Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Karel Tuček

SIMD code generator

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. David Bednárek, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2016

Title: SIMD code generator

Author: Karel Tuček

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstract: The center of our interest is a problem of pipelined realisation of a special case of data processing networks. These realisations are supposed to realise some computations on series of independent data sets while utilizing SIMD instructions. The aim of this paper is to theoretically investigate the possibilities and the problems of employment of control flow in these networks and also to implement a general framework suitable for generation of these realisations.

The main idea is utilisation of an algorithm crawling over partitions of a network factorised with respect to its control flow. Our idea is that SIMD parallelism should take place on the same instruction realised across multiple data sets.

We illustrate the problems relevant to employment of branching and loops in these networks. We especially discuss a problem of data ordering and also provide relevant proofs. In the analytical part, we show implementation of a general framework which we believe to be suitable for processing of these networks. We also provide examples utilising Intel's SIMD Streaming Extensions.

Keywords: Processing networks SIMD Parallelism

Contents

1. Introduction

1.1 Overview of our goals

The first goal of this thesis is a theoretical analysis of a problem of vectorised realisation of a specific class of data processing networks. We are especially interested in networks suitable for representation of small fragments (e.g., basic blocks) of computer programs or — more generally — of some computations. Our second goal is the design and the implementation of a framework which would be suitable for generation of these realisations. Our third goal is the implementation of some subset of concepts theoretically described by this thesis. The implementation should provide a tangible proof-of-concept and verify that our framework can be used for task for which it has been designed. Currently, we target a framework for transformation of graph-described network in C code.

1.2 Motivation

The Bobox project [1] is an example of a distributed environment based on a network of components (*boxes*) interconnected by data streams. The structure of the network is defined by an annotated directed graph [1] which may be supplied by various means, e.g., by transformation from an SQL query. Besides such high-level non-procedural sources, transformation from a traditional procedural language is also attempted ([3], [4]), essentially converting procedural code into a graph representing both the control flow and the data flow. However, the elementary operations of a procedural language are too small to be effective elements of distributed processing; therefore, grouping of elementary operations into larger boxes is necessary.

The suggested transformation of a programme may be structurally understood as a process consisting of the following steps:

1. A programme, represented either by some means, e.g., by an SQL query, by a source code or by some form of byte-code, is analysed and transformed into some graph representation.
2. This graph is optimised.
3. The graph is cut into smaller chunks. This transformation should provide graph representation of modules running as sequential (albeit vectorised) code within a distributed environment.

[1] Graph related theory may be found in [2]. Basic understanding of graph-related terminology is crucial for this text.

4. Finally, the graph needs to be converted into some executable form. Various tools may be employed for this purpose. The approach described by this thesis assumes that the module graphs are transformed into C modules with general interface suitable for streamlined processing. These modules may then be connected into a network as described by the global structure of the graph, using a library for physical data exchange and synchronisation.

Put together, the tools may be integrated as shown in Figure 1.1.

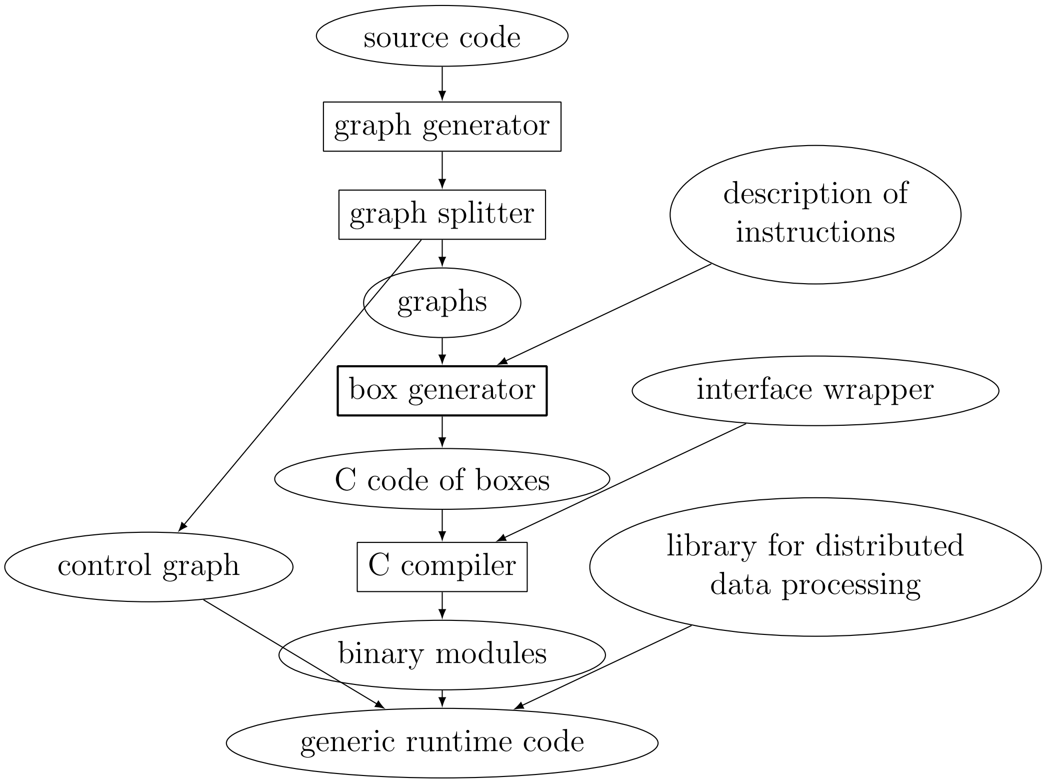


Figure 1.1: The assumed integration of our framework.

Graph generator , such as Parallax ([5]), takes some representation of a programme and transforms it into some graph representation.

Graph splitter cuts the graph into smaller chunks and extracts vectorisable excerpts, which are then send to the box generator. Besides, it creates some representation of a higher level pipeline and some control C code. This piece is currently in early stages of development.

Graphs represent some small computations. Those are the graphs described in the rest of this chapter. The exact form of these graphs may be found in Section 2.1 (Definition 4).

Description of instructions is a definition of the instruction set of the target language and SIMD extension. Exact formats are described by Section 4.5.

Box generator (the goal of this thesis) receives these small vectorisable graphs and transforms them into C modules ('boxes') which may serve as building blocks for higher-level pipelines. These C modules have stream interfaces. This is where our thesis enters the scene. This generator is described by the rest of this thesis.

C code of a box is a C module which realises some computation in a pipelined manner. These boxes provide a general interface (described in Section 4.8).

Interface wrapper is a generic template which serves for translation of general interface of computation generated by the box generator to an interface of some target library. Interface of boxes is further described in Section 4.8. This may be seen as a necessary implementation detail.

A compiler is used to compile all code fragments that require compilation.

Binary modules is a bunch of more or less conventional executable files which contain all introduced computations. These may, for instance, be in a format of dynamic libraries.

Control graph is a description of a task for the target environment. This describes how boxes should be interconnected.

Library for distributed data processing is some library providing tools for distributed data processing. Bobox is an example of such environment. Bobox is further described by [1]. Furthermore, the use of Bobox in this context is covered by [5] and [4].

Generic run-time code is composed of binary modules and of description of the control graph. It is not inaccurate to think of this product as of a distributed environment consisting of computations connected by data streams.

This thesis aims to explore the possibilities of description and realisation of networks in a context similar to the context of the box generator introduced in Figure 1.1. In other words, we would like to know which networks may be realised and how these networks may be realised. Also, for the sake of efficiency, we wish to investigate the possibilities of vectorisation of these networks. Moreover, we would like to design a system capable of generation of code fragments which would realise these networks and which would utilise vectorised SIMD[2] instructions.

Since we wish to output vectorised code, this problem is a problem of a specific case of vectorisation. Vectorisation is usually performed in a procedural context of a single basic block[3] and is well understood in this context. This thesis, however, investigates this problem in an environment in which the computations are

[2]*Single-Instruction Multiple-Data* instructions are instructions which take multiple independent sets of input values and perform an operation parallely on all of them. E.g., performing an element-wise addition of two vectors.

