. Programmable operands which are encoded in the op-code of the instruction are labeled in bold italics. The internal architecture of the parser is depicted in Figure 1.4 in Chapter 1. The parser can issue two instructions in parallel in a VLIW fashion. Moreover, an offset can be added to the program counter for an unconditional branch and the wait counter can be tested in any cycle.

Cycle #	Fields relevant internally	Fields relevant externally
1	IP Header Length (IHL) [4–7]	Type of Service (ToS) [8–15]
2	–	–
3	–	Protocol [8–15]
4	–	Source Address [0–31]
5	–	Destination Address [0–31]
wait (IHL-5) max. 10 for layer-4 header	–	TCP / UDP: Source Port [0–15], Destination Port [16–31]

Table 5.1: *Relevant header fields in IPv4.*

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]

Table 5.2: *Relevant header fields in IPv6.*

Instruction	Effect
`Send`	Extract field from input, write it to output together with ID.
`Send_Reg`	Write register to output together with ID.
`Write_Reg`	Extract field from input, write it to register.
`IP6_Counter`	Extract field from input, compute IPv6 header length, write it to wait counter.
`Nop`	No operation, i.e., wait one cycle.
`Goto`	Unconditional branch.
`If_Counter`	Conditional branch depending on wait-counter value. (∼WFC)
`Init_Case`	Multi-way branch.
`Ld_Const`	Move conditions into `Init_Case` configuration registers.

Table 5.3: *Manually designed header-parser instruction-set.*

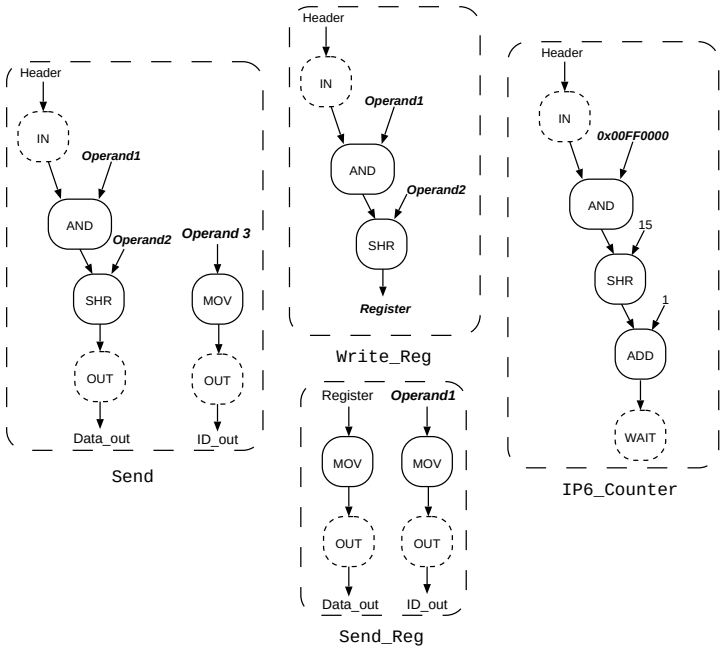


Figure 5.4: *Patterns of manually designed data instructions.*

The parser has been synthesized for a 0.18-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.

In the following we demonstrate how to employ our methodology to derive an instruction set for the header parser. We will then compare the result with the design described above.

5.2.2 Specification of Benchmark Applications

The first step in our methodology is to specify a set of benchmark applications that are representative for the target domain. Integrated in the application code are the timing annotations as introduced in Section 3.2.

Deriving from Table 5.1 an algorithm to parse an IPv4 header, we arrive at the C code with timing annotations given in Figure 5.5. For readability, the number in the label names corresponds to the number of cycles from the start. The timing constraints are defined such that the input register `in` is read in the cycle in which the expected header word appears. The extracted data are written to the output register `out` in the same cycle in order to have the output register available in the following cycle for the next header field. After parsing the relevant fields in the standard IP header the wait statement finds the beginning of the transport header and extracts the port numbers.

Figure 5.6 shows the timed code for IPv6 header parsing, derived in the same manner from Table 5.2. After parsing the IP header the switch statement handles the subsequent extension header or layer-4 header. If an extension header has a *HdrExtLen* field its header length is computed and written to the `counter` variable. This variable is used as the input to a wait node in order to find the beginning of the next header—another extension header

```
#pragma io main::in
#pragma io main::out
#pragma io main::id
#pragma time START l_1 = 0
#pragma time l_1 l_1a = 0
#pragma time l_1a l_1b = 0
#pragma time l_1b l_1c = 0
#pragma time START l_3a = 3
#pragma time l_3a l_3b = 0
#pragma time l_3b l_3c = 0
#pragma time START l_4a = 4
#pragma time l_4a l_4b = 0
#pragma time l_4b l_4c = 0
#pragma time START l_5a = 5
#pragma time l_5a l_5b = 0
#pragma time l_5b l_5c = 0
#pragma wait l_5a l_6a counter 5
#pragma time l_6a l_6b = 0
#pragma time l_6b l_6c = 0

int main(int argc, char argv[]) {
 int in, out, id, counter;
 int ToS, Proto, SrcAddr, DestAddr;
 int L4Ports;
l_1:
 counter = ( (in & 0x0f000000) >> 24 );
l_1a:
 ToS = (in & 0x00ff0000) >> 16;
l_1b:
 out = ToS;
l_1c:
 id = 1;
l_3a:
 Proto = (in & 0x00ff0000) >> 24;
l_3b:
 out = Proto;
l_3c:
 id = 2;
l_4a:
 SrcAddr = in;
l_4b:
 out = SrcAddr;
l_4c:
 id = 3;
l_5a:
 DestAddr = in;
l_5b:
 out = DestAddr;
l_5c:
 id = 4;
/* wait on counter */
l_6a:
 L4Ports = in;
l_6b:
 out = L4Ports;
l_6c:
 id = 5;
}
```

Figure 5.5: *Timed C code for IPv4 parsing.*

or a layer-4 header. The parsing terminates if either a TCP or UDP header or an unknown header is encountered.

A compiler front-end transforms the annotated C code into an mlIR graph as defined in Section 3.3. The timing annotations are parsed employing the productions in Section 3.4. Figure 5.7 shows the multi-layer IR representation for the IPv4 code. Figure 5.8 shows the graph for IPv6. The I/O variables and wait nodes are marked by dashed circles. For each wait node, the minimum input value given by the wait `#pragma` has been used to derive the timing edge to the wait node and to adjust the offset at the wait node input according to Section 4.4.2. Time is one less than the available minimum value because the outgoing timing edge consumes one cycle. To unclutter the representation some timing edges have already been replaced by the appropriate fixed cycle number next to the corresponding I/O variable.