[3]The term *basic-block* is often used in theory of compilers, meaning consistent sequences of instructions without any control flow. Thus, any procedure may be seen as consisting of basic blocks and goto instructions.

guaranteed to be provided with sequences of independent data sets. In other words, our environment is an environment where computations are described by flow graphs. Data in this environment are served in table-like sets with mutually independent rows. Every single row of this imaginary table therefore represents a set of inputs for a predefined computation which is always the same one. We describe this in more detail in Section 1.4 of the Introduction.

A problem which we especially wish to investigate is a problem of employment of control flow[4] in this environment.

1.3 Related work

Overview of methods relevant to compiler construction may be found in [6]. These methods provide a good overview of issues of lower abstraction level, therefore providing us with information how should the target C fragments be composed in order to be efficiently translatable.

A paper focusing on an efficient use of SIMD extensions and on auto-tuning systems is [7]. Papers describing recent attempts of vectorisation in environments containing branching are [8], [9]. Proposed approaches discuss branching methods employing select instructions. These methods provide good results for code with shallow branching, and may be simply employed in our framework by means of preprocessing of an input graph. For this reason we do not investigate branching employed by means of the select operation in this work. However, none of these papers is directly related to our problem, since they assume standard procedural context.

Description of semantics of very general data processing networks may be found in [10]. This description is not suitable for our networks due to its generality. Semantics of processing networks which is more related to our problem may be found in [4] and [3]. This representation is of interest to us since we assume input generated by tools using these representations. However, these flow graphs were designed for different purpose and are unsuitable for our problem.

Information related to the Bobox project is to be found in [1]. Contextual bridge between Bobox and processing networks is provided by [5]. This article provides an alternative view of integration of ParallaX and Bobox. Analysis of another problem, discussing optimisation of processing of SQL queries, which uses a graph-based approach is [11]. This article, together with [3], provides some insight into the actual motivation of ParallaX.

[4]The term *control flow* is generally used for all mechanisms that use some form of branching or loops. Although the general meaning encompasses function calls, we will use this term only for branching and loops.

1.4 Problem introduction, input examples

Our problem is basically turning an intermediate code represented by some oriented graph and stripped of some type of dependencies into a fragment of code in some target language, while utilising some intrinsic[5] SIMD extension. The aim is to achieve a situation in which multiple evaluations of the same computation are processed at the same time by means of the mentioned fine-grained-parallelism.

We assume input in a form of a graph annotated by some operations. We assume this input to be either written by hand, or produced by a specialised software similar to a compiler back-end. We expect this software to identify, extract and preprocess code which is suitable for fine-grained parallelism. An example of this software may be Parallax, mentioned in [5]. We do not provide any integration at the moment since Parallax is still under development and exact interfaces are still unclear.

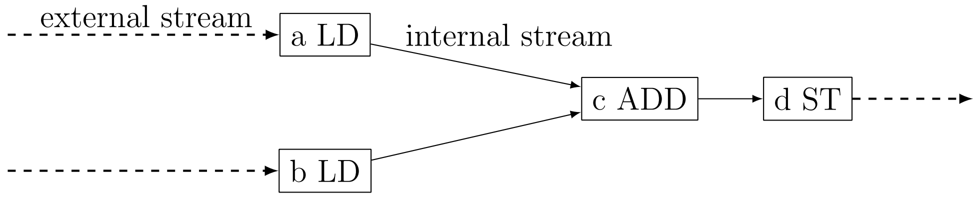


Figure 1.2: A simple adder computation.

Graph in Figure 1.2 represents a computation which takes values from two (external) streams, sums them and produces another (external) stream. The LD operations represent data input for our algorithm, i.e., the LD may evaluate to some effectful[6] expressions. The ADD operation represents addition. The ST operation represents an effectful expression which hands the produced values over to a consumer.

In this explanation, we have used the word *stream* strictly for external streams (managed by a third party). Every external stream stands merely for an input (or an output) for our computation. This makes sense since every computation, represented by a graph, may be seen as a building block of some higher level pipeline. Such pipelines may be (and are supposed to be) constructed by the specialised software mentioned above. However, the internal functionality of these computations also resembles (or will resemble) pipeline computations, and therefore we will use the word *pipeline* for realisations of these computations and the word *stream* for edges in our graphs. We will also talk of these edges as of *queues*.

From this point of view, the load and store instructions may be perceived as operations which merely transform external streams into internal streams. Note that although we needed to make this clear, we will not talk about or work with external streams anymore. Loads and stores will be just data sources and sinks for us. Also, we will not use dashed lines anymore.

[5]Providing direct access to instructions of the target platform

[6]When using the term *effect*, we mean (*impure*) side effects of operations or concealed data flow. We use this word the same way functional programmers do.

This said, this graph surely could be understood as a representation of the C function given in Figure 1.3. However, the semantic meaning we wish to assign to these graphs is different. A more accurate interpretation understands this graph to be a computation[7] which repeatedly invokes the function, with possible overlaps in time. In other words, we have *lifted* the function into a stream context as a functional programmer might have said [8].

```
int f(int a, int b)
{
  return a+b;
}
```

Figure 1.3: C fragment corresponding to Figure 1.2.

Output of our framework is supposed to be a code fragment which receives some input data in a form somehow equivalent to a table such that:

- Columns of this table correspond to input operations in the graph. Any column also may be seen as a single data stream or as a set of synchronous data streams.
- Every line describes input data for one calculation. (For instance, input data for one invocation of a function corresponding to a graph.)
- Different rows represent independent calculations. (These calculations are still defined by the same graph).

The code then produces results per each row of the table, effectively adding some new columns.

a (load)	b (load)	d (store)
1	2	3
4	5	9
7	8	15
3	2	5

Figure 1.4: Example data input and output for Figure 1.2.

The produced code fragment (e.g., C code) may, for instance, take the columns a and b and produce the column d.

The second problem, which we wish to touch on a mostly theoretical level, is employment of control flow in these networks. We wish to implement control

[7]We will try to use this term consistently with its meaning in functional programming, i.e., as a data channel which transforms the data flowing through. However, sometimes we may not manage to keep this promise since in some cases we lack more exact terms.

[8]More information on functional programming may be found in [12].

flow only as an experimental feature meant as a proof of concept. Consider the following C function:

```
int f(int a)
{
  if(a % 2 == 1)
    a = a*7;
  else
    while ( a < 100 )
      a = a+3;
  return a;
}
```

Figure 1.5: Example C fragment employing control flow.

Control flow may be represented in various ways. Since we wish to handle data in a streamlined fashion, splitting and merging data streams seems suitable. We will restrict our attention to control flow in form of branching and loops represented in this fashion. To be more precise, we restrict ourselves to investigate control flow which actually splits data into multiple streams. We also assume nesting of the control flow constructs.

Remark. What we will not develop is the *'computing of all branches on all data and selecting the desired results'*[9] version of the problem. One of the reasons is that it has been satisfactorily investigated before (e.g., [8] and [9]). Another, more pragmatic, reason is that our framework is capable of employing this mechanism by means of standard arithmetic-like operations as long as the input is properly preprocessed.

The control flow thus will be represented by some form of split and merge operations. These may be understood as operations splitting (and merging) data streams in a non-1:1 ratio. In addition, we may use (and require in our input) some special operations. For instance, the provided C function may be represented by the graph in Figure 1.6.

[9]We believe that every serious reader of this work knows what we mean. We provide a more detailed explanation in Section 2.3, but the reader should feel free to skip this reference safely.