5.2.3 Timing-Forced Instructions

Before pattern construction the loop in the IPv6 graph must be unrolled with the loop ripping method in Section 4.4.1. Figure 5.9 shows the CDFG and timing layer with two concatenated iterations and the additional timing edges to guarantee equal schedules in each iteration. The cloned iteration is shaded to mark that it is needed for constraint propagation only but not for pattern construction. The wait node has been replaced by a direct timing edge as required by Section 4.4.2. Timing edges that started and ended at the same node now span across iterations.

The case operation in Figure 5.8 has a timing problem: The preceding control node has operations scheduled in clock cycle 10 and the following control nodes have operations scheduled in cycle 11. As control nodes must not overlap in any clock cycle there is no schedule slot left for the case operation. This problem is reported to the designer. The

```
#pragma io main::in
#pragma io main::out
#pragma io main::id
#pragma time START l_1a = 1
#pragma time l_1a l_1b = 0
#pragma time l_1b l_1c = 0
#pragma time START l_2 = 2
#pragma time START l_3a = 3
#pragma time l_3a l_3b = 0
#pragma time l_3b l_3c = 0
/* ...continue for l_4* to l_9* */
#pragma time START l_10a = 10
#pragma time l_10a l_10b = 0
#pragma time l_10b l_10c = 0
#pragma time START l_11a1 = 11
#pragma time l_11a1 l_11b1 = 0
#pragma time l_11b1 l_11c1 = 0
#pragma time START l_12a1 = 12
#pragma time l_12a1 l_12b1 = 0
#pragma time START l_11a2 = 11
#pragma time l_11a2 l_11a2 = 2
#pragma time START l_11a3 = 11
#pragma time l_11a3 l_11b3 = 0
#pragma time START l_11a4 = 11
#pragma time l_11a4 l_11b4 = 0
#pragma wait l_11a3 l_11a3 counter 2

int main(int argc, char argv[]) {
 int in, out, id, counter;
 int Flow, NextHdr;
 int SrcAddr, DestAddr, L4Ports;
l_1a:
 Flow = in & 0x0fffffff;
l_1b:
 out = Flow;
l_1c:
 id = 1;
l_2:
 NextHdr = (in & 0x0000ff00) >> 8;
l_3a:
 SrcAddr = in;
l_3b:
 out = SrcAddr;
l_3c:
 id = 2;
/* ...continue for 2nd SrcAddr *
 *    to 3rd DestAddr word     */
l_10a:
 DestAddr = in;
l_10b:
 out = DestAddr;
l_10c:
 id = 9;
next_header:
 switch ( NextHdr ) {
 case 6: case 17: /* TCP or UDP */
l_11a1:
  L4Ports = in;
l_11b1:
  out = L4Ports;
l_11c1:
  id = 10;
l_12a1:
  out = NextHdr;
l_12b1:
  id = 11;
  break;
 case 44: /*  Fragment Header */
l_11a2:
  NextHdr = (in & 0xff000000) >> 24;
  goto next_header;
 case 0: case 43: case 51: case 60:
 /* all other extension headers */
l_11a3:
  NextHdr = (in & 0xff000000) >> 24;
l_11b3:
  counter =
     ( (in & 0x00ff0000) >> 15 ) + 2;
  /* wait for counter */
  goto next_header;
 default: /* unknown header */
l_11a4:
  out = NextHdr;
l_11b4:
  id = 11;
 } /* switch */
}
```

Figure 5.6: *Timed C code for IPv6 parsing.*

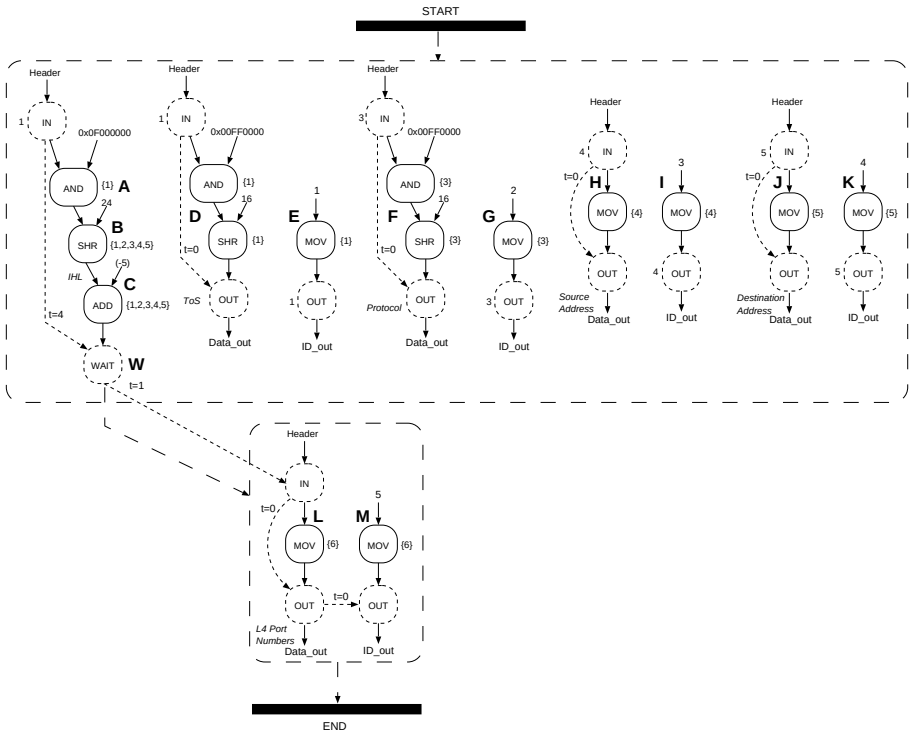


Figure 5.7: *Multi-layer IR graph for IPv4 header parsing.*

solution shown in Figure 5.9 is to merge the first instance of the case control-node with the first control node. In Section 4.4.3 we found that a conditional branch must always be the last operation in a basic block. Therefore, the case operation has implicit timing edges to the other DFGs in its basic block with $t \geq 0$.

According to our heuristic in Section 4.2.3, constraint propagation determines the scheduling freedom for each node. The resulting domains with all possible clock cycles are annotated in curly brackets with the nodes in Figures 5.7 and 5.9. The `dfsVisit` procedure traverses the graphs to schedule any nodes with undetermined clock cycle, i.e., nodes with more than one value in their domain. In Figure 5.7 this is the case for the nodes marked with a bold B and C. In the process, the `balance` procedure is called for the path $A \rightarrow B \rightarrow C \rightarrow L$ with a `midTime` of 4. The middle of this path is node C which has a 4 in its domain and is therefore scheduled in cycle 4. Then `balance` recurses for $A \rightarrow B \rightarrow C$ with a `midTime` of 3 which is therefore assigned to node B. Similarly, in Figure 5.9, node D is scheduled in cycle 6.