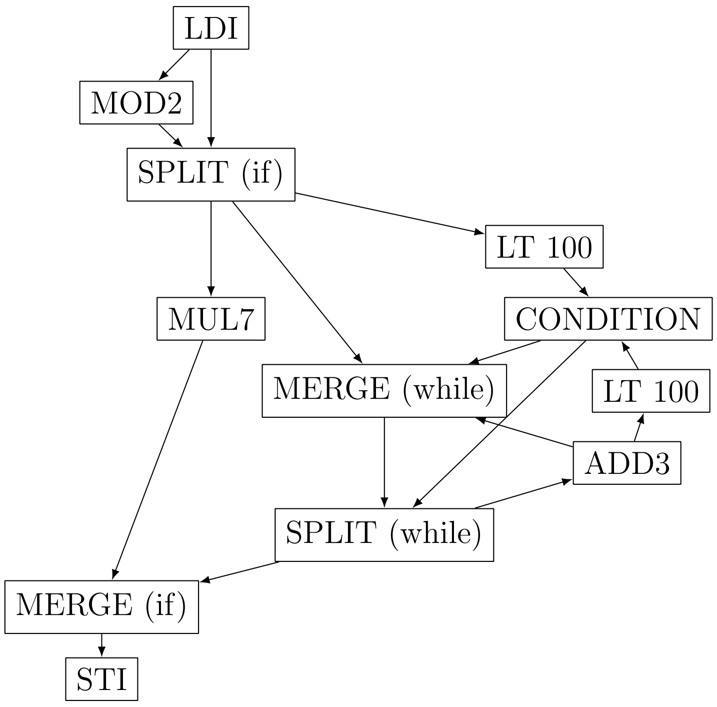


Figure 1.6: Graph representation of Figure 1.5.

ST, LD - again some effectful IO expressions.

LT - *less than* condition with 100 as its second operand.

ADD, MOD, MUL - addition, modulo and multiplication, all used with constant second operands.

SPLIT - a special control flow operation which takes a value and a Boolean and sends the value either to its right or left output depending on the supplemented Boolean.

MERGE - exact opposite of the SPLIT operation.

CONDITION - a special control flow operation which manages loops.

1.5 Overview of employed methods

In this paper, we begin by showing that networks without control flow may be easily transformed into the anticipated form. We produce these results by generating vertices of the input graph in the topological order. This way, the output of every vertex gets stored in a unique variable.

Next thing we show is that every network without control flow may be realised by SIMD instructions only (as long as there is a sufficiently complete SIMD set available). The first step is fusing corresponding operations across multiple invocations of the same computation into SIMD instructions. The second step is adding some data conversions on edges which connect instructions of different vector width or alignment. This way we can process data in packs of LCM[10] of widths of the used instructions. This way the instructions may be fused into blocks which may, for instance, correspond to Figure 1.7:

	index of the processed row							
some operation in G	1	2	3	4	5	6	7	8
another operation in G	1-2		3-4		5-6		7-8	
...	1-4				5-8			
...	1-2		3-4		5-6		7-8	
...	1-8							
...	1	2	3	4	5	6	7	8

Figure 1.7: An example illustrating how operations may by joined across different invocations of a computation.

Regarding control flow, our approach is to cut the input graph into partitions (basic blocks) which process data regularly. That means cutting all control flow nodes[11] into multiple parts and introducing some new data flow between these by some sort of effects. This way we obtain a new graph which consists of basic blocks and control flow edges. We further modify this result to obtain an acyclic graph (or more precisely a graph with some sort of topological ordering[12]). This is shown in Figure 1.8.

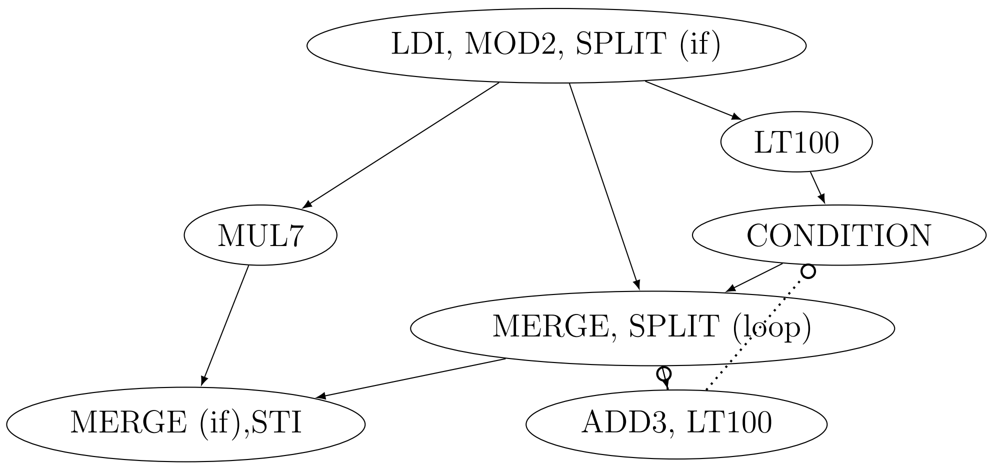


Figure 1.8: Graph from Figure 1.6 split into partitions suitable for processing.

After obtaining an acyclic graph consisting of basic blocks, we try to find an algorithm which crawls this new graph and gradually pushes data through

[10]Least Common Multiple

[11]We use the term *node* as a term relating to some kind of realisation of a *vertex*. We use these terms interchangeably.

[12]Ordering of an acyclic graph, which satisfies dependencies represented by edges. This ordering typically represents an order in which steps of a process may be carried out, e.g., steps in the process of cooking a meal. See [2] for exact definition.

the basic blocks. We use (constant-size) buffers [13] to store intermediate results between these basic blocks.

Remark. To be precise we allow cycles in a specific form which does not void the *semantic* meaning of topological ordering. This may sound strange, but when thinking about it, it is *exactly* what happens when we use loops in programming languages. We discuss this problem in Section 2.3.1.

We should also clarify that our purpose is not to study low-level problematics of compiler construction (these are also well-understood [6]). Although we wish to have all intermediate results allocated into CPU registers, we do not deal with register allocation nor we do restrict our register resources. In many aspects, we heavily rely on standard features of target compilers, such as on register allocation or dead code elimination.

To address the question of assumptions for analytical part of this work, we should state the following points:

- We assume to have at least $O(n)$ space available in CPU registers with respect to size of input graphs. The actual amount of space needed and available is left up to the provider of our input and to the target compiler. Thus, it is a responsibility of the provider to carve up the input into pieces which the target compiler can fit into the registers (or into the first level cache) if the provider wishes them to be fitted there.
- We wish to minimise the amount of transfers between RAM and CPU.
- We wish to utilise fast aligned stores and loads.
- Maximal utilisation of the SIMD extension is preferred, even if the cost needed for data conversions exceeds actual performance gains.

Summary of our results

As we have already stated, we show how graphs without control flow may be solved while utilising a SIMD extension on all data except for some epilogue (e.g. the last few rows of data which do not suffice to fill entire vector). This topic is discussed in more detail in Section 2.2.

Regarding control flow, there is a problem of data ordering (i.e., requirement that all data should be processed and outputted in the original input order). This problem is discussed in chapters dedicated to control flow (sections 2.3.2 and 2.3.3). Furthermore, we discuss the problem of loops, which turns out to be more difficult than originally expected (again, sections 2.3.2 and 2.3.3).

We show two different approaches to this problem - one which preserves the order of the data and one which does not.

[13]This is a memory area which is used to compensate for different processing speeds of components or computations. We use the terms *queue* and *buffer* almost interchangeably.

The first proposed solution (Section 2.3.2), which preserves ordering, may possibly use SIMD instructions on all data in a network without loops. This solution, unfortunately, relies on pull semantics, which allows processing of basic blocks without parallelism. Hence, we have almost no guarantee that SIMD instructions will be employed in runtime. Also, this solution does not work well with loops due to the ordering requirement. Loops can be employed only using the pull semantics, causing data to be processed in non-vectorised manner (in loops).

The second approach (Section 2.3.3) promises a possibility of an unordered solution of networks which contain only branching such that all data are processed by SIMD instructions only (except for some epilogues and some split and merge internals). We show that loops can be added to this solution with some memory-consumption penalty without loss of parallel processing. This solution, unfortunately, relies on the ability of deciding whether two conditions are equivalent, which is something we are at the very least not able to verify in some cases since this guarantee may lay in effects. Still, implementation of this solution may be possible with some heuristics on general graphs and is possible with some additional restrictions on form of control flow present in input.

We implement a general framework designed for the introduced task. Our implementation is designed to be easily extensible and also to be general with respect to SIMD extension, programming language and target environment — it may also be seen as a highly sophisticated text processing environment.

We implement the basic SIMD code generator and also all graph transformations needed for the ordered solution of control flow. Furthermore, we implement a generic set of macros which implements a register-allocated buffer and use these macros for implementation of generic split and merge instructions. Using these, we implement a showcase of the first solution which can handle branching. Our solution has significant overhead on control flow instructions, but that has been expected. There is a room for improvement, but the amount of engineering efforts needed for full and optimal implementation greatly exceeds the scope of this work.

Besides the actual framework, we provide an example instruction table for the Intel Streaming SIMD Extension (SSE) in version 4.2. This instruction set contains all standard operators of the C language implemented for variety of data types (8-64 bit signed and unsigned integers, bools, floats and doubles). This table, although being thoroughly tested using a test-graph generator tailored for this purpose, is not guaranteed to be bug-free due to its vast scope.

1.6 Note on thesis structure

The rest of this thesis is organised as follows:

- Chapter 2 begins by formal introduction of the problem. Then Section 2.2.1 follows and deals with the simple case of networks without any control flow. Section 2.3 discusses the problem of control flow in-depth.
- Chapter 3 introduces the architecture of our framework on high level of abstraction, explains its basic working principles and mentions concrete algorithms which were employed.
- Chapter 4 describes technical details of the implementation, APIs, input file formats and exact semantics of their fields.
- Chapter 5 provides a brief introduction to the SSE extension and explains how the provided example SSE instruction set is built.
- Chapter 6 summarises our results in concrete context of the previous chapters.

2. Theoretical analysis and solution

2.1 Formal introduction, definitions, notation

We provide a significant number of definitions of our own. We construct this formalism since this problem has not yet been described in this exact form (as far as we know). We provide this formalism mainly for the sake of clarity. We only define the minimum formalism necessary for this thesis. Since introducing the entire formalism at once does not seem wise, we choose to build it incrementally. We provide some forward references and some limited amount of forward context to keep the flow of information as consistent as possible, but our main goal will be providing information at places where these are actually used.

In this section, the reader should get some intuitive understanding of the meaning of flow graphs, of what flow graphs represent and should note the difference between instructions and operations. The reader should feel free to skip formal details.

Introduction of flow graphs

Definition 1 (Edge incidence)**.** *Let $G(V, E)$ be a multigraph. We define the relation $\sim$ by the following rules:*

- $\sim \subseteq E \times (V \times V)$
- $(\forall e \in E)(\forall a, b \in V)(e \sim (a, b)$ *if and only if edge e leads from vertex a to vertex b*)

Definition 2 (Powerset)**.** *We will use the symbol $\wp(V)$ to denote the* power set *of the set V. The power set of the set of V is defined as a set of all subsets of the set V.*

Definition 3 (Data queue)**.** *We will use the term* data queue *or simply* a queue *for a first-in first-out container with the following operations:*

push *adds an element to the container*

pop *returns the oldest element present in the container and removes it from the container*

We will use the term data queue *in its conceptual meaning. In real implementation, we use constant-size buffers or simple memory fields for objects which we theoretically describe as queues.*

Definition 4 (Flow graph)**.** *We will use the term* flow graph *for any tuple* $(G(V,E), O, \mathit{From} : E \to \mathbb{N}, \mathit{To} : E \to \mathbb{N}, \mathit{Op} : V \to O, \mathit{Inputs} : () \to \wp(V),$ $\mathit{Outputs} : () \to \wp(V))$ *which satisfies the following conditions:*

- $G(V,E)$ *is a nonempty, connected, directed multigraph with annotations defined by* From*,* To *and* Op*.*
- *Let* G *be acyclic except for edges which are explicitly allowed to be on cycles by semantics of* O *and by node classification provided later.* [1]
- O *is a set of operations.*
- *Vertices in* Inputs *have no incoming edges and vertices in* $\mathit{Outputs}$ *have no outgoing edges. Also, we require that the* Op *annotations of these vertices have semantics of input or output nodes as described later in Definition 13.*
- *Any directed edge is uniquely identified by its destination vertex and the* To *annotation. I.e.:*

$$\begin{gathered}(\forall e, f \in E(G))(\exists a, b, c, d)\\ (e \sim (a,b) \wedge f \sim (c,d) \wedge b = d \wedge to(e) = to(f) \Rightarrow e = f)\end{gathered}$$

We will write $G(V,E)$ *(with* O*) for brevity.*

This definition resembles the concept of Kahn networks ([10]) as well as Hybrid flow graphs ([4]).

Flow graphs represent pipeline computations. Edges represent data flow. The edges may also be understood as data queues. Vertices represent data transformations. The *To* annotation identifies arguments for operations represented by members of O. The *From* annotation identifies outputs of operations which return more than one value. The nullary functions *Input* and *Output* denote sets of nodes which are meant as data inputs or outputs.

We provide an example of a simple flow graph with all annotations explicitly shown in Figure 2.1. Later, we will omit the *From* and *To* annotations entirely, while showing the *Op* annotation without explicit labels.

A semantical meaning of the graph may be:

a,b - First, the two load (LD) operations produce two values from somewhere, typically from some two (distinct) containers stored in RAM, and store these values into the corresponding outgoing queues of a and b (which will be typically implemented as some register-allocated variables).

[1]This requirement may be ignored for the time being since there are no general restrictions on semantics of O. We include this requirement since implicit acyclicality will greatly simplify enumerations of assumptions.

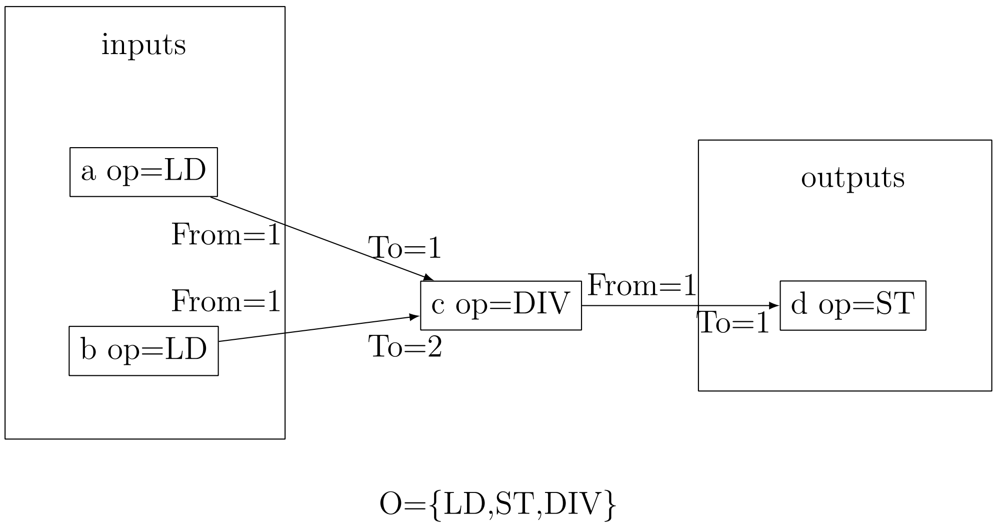


Figure 2.1: A flow graph with all its annotations shown.

c - The DIV operation gathers the two values from the incoming queues, performs division, and pushes the result into its outgoing edge. Note that the *To* annotation provides the information that the input from a should be divided by the output from b and not vice versa.

d - The store operation retrieves the result from its incoming queue and stores them somewhere. Again, this operation may represent almost any effect.