In the final schedule, each set of nodes that have been scheduled in the same time step and are connected by data dependencies in the CDFG represents a pattern. The timing-forced sequential patterns in the IPv4 case are the DFGs marked D and F, and in the IPv6 case, P, Q, and R. The result for IPv4 is intuitive as D and F are locked in a timing constraint with $t = 0$. A, B, and C, on the other hand, are three nodes with a timing constraint of $t = 4$, granting each node a private clock cycle. For IPv6, the patterns are forced in the loop which has a timing constraint of $t = 2$ between iterations. As the loop comprises two control nodes, each has only one cycle available, forcing all DFGs in the loop as sequential patterns. Nodes C, D, and I, on the other hand, together with an operation from the following control node, are four nodes with a timing constraint of $t = 9$, granting each an individual cycle.

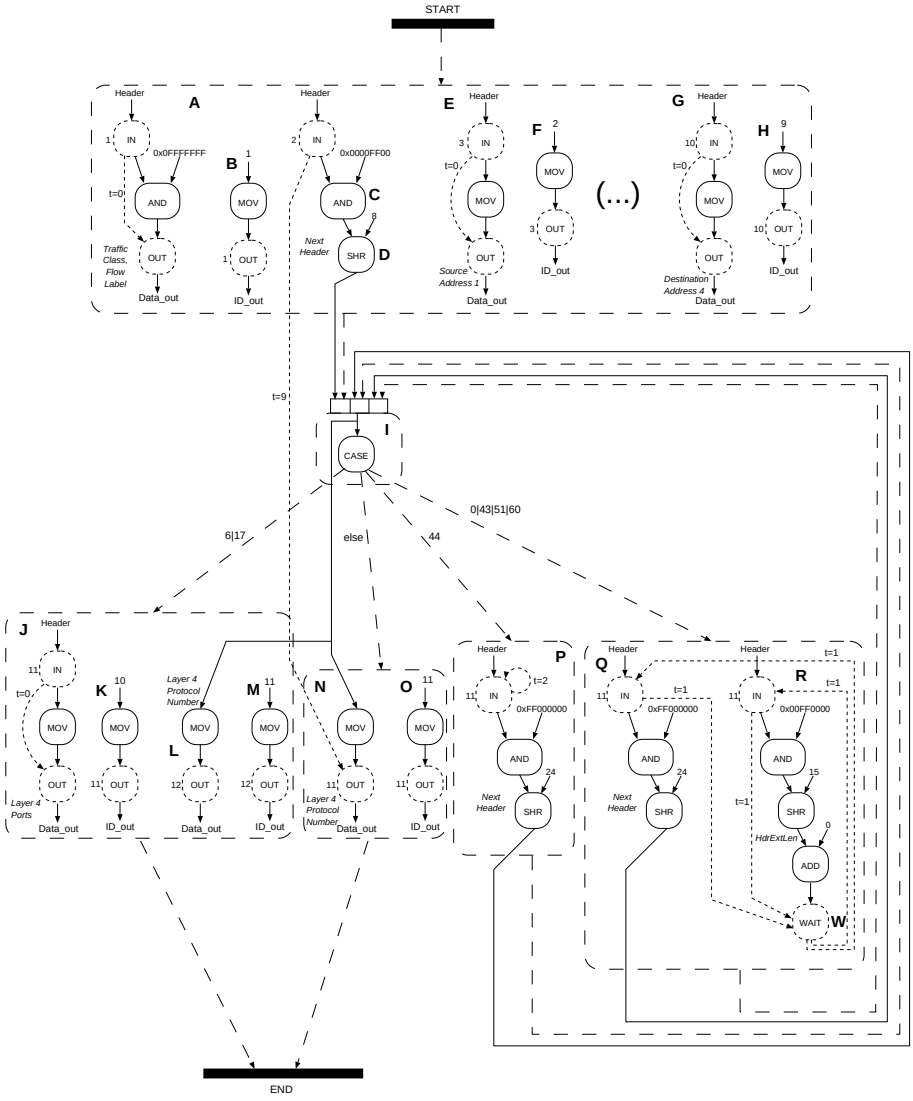


Figure 5.8: *Multi-layer IR graph for IPv6 header parsing.*

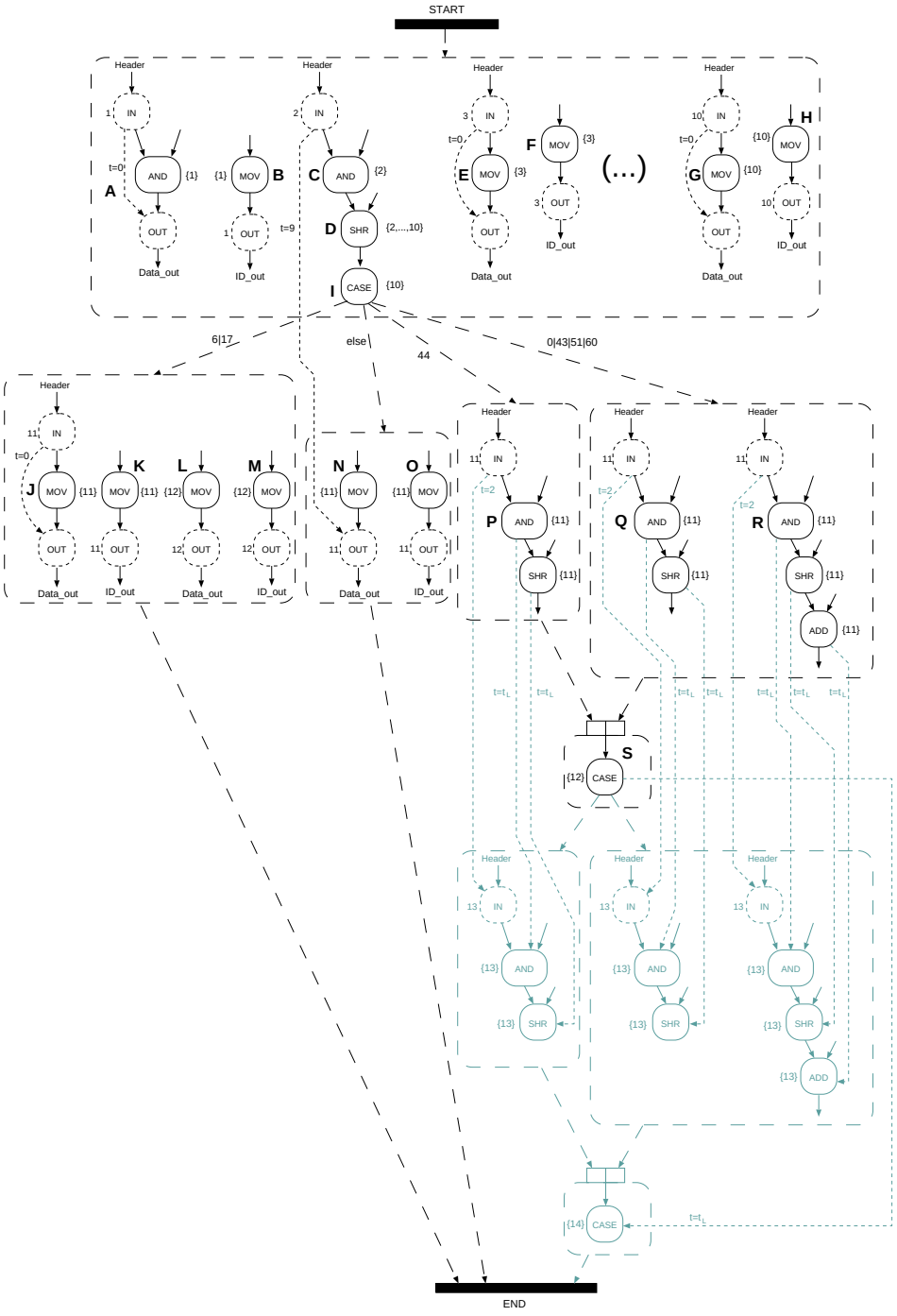


Figure 5.9: *IPv6 graph after loop ripping.*

5.2.4 Constraining Parallelism

The manually designed header parser can issue two instructions in parallel. In order to get comparable results we also constrain the number of parallel instruction issues to two as an input parameter for our second heuristic, described in Section 4.3.3. The first step in the heuristic is to derive the partial schedule for both application graphs, using the forced patterns from the first heuristic as instructions. Table 5.4 shows both schedules, including the parallelism par_{step} for each control step. The letters refer to the pattern labels in Figures 5.7 and 5.9, respectively. Horizontal lines mark the borders of basic blocks. The arrows indicate the mobility of an instruction.

(a) IPv4 Parsing.

Step	Patterns				par_{step}
1	A		D	E	3
2	B				1/3
3	|	C	F	G	7/3
4	$\veebar$	|	H	I	7/3
5		$\veebar$	J	K	7/3
6	L		M		2

(b) IPv6 Parsing.

Step	Patterns			par_{step}
1	A	B		2
2	C			1
3	D	E	F	15/7
4	|	E′	F′	15/7
...	...	...	...	...
8	|	E^V	F^V	15/7
9	$\veebar$	E^{VI}	F^{VI}	15/7
10	I	G	H	3
11	J	K		2
12	L	M		2
11	N	O		2
11	P			1
11	Q	R		2
12	S			1

Table 5.4: *Partial schedules.*

Combining the patterns in each control step and inserting the according pairing edges into the the pattern library results in the library PSG shown in Figure 5.10. The pairing edges are annotated with their parallel values. Based on the parallel values we choose a pair from the control steps with the highest parallelism: step 1 in Table 5.3(a) and step 10 in Table 5.3(b). The pairs that occur in these steps are $(and \parallel and \rightarrow shr)$, $(and \parallel mov)$, $(and \rightarrow shr \parallel mov)$, $(mov \parallel mov)$, and $(case \parallel mov)$ with the parallel values 1, 2, 2, 14, and 1, respectively. Hence, the first choice is $(mov \parallel mov)$ and all occurrences of two parallel moves are replaced by a single pair instruction. The process iterates two more times before the parallelism constraint is satisfied, choosing two more pairs: $(and \parallel mov)$ and $(and \rightarrow shr \parallel mov)$. This set of pairs covers all occurrences of a move operation. Therefore, the individual move is dropped from the pattern set.

In a final step, the IOG of the instruction set is constructed in order to remove instructions that are covered by others. The operations *and* and *mov* both appear in the IOG of $and \rightarrow shr$. Therefore, the pair $(and \rightarrow shr \parallel mov)$ dominates the other two chosen pairs which are consequently removed from the instruction set. All operations that have not been covered by pattern pairs are covered by single patterns. In the IPv4 graph these are the patterns B, C, and D, and in the IPv6 graph C, D, I, P, Q, R, and S. Of these patterns the ones that are not dominated by any other pattern in the IOG in Figure 5.10 are the *case* operation and $and \rightarrow shr \rightarrow add$. The final instruction set therefore consists of the derived patterns $and \rightarrow shr \rightarrow add$, $(and \rightarrow shr \parallel mov)$ and *case*, and the mandatory control

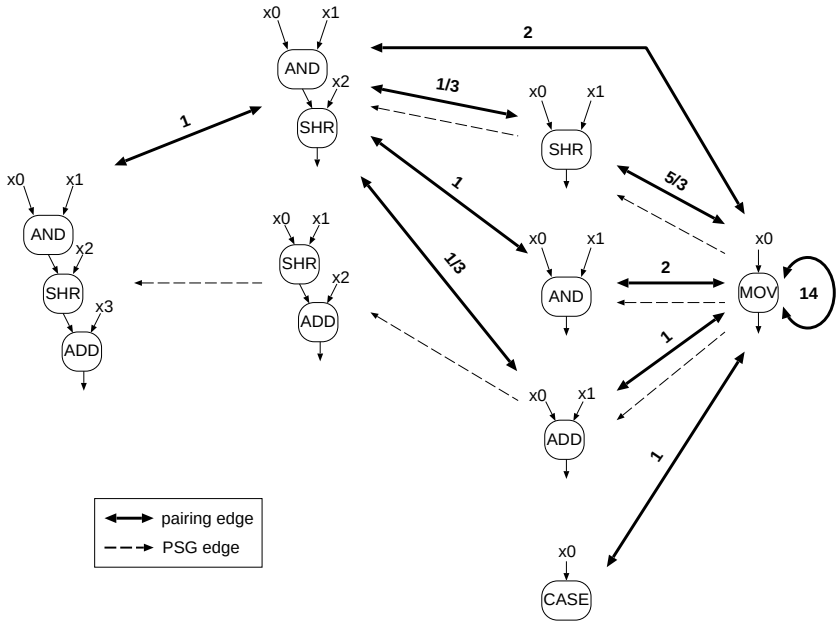


Figure 5.10: *Forced patterns and pairing edges.*

instructions *nop*, *goto* and *WFC*. Figure 5.12 shows the compound patterns that have been chosen.

5.2.5 Comparison with Manual Design

The mandatory control instructions *nop*, *goto*, and *WFC* as well as the *case* instruction are present in the manually designed instruction set as well as the one derived by our methodology. The difference between the two sets are in the compound data-only instructions shown in Figure 5.11 for the manual design[1] and in Figure 5.12 for our methodology.

The `Send` instruction in Figure 5.11 matches `Instruction2` in Figure 5.12 and `IP6_Counter` matches `Instruction1`. In both cases, our methodology suggests instructions with the same structure as the manually designed ones. The reason why the remaining instructions `Send_Reg` and `Write_Reg` are not part of our derived instruction set is that the IOG showed that they are dominated by the two other instructions, respectively—they can be implemented with the present instructions by means of ID operands. Applying a zero as operand 4 to `Instruction1` results in the `Write_Reg` instruction; applying zeros as operands 2 and 3 transforms `Instruction2` to `Send_Reg`. Therefore, the dominated patterns have been eliminated from the instruction set in the last steps of the previous section.