Remark. The reader may be asking why G is required to be (implicitly) acyclic. The main reason is that this way of thinking of flow graphs matches the way we use the flow graphs. More specifically, the apparent problem is that unrestricted loops may introduce unsolvable dependencies. Later, we will define special node types which are allowed to void acyclicality in exchange for a guarantee of not introducing cyclical dependencies. This way, a topological ordering in its *semantic* sense still exists. (Although not in its definitorical meaning.)

Relation between flow graphs and computations

This subsection formalises how flow graphs generally describe algorithms. This formalism roughly corresponds to the information informally introduced by semantics of our example.

Definition 5 (Pipeline of a flow graph)**.** *Let $G(V, E)$ with O be a flow graph. We will use the term* pipeline *(of flow graph G) for a tuple $P(G(V, E), O, Q, Queues)$ which adds* a data queue *to every edge of G. I.e., we add a set Q of data queues and a bijection $Queue : E \to Q$.*

This definition adds operational semantics to flow graphs. While flow graphs describe input computations, pipelines will let us reason about actual evaluation.

Definition 6 (Data row)**.** *Since input data for realisations of flow graphs may be understood as tables consisting of mutually independent rows of data, we will* *use the term* data row *for a single set of input values for a flow graph G. I.e. we will use the term* data row *for a function $\mathit{Inputs}_G \to D$ for an arbitrary set D of values which is consistent with semantics of operations of G.*

We will also (informally) say that all the values computed from a data row belong to this data row. This will be always obvious from context.

Definition 7 (Operation and instruction)**.** *We will use the term* operation *for general identification of an operation which is to be performed on some data. The term* instruction *will be used for a specific way of performing an operation. This way, multiple instructions may be associated with a single operation.*

E.g., multiplication by a constant power of two is a unary operation. Associated instructions may be:

- binary shift to left
- multiplication using the asterisk operator
- a vectorised version of multiplication

Definition 8 (Semantics of an operation)**.** *When talking about semantics of an operation (or of a set of operations), we mean:*

- *Structural requirements on any flow graph which contains vertices representing the operation, such as:*
  - *requirements on existence and counts of incoming and outgoing edges and their From and To annotations;*
  - *data type semantics of O;*
  - *any other semantic restrictions of O relating to the graph structure;*
- *Description of a computation which transforms its input data into its output data.*

We do not consider consistency of runtime semantics (such as semantics of division by zero) of operations in this text. We assume that either all problematic semantics are solved as part of the semantics defined above or that all data are valid with respect to the structure of the flow graph which is supposed to process them.

Definition 9 (Data transformation on vertex)**.** *Let P be a pipeline of a flow graph $G(V, E)$.* Data transformation *on a vertex v will pop some elements from some incoming queues of v and push some elements into outgoing queues of v. An element pushed into an outgoing queue e of v will be determined by the elements taken from the incoming queues of v, their To annotations and the annotation $\mathit{From}(e)$ according to the semantics of $\mathit{Op}(v)$. The transformation is allowed to perform at most one operation on every input or output queue of v.*

Algorithm 1 (Greedy realisation algorithm)**.** *Let P be a pipeline of a flow graph $G(V, E)$ with O with possibly nonempty queues. We shall call the following Algorithm* greedy realisation algorithm *(GRA).*

```
fun GRA(P as defined above)
{
  while P contains a vertex v s.t.
    - incoming queues of v contain amount of data
              required by semantics of op(v)
    - v does not belong to inputs or outputs of G
  {
    perform transformation denoted by op(v) on P;
  }
  return P;
}
```

Definition 10 (Realisation of a flow graph on a single data row)**.** *Let $G(V, E)$ be a flow graph.* Realisation of a flow graph *G will be an algorithm satisfying the following conditions which produces content of incoming queues of vertices in $Outputs_G$ when supplemented with a single data row d:*

- *G is consistent with semantics assigned to O.*
- *Input consists of exactly one value per every outgoing queue of a vertex in $Inputs_G$.*
- *Output consists of exactly one value per every incoming queue of a vertex in $Outputs_G$.*
- *Output is equivalent to the output obtained by the following procedure and is unique*[2]*:*
  1. *Let P be a pipeline of G with queues initialised so that:*
     - *A queue $q \in Inputs_G$ is initialised to a single value, defined by the data row d.*
     - *The queue q is initialized to an empty queue otherwise.*
  2. *Run GRA on P.*
  3. *Retrieve all values from incoming queues of vertices in $Outputs_G$ as output.*
- *The GRA is required to consume data from all queues except the output queues, i.e., the incoming queues of $Outputs_G$.*

Definition 11 (Realisation of a flow graph)**.** *Let G be a flow graph. Let D be a set of (input) data rows for graph G.* Realisation of the flow graph *G will be an algorithm which (being supplemented with D) produces results which are equivalent to the results obtained by a separate application of the* realisation of a flow graph on a single data row *on every data row in D.*

[2]This is an implicit requirement on semantics of O. This requirement causes that realisation does not exist if the semantics are not unambiguous.

Note that our definition does not perform any input or output operations. The reason for this is that we do not wish to deal with effects in the theoretical analysis.

Also, note that our definition of flow graphs does neither require nor ensure semantics of O to be consistent in G. Also, the definition of a realisation requires but does not ensure that GRA produces exactly one value per every output vertex. Thus, some flow graphs may have no realisation. We will implicitly assume our flow graphs to be consistent with some semantics assigned to O. This semantics will always be clear from their context.

Definition 12 (Consistency of a flow graph)**.** *We shall say that a flow graph $G(V, E)$ with a set of operations O is* consistent *if and only if a realisation of this flow graph on a single data row exists.*

To summarise the definitions provided so far, we have defined what a flow graph is and how it describes some class of algorithms. We have also described many formal details which may be seen as implementation details but which were crucial for formal correctness of the provided formalism.

Definition 13 (Node types)**.** *We shall distinguish the following types of operations (this type of operation is to be considered a part of semantics of operation):*

regular node *will be a node with an operation whose semantics is taking one element from every incoming queue of a vertex and pushing one element into every outgoing queue of this vertex. Data to be pushed into an outgoing queue may depend only on the input data and the* from *annotation of the outgoing queue.*

input or output operation *nodes represent either consumption or production of exactly one value per data row. These nodes are required to be present either in $Inputs_G$ or $Outputs_G$. These will not participate in the GRA algorithm due to reasons described above.*

control flow node *will be any other node. We will classify these in Definition 16 since exact semantics of these nodes is implementation-dependent.*

Some SIMD related definitions are postponed into Section 2.2.2.

2.2 Code generation

2.2.1 Simple code generation

Consider the following problem:

Problem 1 (Simple code generation)**.** *Let $G(V, E)$ be an acyclic flow graph consisting of regular operations. Let O denote some subset of arithmetic operations of the C language. Also, let G be consistent with the semantics of these operations. We wish to generate a C code which realises the graph G.*

In other words, we wish to generate code for a simple basic block without any control flow, such as the one shown below:

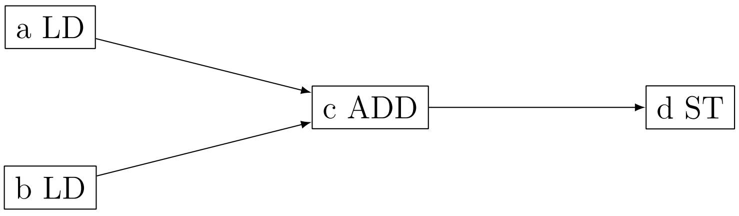


Figure 2.2: Adder computation.

We wish to generate one instruction per every vertex so that every vertex receives data from its predecessors. The resulting fragment of code may look like this:

```
int var_a = input_0[i];
int var_b = input_1[i];
int var_c = var_a + var_b;
output_0[i] = var_c;
```

Figure 2.3: Code fragment realising the adder computation from Figure 2.2.

Remark. We shall explicitly note that the input and output arrays, as well as the i variable, are some environment-dependent effectful expressions. These make sense for some loop processing, but are by no means part of a generalised algorithm. Other variable names are chosen arbitrarily by the algorithm. The plus operator is an instruction assigned to the addition operation. Types are part of semantics of operations that were translated into the results.