The result is an instruction set that is functionally equivalent to the manual design since all manually derived instructions are covered. However, the instruction set is leaner because it exploits the synergies between patterns provided by identity-operand transformations.

The manual design of the header-parser instruction-set is a complex and therefore time-consuming and error-prone task. In the case study we have shown that our methodology

[1] Since our methodology does not constrain the operand types, fixed operands have been replaced by generic operands in the figure for comparison.

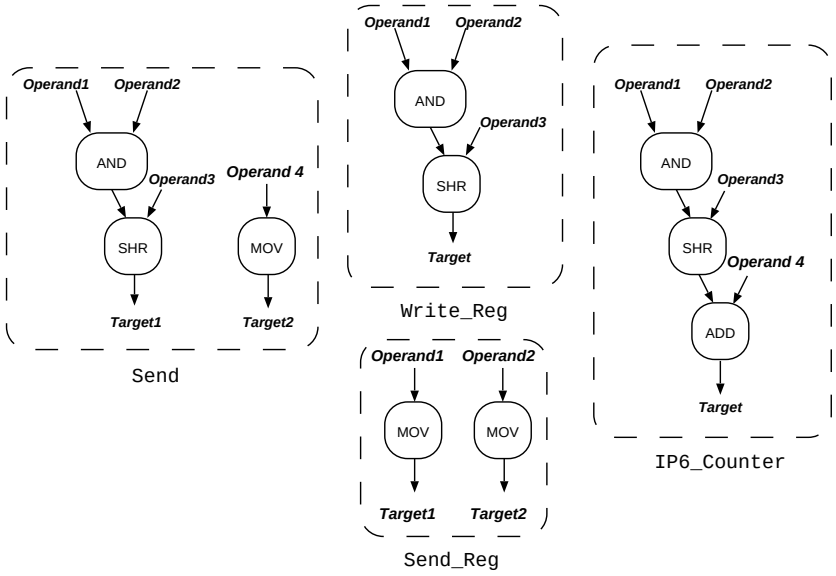


Figure 5.11: *Manually derived compound instructions.*

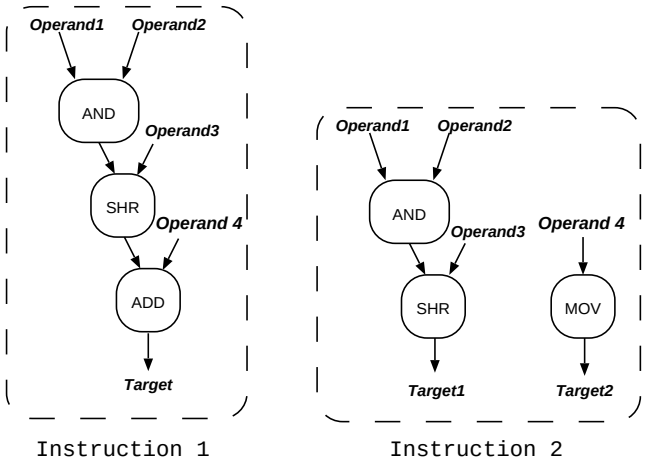


Figure 5.12: *Automatically derived compound instructions.*

comes to a result that is superior to the manual design. This demonstrates the viability of our approach. Moreover, our methodology speeds up the design process through automation and it can handle more complex designs than would be possible manually.

5.3 Summary of Experimental Results

In this chapter we presented performance measurements for our graph-based organization of pattern libraries and compared them with the commonly used linked-list implementations. The results demonstrate that our PSG improves search times on large libraries by orders of magnitude and that the performance gains grow with the library size. For a library of 41233 DAG patterns we measured a maximum speed-up by a factor of 1743. Furthermore, we demonstrated that the size penalty for constructing a combined PSG/IOG stays significantly below the theoretical maximum while the search time advantage is equal to the PSG case. We conclude that using an IOG is viable also for larger problems, providing synergies between patterns.

In the second part of the chapter we presented a case study of an ASIP for network header parsing. We showed step by step how our methods analyze the specified applications and derive an instruction set that meets the timing constraints. We compared the generated instruction set with a manually designed ASIP and showed that our methods arrive at an instruction set that provides the same performance as the manual design but with fewer instructions. The study demonstrates that our methodology is suitable to automate instruction-set generation for control-dominated applications. Automation not only speeds up the design process significantly; automation enables the design of ASIPs for larger benchmark suites that are too complex to be handled manually with reasonable effort.

6 Conclusions

In this thesis, we investigated problems in designing ASIPs in both data-dominated and control-dominated application domains. We observed that current methods focus exclusively on the data-dominated domain and that many of these methods suffer from slow searches in unstructured pattern libraries. Furthermore, none of the known ASIP design methods takes into consideration the synergies between similar patterns that can be combined for leaner implementation.

For instruction-set generation in the control-dominated domain we argued that the main concern is to meet fine-grained timing constraints. None of the known approaches caters for timing. The control-dominated ASIPs described in the literature have been designed by intuition rather than with the help of formal methods.

In the research presented here, we addressed these problems by means of a new library organization on the one hand, and by introducing the first design methodology for control-dominated ASIPs on the other hand. Our proposed solutions are summarized in Section 6.1. We conclude the thesis in Section 6.2 with an outlook on future work for which this research laid the foundation.

6.1 Contributions of this Thesis

The research presented in this thesis has contributed two major concepts to the field of instruction-set generation for ASIPs: a set of new organization methods for pattern libraries and the first ASIP design methodology in the control-dominated domain.

6.1.1 Efficient Pattern Libraries

- We have proposed a novel structure for pattern libraries which we call pattern search graph (PSG). In a PSG, the patterns are arranged as a tree that exploits the structure of patterns to order them hierarchically. We have given algorithms for inserting and searching patterns that have a linear computational complexity w.r.t. the number of edges in the pattern. This is a significant improvement over the multiplicative complexity in linked-list libraries, in particular as PSGs remove the dependency between search time and library size. Hence, PSGs enable the handling of large libraries, eliminating the need for pruning heuristics and permitting the use of exact methods. In our experiments, searches in a PSG were up to 1700 times as fast as in a linked list. The speed-up grows with the library size.
- We have introduced a new method that employs identity operands to disable operation nodes in a tree pattern, in this way reducing it to simpler patterns. The original pattern together with the simpler patterns generated from it are arranged in an identity operand graph (IOG). Each pattern in the IOG can be implemented with the parent pattern by disabling operations using identity operands. Therefore, the IOG represents synergies between patterns that can be exploited for leaner instruction sets and

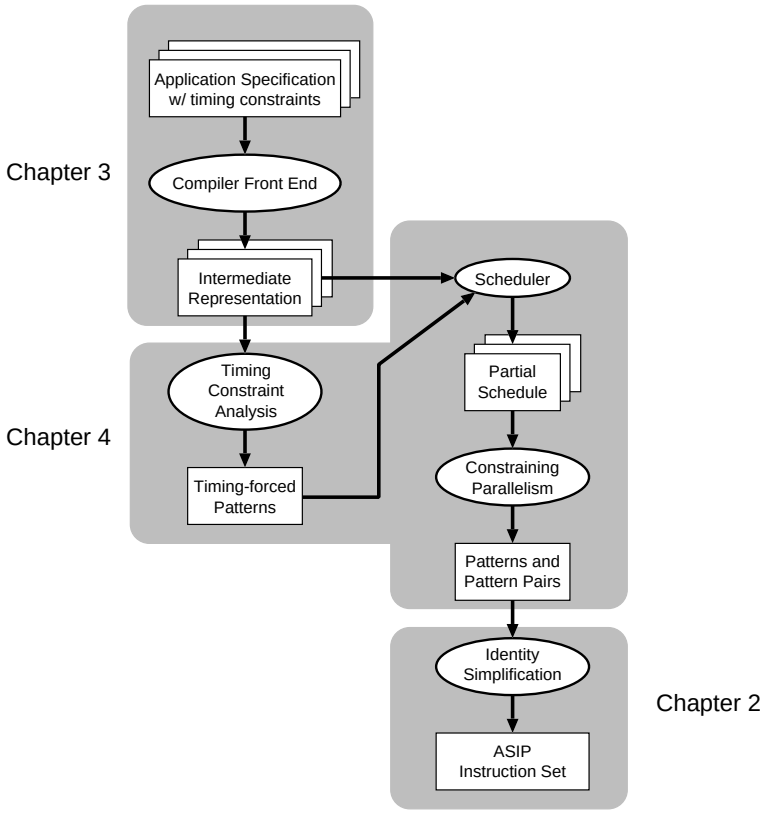


Figure 6.1: *The complete methodology.*

data-path sharing. We have given algorithms to efficiently construct an IOG together with the according PSG, providing the IOG with the PSG access speed. In our experiments, the overhead in library size due to the additional patterns in the IOGs was up to nine-fold compared to a linked list which is far below the theoretical worst-case overhead.

6.1.2 Design Methodology for Control-Dominated ASIPs

- For the specification of the fine-grained timing constraints that are characteristic of control-dominated applications we have integrated new constructs into a C compiler front-end. By declaring variables to be timing-critical we can specify the timing constraints with a granularity of an individual register access. Our method is ANSI C compliant for compatibility with existing tools. Furthermore, C compatibility allows the reuse of an existing code base which for embedded systems in most cases has been written in C.

- We have integrated the application information required for the control-dominated domain in our multi-layer IR (mlIR). It combines the following IRs: DFGs for the data flow within a basic block; the SSA form for the data flow between basic blocks; a CFG for the control flow; a timing layer for the timing constraints between interactions with the environment. We have demonstrated the expressiveness of the mlIR by the example of an optimization technique we call branch postponing that requires information from each layer in the IR. Branch postponing is a new variation of spec-

ulative execution that moves operations out of the critical path of fine-grained timing constraints.

- Control-dominated applications often wait for events in their environment whose timing is only determined by system input at runtime. To model these cases we have introduced a novel data-dependent wait operation, in our timing constructs in C as well as the timing layer of the mlIR. The wait operation can be implemented in several ways, with or without hardware support and with several scheduling possibilities. Our abstract notation makes wait constructs amenable to optimization. We have devised methods to include wait operations in pattern construction and scheduling.

- We have formally defined the optimization problem of instruction-set generation under fine-grained timing constraints. We have presented a heuristic that analyzes given timing constraints and derives the patterns that are necessary to meet the constraints. We achieve this with a novel scheduling algorithm that distributes the available time evenly along paths through the IR. Dependent operations that are scheduled in the same cycle form a timing-forced pattern. The algorithm aims to balance the size of the resulting patterns.

- We have proposed a new method to constrain the number of parallel instruction issues needed by the ASIP down to a number specified by the designer. Applications are partially scheduled to determine the mobility of patterns. Patterns that are scheduled in parallel are bundled into pairs. The pattern pair that has the highest potential to reduce the required parallelism is chosen to be implemented as a combined instruction. The process is iterated until the constraint on parallel issues is met.

We have integrated our methods into a design methodology for control-dominated ASIPs, depicted in Figure 6.1. To our knowledge this is the first methodology that has been proposed in that domain. By comparison with a manually designed ASIP we have demonstrated the viability of our approach: The resulting instructions are similar in both cases, but the IOG optimization results in a leaner instruction set at the end of our formal approach. Moreover, our methodology automates the process which not only provides faster turnaround times but also handles more complex designs than would be possible manually.

6.2 Future Work

6.2.1 Pattern Libraries

The advantageous properties of the PSG opens new possibilities for any method that relies on large pattern libraries. Besides the immediate gain in access speed it will be interesting to see what impact the possibility to handle very large libraries will have. There remains a problem, however, that the PSG shares with many other structures: How to recognize patterns that expose the same behavior but have different internal structures due to commutativity and associativity?

For the IOG we see a applications in HLS for data-path sharing and in code generation for processors with complex instructions.

6.2.2 ASIP Design Methodology

In our design methodology we see two major areas for improvement:

1. In Section 1.3.4 we stated the underlying assumptions and limitations of this work. The most severe of these assumptions is the exclusion of data memories that led to the assumption of unlimited registers. These can be alleviated by the memory-disambiguation techniques devised for VLIW compilers [Ell85]. Furthermore, the inclusion of multi-cycle instructions could take pressure from the register file as it would provide a path to pass data from one cycle to another without touching the register file.

2. Our heuristics leave a lot of room for optimization. Many sophisticated algorithms can be found in other fields, in particular the scheduling algorithms and optimization runs proposed for HLS and compilers. We have mentioned a number of them throughout this work. Transferring these methods to ASIP design will likely improve the results of our methodology significantly.

Bibliography

[AC01] 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.

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[AG85] Alfred V. Aho and Mahadevan Ganapathi. Efficient tree pattern matching: an aid to code generation. In *Proceedings of the 12th SIGACT-SIGPLAN*, pages 334–340. ACM Press, 1985.

[AHM97] David I. August, Wen-mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 92–103, 1997.

[AKMN02] D.I. August, K. Keutzer, S. Malik, and AR Newton. A disciplined approach to the development of platform architectures. *Microelectronics Journal*, 33:881–890, 2002.

[AKPW83] 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.

[API03] 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.

[Arc] Arc International. `http://www.arc.com/`.

[Arn01] Marnix Arnold. *Instruction Set Extension for Embedded Processors*. PhD thesis, Delft University of Technology, Delft, The Netherlands, March 2001.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, July 1998.

[Ben01] Mirko Benz. An architecture and prototype implementation for TCP/IP support. In *Proceedings of the TERENA Networking Conference 2001*, May 2001.

[BHSV90] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[BKKS02] 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.

[BKS04] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st Design Automation Conference (DAC 2004)*, pages 395–400. ACM Press, 2004.