Solution to this problem is straightforward. First, we order vertices of G topologically. Then we take vertices in this order and process them one by one. For every vertex v we use its annotation to transform its input data into its output data. Then we save the output data as a new variable under some unique identifier for later use. Corresponding algorithm is Algorithm 2.

Algorithm 2 (Simple code generator)**.** *Let G be a flow graph consisting of regular nodes. We define the* simple code generator *algorithm as follows:*

```
fun enc(operation o)
  { return any instruction associated with o}
fun gen_instruction(instruction, result_name, arg1, arg2, ...)
  { generate actual text representation from arguments; }
fun vertex::result
  { return the name used for storage of the result }
fun vertex::in(index)
  { return u such that there exists e == (u, v) and to(e) == index; }

fun generator(graph G)
{
  for( v vertex in V in topological order)
  {
    arguments;
    result_name = "var_$v";
    gen_instruction(enc(op(v), result_name,
      v.in(1).result(), v.in(2).result(), ...);
    v.result() = result_name;
  }
}
```

Form of patterns

We may have noticed that vertices are basically of three types. This allows us to construct simple patterns which may be used for actual code generation (e.g., by the *gen_instruction* function shown in Algorithm 2). These may be the following:

- Input – `"$type $name = $input;"`
- Operation – `"$type $name = $operation;"`
- Output – `"$output;"`

Where `$input` and `$output` evaluate to environment-dependent expressions, `$name` evaluates to a new identifier and `$operation` evaluates to a pattern identified by the *Op* annotation. The operation pattern may contain `$arg1`, `$arg2`, `$arg3`... expressions. These evaluate to the saved names from incoming vertices.

This scheme of generation is actually used — our generator is driven by quite complex text-processing system which is based on recursive evaluation of shell-like variables in broader contexts provided by various extensions.

2.2.2 vectorised code generation

Definition 14 (Instruction width)**.** *We will use the term* width *to denote the number of independent data rows a which are operated by a single SIMD instruction.*

Definition 15 (Width conversion)**.** *We will use the term* width conversion *for an instruction which takes data from an instruction of some width and transforms them to data suitable for another instruction which is of a different width.*

Consider the following problem.

Problem 2 (vectorised code generation)**.** *Let $G(V, E)$ and O be an acyclic flow graph defined as in the* simple code generation *problem. Furthermore, let there be a set I (of SIMD instructions) and a partial function $Enc(Operation, Width)$* [3] *$: O \times \mathbb{N} \to I$. Let Enc be defined at least on $O \times \{1\}$. Also, let there be a set of width conversions sufficient for direct conversion between any two widths present in I. Finally, let all widths be powers of 2. We wish to generate C code which will realise the graph G using instructions of highest possible widths. In other words, every vertex will be performed by the broadest associated instruction (I.e., the broadest instruction which realises the operation defined by its Op annotation).*

If highest instruction widths of all operations present in gG were the same ones, we could use the *simple generator.* We would simply replace all types by the corresponding vector versions and use vectorised counterparts of all instructions. Typically, the width will vary with the bit length of the data type in question and therefore we cannot use this approach with more than one data type present in G.

This algorithm may be easily corrected:

- Generate a number of SIMD instruction which suffices for processing of some w data rows. Do this for every vertex.
- Add some width conversions which adapt data outputted by one instruction for the width of the next instruction.

As an example, we provide a simple computation which performs modulo 4 on integers. We use non-vectorised store and load instructions and a vectorised mod4 instruction on 16 bit integers as our instruction set. We represent vectorised instructions by well known C constructs since we believe such representation to be brief and readable.

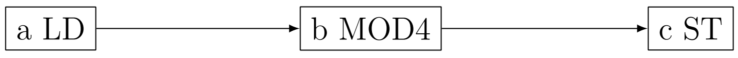


Figure 2.4: A computation performing modulo 4 on a data stream.

[3] We are merely saying that we wish to have some vectorised instructions available and that every operation should be defined at least in its non-vectorised version.

```
uint_16 a_0 = input_a[i++];
uint_16 a_1 = input_a[i++];
uint_32 a_conversion_w2_0 = a_0 | (a_1 << 16);
uint_32 b = a_conversion_w2_0 & 0x00030003;
uint_16 b_conversion_w1_0 = b & 0xFFFF;
uint_16 b_conversion_w1_1 = (b & 0xFFFF0000) >> 16;
output_c[j++] = b_conversion_w1_0;
output_c[j++] = b_conversion_w1_1;
```

Figure 2.5: C fragment realising the modulo 4 computation from Figure 2.4 in vectorised manner.

In this example, we have generated two LD instructions (lines 1-2), one MOD4 instruction (line 4) and two ST instructions (lines 7-8). Line 3 merges the outputs of vertex a, adapting the data for the vectorised modulo operations. Lines 5 and 6 split their output into two values which are suitable for single-value stores on lines 7 and 8.

This problem may be better visualised by a table whose cells represent actually generated instructions. Numbers in cells represent indices of data rows which the instruction processes. The width conversions are not shown, but may come in as new rows. Assume that we want our algorithm to process four values in one step instead of two (or just assume we are showing two invocations at the same time).

a (LD)	1	2	3	4
b (MOD4)	1-2		3-4	
c (ST)	1	2	3	4

Figure 2.6: Table visualising operations joined across different data rows.

Remark. The condition requiring all widths to be powers of two ensures two things:

- All data vectors are aligned in the meaning that the difference between indices of the first values of any two instances of instructions is a multiple of GCD of their widths. But GCD of two powers of two is always the smaller of the two numbers. This means that the first index of the wider of the two instructions is always aligned with the first index of some instance of the smaller one.[4]
- We are able to process the same amount of data (w) at every instruction.

Without this condition, we would need w to be the least common multiple of the maximum widths of all operations in G.

[4]If the meaning of this claim is unclear, the reader may like to try looking at the table in Figure 2.6 for a few seconds. We have merely said that the left boundary of the 3-4 cell in the MOD4 row is aligned with the left boundary of the cells with index 3 at the rows of ST and LD operations.

Thus, pseudo code generating the result from Figure 2.5 may be:

```
for(v in V) in topological order
{
  generate adaptation for width of v
  generate code of v
}
```

The same algorithm in more detail — first, we find maximum width w of an instruction which may be used in G (we do not consider operations which are not in G at all). Then we generate code processing w data rows in one iteration. Again, we take vertices of G in topological order and process them one by one. For a vertex whose broadest instructions is of width n we generate w/n instances of its broadest instruction. Arguments are again retrieved from incoming vertices, only now we work with a set of names (of variables holding results of the vertex in question) per every width that was generated. I.e., when generating a code of a vertex we check whether there is a set of names (variables) for width n. If there is one, we use it. If there is no such set, we use some width conversion to obtain data for width n (i.e., generating a new set of variables for width n). We show code of this algorithm in Algorithm 3.

Exactly the same algorithm, just with some implementation details filled in:

Algorithm 3 (vectorised code generator)**.**

```
fun gen_instruction(instruction, result_name, arg_1, arg_2, ...)
  { generate actual text representation from arguments; }
fun gen_conversion_ins(from_width, to_width, result_names, arguments)
  { generate actual text representation from arguments; }
fun vertex::in(index)
  { return u such that there exists e == (u, v) and to(e) == index; }
fun vertex::results(w)
  { return array of names for width w }

fun gen_conversion(v, to_width)
{
  from_width = native width of v;
  for(int i = 0; i < w; i+= max(from_width,to_width))
  {
    result_names, arguments;
    for(int j = 0; j < from_width/to_width || j == 0; ++j)
      result_names.push("$v_conversion_w${to_width}_${i+j}");
    for(int j = 0; j < to_width/from_width || j == 0; ++j)
      arguments.push(v.results(from_width)[i+j]);
    gen_conversion_ins(from_width, to_width, result_names, arguments);
    v.results(n).push(result_name);
  }
}
```

```
fun gen_vertex(v)
{
  for(vertex u : incoming to v)
    if( u does not have results for width n )
      gen_conversion(u, n);
  for(int i = 0; i < w/n; ++i)
  {
    result_name = "$v_$i";
    gen_instruction(enc(op(v), n), result_name,
        v.in(1).results(n)[i], v.in(2).results(n)[i], ...);
    v.results(n).push(result_name);
  }
}

fun generator(graph G)
{
  for(v in G in topological order)
    gen_vertex(v);
}
```

2.2.3 Indirect width conversions

One thing which should be addressed at this point is the problem of indirect width conversions. We have avoided this problem by assuming that direct width conversions always exist in the *vectorised generator* problem (Problem 2). Although this problem does not seem difficult, we have not been able to find an optimal solution which would work in a feasible complexity.