[CEP99] Luis Alejandro Cortés, Petru Eles, and Zebo Peng. A survey on hardware/software codesign representation models. Technical report, Dept. of Computer and Information Science, Linköping University, June 1999.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGH+94] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal methodology for hardware/software co-design of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[CKG+96] Miguel R. Corazao, Marwan A. Khalaf, Lisa M. Guerra, Miodrag Potkonjak, and Jan M. Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Trans. on CAD*, 15(8):877–888, August 1996.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[CPHC04] Newton Cheung, Sri Parameswaran, Jörg Henkel, and Jeremy Chan. MINCE: Matching instructions using combinational equivalence for extensible processor. In *Proceedings of DATE'04*, Paris, France, February 2004. ACM Press.

[CPP+01] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, Daniel Weil, and Sergio Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification 2001*, pages 391–395, July 2001.

[CPP+02] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting ESTEREL semantic on a sequential execution structure. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[CSS98] Keith Cooper, Philip Schielke, and Devika Subramanian. An experimental evaluation of list scheduling. Technical Report TR98-326, Rice University, Houston, TX, September 1998.

[Das85] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[Dit00] 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.

[Dit03] Gero Dittmann. Organizing pattern libraries for ASIP design. Technical Report RZ3488, IBM Research, `www.zurich.ibm.com/~ged/`, April 2003.

[EB94] R. Ernst and Th. Benner. Communication, constraints and user directives in COSYMA. Technical Report CY-94-2, Technical University of Braunschweig, Institute of Computer Engineering, Germany, June 1994.

[EKPD95] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, and Alexa Doboli. Timing constraint specification and synthesis in behavioral VHDL. In *Proceedings of EURO-DAC/EURO-VHDL 95*, pages 452–457, Brighton, UK, September 1995.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985.

[ENF00] Frank Engel, Johannes Nuhrenberg, and Gerhard P. Fettweis. A generic tool set for application specific processor architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES 2000)*, pages 126–130, May 2000.

[EZ99] Network processor designs for next-generation networking equipment. White paper, EZchip Technologies, December 1999.

[GDD+03] 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.

[GM97] Rajesh K. Gupta and Giovanni De Micheli. Specification and analysis of timing constraints for embedded systems. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 16(3):240–256, March 1997.

[Gon00] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March/April 2000.

[Gsc99] Michael Gschwind. Instruction set selection for ASIP design. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*, pages 7–11, May 1999.

[GSK+01] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil D. Dutt, Rajesh K. Gupta, and Alexandru Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of DAC'01*, pages 269–272, 2001.

[HD94] Ing-Jer Huang and Alvin M. Despain. Generating instruction sets and microarchitectures from applications. In *Proceedings of ICCAD-94*, pages 391–396, November 1994.

[HE99] Dirk Herrmann and Rolf Ernst. Improved interconnect sharing by identity operation insertion. In *Proceedings of ICCAD-1999*, pages 489–493, 1999.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Journal of Operations Research*, 9(6):841–848, November 1961.

[Jac90] Van Jacobson. Compressing TCP/IP headers for low-speed serial links. IETF RFC 1144, February 1990.

[Jer04] Ahmend A. Jerraya. HW-SW interfaces abstraction for multi-processor SoC. In *Tutorial at Design, Automation and Text in Europe (DATE'04)*, chapter Programming Models for Multiprocessor SoC. Paris, February 2004.

[Keu87] Kurt Keutzer. DAGON: technology binding and local optimization by DAG matching. In *Proceedings of the 24th DAC*, pages 341–347. ACM Press, June 1987.

[KM90] David Ku and Giovanni De Micheli. HardwareC - a language for hardware design. Technical Report CSL-TR-90-419, Stanford University, April 1990.

[KM92] David C. Ku and Giovanni De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, Norwell, MA, USA, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[Küç99] Kayhan Küçükçakar. An ASIP design methodology for embedded systems. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*, pages 17–21, May 1999.

[KW01] Daniel Kästner and Sebastian Winkel. ILP-based instruction scheduling for IA-64. In *Proceedings of LCTES*, pages 145–154. ACM Press, 2001.

[Lae03] Christian Laetsch. A multi-layer intermediate representation for ASIP design. Master's thesis, EPFL, Lausanne, Switzerland, September 2003.

[LHL89] 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.

[LM97] Birger Landwehr and Peter Marwedel. A new optimization technique for improving resource exploitation and critical path minimization. In *Proceedings of ISSS'97*, pages 65–72, 1997.

[LPMS97] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, pages 330–335, Dec. 1997.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MAHM02] Nahri Moreano, Guido Araujo, Zhining Huang, and Sharad Malik. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 38–43. ACM Press, 2002.

[MBL+96] 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, The Netherlands, 1996.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[Mor98] Robert Morgan. *Building an Optimizing Compiler*. Elsevier Science, Burlington, 1998.

[MS] Michael D. Smith's Research Group on Compilation and Computer Architecture, `http://www.eecs.harvard.edu/~hube/software/`. *Machine SUIF*.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[NT92] John A. Nestor and Vili Tamas. Exploiting scheduling freedom in controller synthesis. In *Proceedings of the Sixth International Workshop on High-Level Synthesis*, pages 74–86, November 1992.

[NW99] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1999.

[OSCI02] The Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0*, version 2.0-q edition, April 2002.

[PD94] Miodrag Potkonjak and Sujit Dey. Optimizing resource utilization and testability using hot potato techniques. In *Proceedings of DAC'94*, pages 201–205, 1994.

[PG87] 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.

[PHZM99] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA: Machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 35th Design Automation Conference (DAC'99)*, pages 933–938, June 1999.

[PK89] 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.

[PK91] 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.

[PKB01] Pierre G. Paulin, Faraydon Karim, and Paul Bromley. Network processors: A perspective on market requirements, processor architectures and embedded S/W tools. In *Proceedings of Design Automation & Test in Europe (DATE 2001)*, pages 420–429, March 2001.

[PSM02] Armita Peymandoust, Tajana Simunic, and Giovanni De Micheli. Low power embedded software optimization using symbolic algebra. In *Proceedings of DATE'02*, 2002.

[RTJ] Real-time for Java. `http://www.rtj.org/`.

[SC98] Jochen H. Schiller and Georg J. Carle. Semi-automated design of high-performance communication subsystems. In *Proceedings of the 31th Annual IEEE Hawaii International Conference on System Sciences (HICCS'98)*, 1998.

[Sha01] Niraj Shah. Understanding network processors. Master's thesis, University of California, Berkeley, September 2001.

[SLWSV99] Marco Sgroi, Luciano Lavagno, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proceedings of DAC'99*, pages 805–810. ACM, June 1999.

[SM01] Hideyuki Shimonishi and Tutomu Murase. A network processor architecture for very high speed line interfaces. *Journal of Communications and Networks*, 3(1), March 2001.

[SRM+94] M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. G. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, HP Labs, November 1994.