The simplest approach is generating all conversions which lay on the shortest (or cheapest) path between the data provider and the data consumer. This approach may produce suboptimal results if there are multiple consumers. This algorithm may be further improved if we recalculate distances after production of every single path (considering vertices of the generated path to have distance 0). Unfortunately, this improvement still does not ensure optimal results, as is shown by the following counter-example.

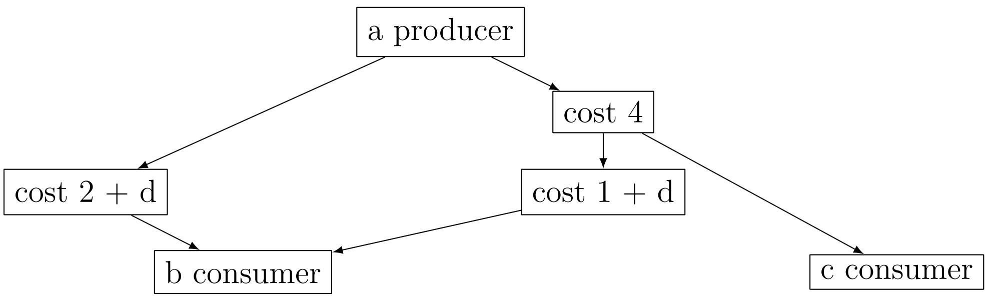


Figure 2.7: An example showing that iterated shortest path algorithm is not optimal.

Producer, consumers are standard instructions (representatives of operations).

Other nodes represent width conversions and are annotated by some cost. These nodes may be replaced by simple paths if we wish to have a counter-example without costs.

Edges represent possible data flow, i.e., edges connect nodes whose outputs are of the same width as the width of the input of the other node.

The improved shortest/cheapest path algorithm may first choose the path $a \to 2 + d \to b$ and then the path $a \to 4 \to c$, yielding a path of cost $6 + d$. The cheapest path is apparently $5 + d$. The variable d shows that strategies like *cheapest path first* or *most expensive path first* do not solve the problem.

This problem may be formulated as an integer linear programme[5]. However, this does not imply existence of a polynomial solution since integer programming is NP-hard.

Algorithm 4 (ILP of minimal distribution tree)**.** *Let P be the set of a single producer, C be the set of all consumers and W the set of all possible widths. Let $G(V, E)$ be a graph such that:*

- $V = P \cup C \cup W$
- $E = \{(u, v) | u, v \in V \wedge \textit{there exists a width conversion from } u \textit{ to } v\}$

Define variables and constants as follows:

- $m = \max_{v \in V} deg(v)$
- $(\forall e \in E)$*: let there be a variable* $f_e \in \mathbb{N}_0$.
- $(\forall e \in E)$*: let there be a variable* $i_e \in 0, 1$.

Define restrictions as follows (we construct flow in network):

- $(\forall v \in W)$: $\sum_{e \in in(v)} f_e - \sum_{e \in out(v)} f_e = 0$ *(Kirchhoff's law)*
- $(\forall v \in P)$: $\sum_{e \in in(v)} f_e - \sum_{e \in out(v)} f_e = -|C|$ *(source)*
- $(\forall v \in C)$: $\sum_{e \in in(v)} f_e - \sum_{e \in out(v)} f_e = 1$ *(sinks)*
- $(\forall e \in E)$: $f_e - m * i_e \leq 0$ *(indicator for minimisation)*

minimise $\sum_{e \in E} i_e$

[5]Linear programming formulates problems as systems of linear equations. More information may be found in [13]

Proof of correctness of a feasible solution. Suppose for a contradiction that a feasible solution exists but there is no nonzero flow from $p \in P$ to a vertex $c \in C$. Let $A = \{v \in V|$ path from $p \in P$ to v exists over edges with $f_e > 0\}$. We shall count the sum $S = \sum_{v \in A}(\sum_{e \in in(v)} f_e - \sum_{e \in out(v)} f_e)$ in two ways:

- By sum of the restrictions $S = |C \cap A| - |C|$. This is a negative number since $|A| < |C|$.
- By contribution of edges:
  - Edges in A contribute by 0.
  - There are no edges with $f_e > 0$ going from A due to the definition of A. Thus, all outgoing edges contribute by 0.
  - All edges going to A contribute by a non negative number.

  This implies that $S \geq 0$. But that contradicts the previous result.

Thus, if a feasible solution exists, then also a path from $p \in P$ to any $c \in C$ with nonzero indicators i_e exists. Therefore, this solution is correct. $\square$

Proof of existence of a feasible solution. For every $p \in P$ and $c \in C$ construct a flow of size 1 from p to c. Let f_e be the sum of all these flows. Furthermore, if $f_e > 0$ let $i_e = 1$. Let $i_e = 0$ otherwise. This evaluation apparently fulfils all restrictions and therefore if a solution of the original problem exists, a feasible solution of the presented ILP also exists. $\square$

2.3 Handling control flow

At this point, we would like to introduce control flow into flow graphs (i.e., branching and loops). Branching may be done in two ways:

- By performing both branches on all data rows and then joining the data rows using a *select* operation. This approach is, for instance, described in [8] and [9], and may be easily employed in our framework by regular instructions.
- By reordering data vectors and performing every branch only on data that are supposed to be handled by the branch in question. This approach has significant overhead on reordering, but may be advantageous in case of nested branching.

We will investigate only the second approach, since the first approach may be easily employed by use of regular operations, provided that the input is in a form using these operations and also that there is an implementation of the *select* operation available. The second approach also allows processing of loops. [6]

First, we shall introduce some new types of non-regular nodes. These will not be required to fulfil the condition that an operation has to pop/push the same amount of data to/from every queue (defined by regularity in Definition 13). Note that this definition addresses semantics in context of a single data row.[7]

Definition 16 (Node types). *We shall distinguish the following types of operations:*

regular operation – *As defined in Definition 13.*

input or output operation – *As defined in Definition 13.*

split operation *will take one data input and one condition input (Boolean value typically). Depending on the condition input it will decide upon exactly one of its outgoing From annotations and push data into all outgoing queues with this annotation. All other outgoing queues will remain unchanged. The data input and the condition input will be consumed.*

[6]The conventional approach for loops would be expanding the loop body multiple times and then optimising this result, possibly pipelining operations of one loop iteration over multiple resulting iterations and possibly employing SIMD instructions on any nondeterministicaly discovered pairing of instructions ([6]). Also, some more advanced approaches such as polyhedral analysis of loops may be employed ([14]). The conventional methods have the benefit of low overhead and lower independence requirements. Our situation differs by the fact that the main parallelisation takes place above our generator. This means that the loops present in our input will typically have very small bodies which makes them unsuitable for the cited methods.