[SUIF] The Stanford SUIF Compiler Group, `http://suif.stanford.edu/`. *SUIF2*.

[Ten] Tensilica Inc. `http://www.tensilica.com/`.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

[TTSV00a] Bassam Tabbara, Abdallah Tabbara, and Alberto 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.

[TTSV00b] Bassam Tabbara, Abdallah Tabbara, and Alberto Sangiovanni-Vincentelli. Task response time optimization using cost based operation motion. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES 2000)*, pages 110–114, May 2000.

[TW04] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2/3):157–177, November / December 2004.

[VHDL02] IEEE, New York, NY. *Standard VHDL Language Reference Manual*, 2002.

[VLW00] Seppo Virtanen, Johan Lilius, and Tomi Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proceedings of the 18th IEEE NorChip conference*, November 2000.

[VNO+03] K. Vlachos, N. Nikolaou, T. Orphanoudakis, S. Perissakis, D. Pnevmatikatos, G. Kornaros, J. A Sanchez, and G. Konstantoulakis. Processing and scheduling components in an innovative network processor architecture. In *Proceedings of the 16th International Conference on VLSI Design*, pages 195–201, New Delhi, India, January 2003.

[vPGLM94] J. van Praet, G. Goossens, D. Lanner, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th IEEE/ACM International Symposium on High-Level Synthesis*, May 1994.

[WBC+00] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Venier, and Jacques Pulou. Efficient compilation of ESTEREL for real-time embedded systems. In *Proceedings of CASES'00*, pages 2–8. ACM Press, 2000.

[WC95] Robert A. Walker and Samit Chaudhuri. Introduction to the scheduling problem. *IEEE Design and Test of Computers*, 12(2):60–69, 1995.

[Wea95] G. Weaver. Compiler representations for heterogeneous processing. Technical Report UM-CS-1995-102, University of Massachusetts, November 1995.

[WL01] Jens Wagner and Rainer Leupers. C compiler design for an industrial network processor. In *Proceedings of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001)*, pages 155–164, June 2001.

[WM05] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE'05)*, pages 600–605, Munich, Germany, March 2005. IEEE.

[ZS01] Min Zhao and Sachin S. Sapatnekar. A new structural pattern matching algorithm for technology mapping. In *Proceedings of the 38th Conference on Design Automation*, pages 371–376. ACM Press, 2001.

Biography

Gero Dittmann was born in Rüsselsheim, Germany. During his studies at Darmstadt University of Technology (TU Darmstadt), Germany, he worked as an intern for EDS in Plano, TX, U.S.A., in 1997, and for IBM Research, Zurich Research Laboratory (IBM ZRL), Switzerland, in 1999. Also in 1999, he wrote a research thesis entitled “Intelligent ATM VC Management for Quality of Service Sensitive IP Flows” at the Multimedia Communications Lab of TU Darmstadt. In 2000, he completed his graduate thesis entitled “Programmable Finite State Machines for High-speed Communication Components” at IBM ZRL and received a *Dipl.-Ing.* degree in Electrical Engineering ($\sim$ MSEE) from TU Darmstadt. In the same year, he joined IBM ZRL as a researcher, first in the Network Processor Hardware group, later in the I/O Network Architecture group.

Publications

- David E. Taylor, Andreas Herkersdorf, Andreas Doering, and Gero Dittmann. Robust Header Compression (ROHC) in Next-Generation Network Processors. In *IEEE/ACM Transactions on Networking*, Vol. 13, No. 4, pp. 755–768. August 2005.

- Gero Dittmann and Paul Hurley. Instruction-Set Synthesis for Reactive Real-Time Processors. An ILP Formulation. IBM Research Report RZ3611, June 2005.

- 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*, Lisbon, Portugal. December 2004.

- Gero Dittmann. Pattern Libraries for Fast Searching and Data-Path Sharing. In *Proceedings of the 3rd Workshop on Application Specific Processors (WASP 2004)*, Stockholm, Sweden, pp. 76–83. September 2004.

- Gero Dittmann. US Patent Application, US 10/776788, February 2004. To appear.

- Gero Dittmann. Organizing Libraries of DFG Patterns. In *Proceedings of Design, Automation and Test in Europe (DATE'04)*, Paris, France. February 2004.

- Gero Dittmann, Laurent Frelechoux, and Andreas Herkersdorf. Method and System for Processing Data Packets. US Patent Application, US 2004/0042456 A1, August 2003. Pending.

- Maria Gabrani, Gero Dittmann, Andreas Doering, Andreas Herkersdorf, Patricia Sagmeister, and Jan van Lunteren. Design Methodology for a Modular Service-Driven Network Processor Architecture. In *Computer Networks - Special Issue on Network Processors*, Elsevier Science, Vol. 41, No. 5, pp. 623–640, April 2003.

- Gero Dittmann Organizing Pattern Libraries for ASIP Design. IBM Research Report RZ3488, April 2003.

- Gero Dittmann and Andreas Herkersdorf. Timeout Determination Method and Apparatus. US Patent Application, US 2003/0202482 A1, April 2003. Pending.

- Gero Dittmann and Andreas Herkersdorf. Multi-Layer Intermediate Representation for ASIP Design and Critical-Path Optimization. IBM Research Report RZ3484, February 2003.

- François Abel, Alan Benner, Gero Dittmann, and Andreas Herkersdorf. Method and Systems for Ordered Dynamic Distribution of Packet Flows over Network Processing Means. International Patent Application, WO 03/075520 A2, February 2003. Pending.

- David E. Taylor, Andreas Herkersdorf, Andreas Doering and Gero Dittmann. Header Compression (ROHC) in Next-Generation Network Processors. Washington University in Saint Louis Technical Report WUCSE-2004-70, September 2002.

- Gero Dittmann and Andreas Herkersdorf. Network Processor Load Balancing for High-Speed Links. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002)*, San Diego, California, pp. 727–735. SCS, July 2002. IBM research report RZ3418.

- Gero Dittmann. Load balancer - switch - interface. In *Research Disclosure*, number 454, pages 354–355, disclosure number 454184. Kenneth Mason Publications, February 2002.

- P. Sagmeister, G. Dittmann, A. Herkersdorf, and D. Webb. Scaling Network Processor Performance to 40 Gbps (extended abstract). *IEEE Gigabit Networking Workshop (GBN 2001)*, Anchorage, Alaska, April 2001.

- P. Sagmeister, G. Dittmann, A. Herkersdorf, and D. Webb. Methodology for Testing High-Speed Network Devices with Predicted Traffic (extended abstract). *IEEE Gigabit Networking Workshop (GBN 2001)*, Anchorage, Alaska, April 2001.

- Gero Dittmann. Programmable Finite State Machines for High-speed Communication Components. Master's Thesis, Darmstadt University of Technology, Germany, March 2000.

- Gero Dittmann. Intelligent ATM VC Management for Quality of Service Sensitive IP Flows. Research Thesis, Darmstadt University of Technology, Germany, October 1999.