[7]Semantics in vectorised case are defined to be equal to multiple instances of a single context.

merge operation *will take two data inputs and one condition input. Depending on the condition input it will take data from one of the inputs and push them to all output queues. Data in the chosen input and in the condition input will be consumed. All other incoming queues will remain unchanged.*

loop merge operation *will take two data inputs. If there is any data in the second input* [8]*, this node takes the data from the second input and puts it into all outgoing queues. Otherwise, it takes the data from the first input. Moreover, we will allow the second input to be on cycles*

loop condition *will be an operation which takes two condition inputs and provides two condition outputs. If there is any data in the second input, this node takes an element from the second input. Otherwise, an element from the first input is taken. Results are shown in the following table (this way one column is applied per every consumed element).*

index of input, value	1, false	1, true	2, false	2, true
output 1	true	true	false	false
output 2	true	false	true	false

Figure 2.8: Function of the *loop condition* node

This table may be interpreted as follows:

	true	false
output 1	process element which enters the loop	process element which already is in the loop
output 2	send this element out of the loop	send this element into next iteration

Figure 2.9: Alternative semantics of table from Figure 2.8

In addition, we allow this node to have a non-negative integer as an internal state. This integer will be always equal to the difference of the numbers of true and false values sent into the second output. Furthermore, we will restrict the option of taking data from the first input by requirement that (the absolute value of) the difference is less or equal to some positive number. This integer will be denoted by a partial function $Limit : V \to \mathbb{N}$. Thus, the internal state represents the number of data rows present in body of the loop managed by this condition.

Since we define semantics of a single data row, this remark may be effectively left out since this condition always holds in an environment containing just one data row (which is how all semantics are defined).

[8]This is meant with respect to the definition of realisation on a single data row. In real implementation this means that we have to flush the entire loop *somehow* before checking this condition.

explicit queue/explicit buffer *is a queue which takes a single element from its only input and pushes it into all outputs. Explicit buffer nodes will help us formalising placement of real buffers.*

Using the standard *split* and *merge* nodes, we will be able to perform standard branching, i.e., to realise the `if {} else {}` construct. The *loop* node construct will allow us to realise the `while(condition) {}` construct. We formalise this in Definition 17.

Remark. Until now, we have used only rectangular nodes in our graphs. From now on we will also use circular/elliptical nodes which will denote connected parts of graphs.

Definition 17 (Schemes of node usage)**.** *We shall define the following three schemes of node usage by Figure 2.10.*

- branching *scheme*
- loop merge *scheme*
- loop condition *scheme*

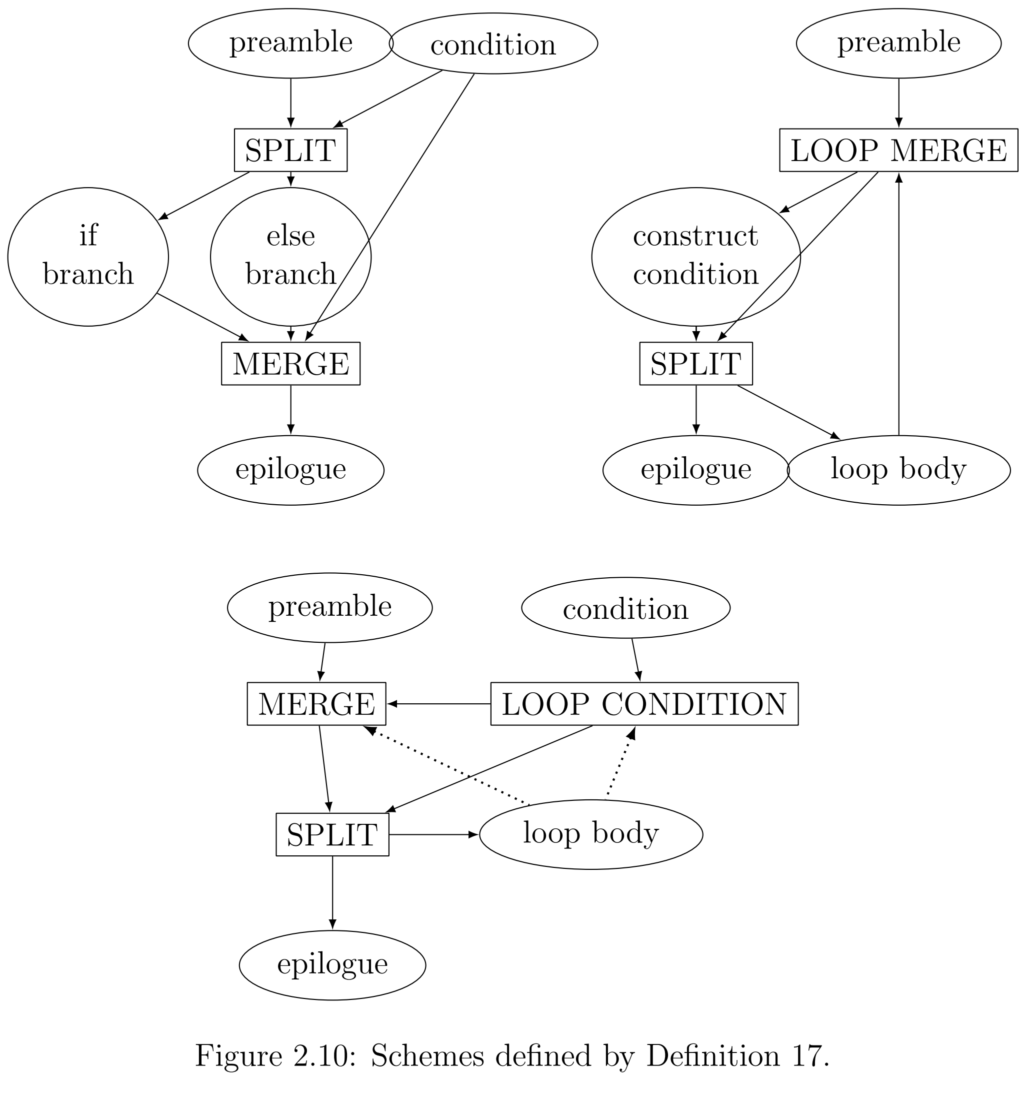


Figure 2.10: Schemes defined by Definition 17.

Remark. Note that although we have specified some possible use of these nodes, we do not (yet) restrict flow graphs to these schemes.

Until now, we have performed all data transformations *regularly*. Thanks to the fact, we did not have to check the consistency of flow graphs (defined in Definition 12 by properties of realisation of a flow graph on a single data row). Apparently, not all graphs satisfy the conditions defined by the definition of consistency. Figure 2.11 shows one such graph. The ADD operation will never be able to process anything since it never gets both operands due to the merge operation. Or we may understand this graph in such way that elements of two different data rows meet if we try to process multiple data rows at the same time.

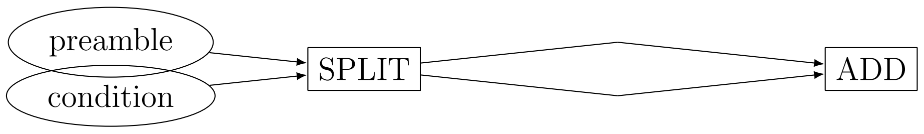


Figure 2.11: An example of a graph which is inconsistent due to improper use of the *split* node.

We will not present any way of testing of the consistency of general flow graphs. We will content ourselves with an observation that nontrivial consistent flow graphs exist.

Observation 1 (Consistency of schemes)**.** *A flow graph G in form of one of the examples of node type usage is consistent if the following conditions hold:*

- *Branch or loop bodies, conditions, preamble and epilogue consist only of regular operations.*
- *There are no concealed paths in between:*
  - *if branch and else branch*
  - *any branch and a preamble/condition/epilogue*
  - *loop body/condition and a preamble/epilogue*
- *There are no input and output operations contained in loop condition or in loop and branch bodies.*
- *Semantics of O are preserved.*

Proof. Thus, we need to show the following: provided that every input node produces one value, the GRA algorithm produces exactly one value at every output node and that no value remains in the graph (except for queues of output nodes).

Assume a node remains with some input empty and some input nonempty.

- All partitions depicted as circular/elliptical nodes consist of regular operations only, hence are acyclic. It holds that either all inputs of such partition receive exactly one input or all receive none. Furthermore, it holds, a path exists from an input (i.e., either from an input queue of the partition in question or from an input node) into any queue of any node in such partition. It follows that every operation in such partition is invoked exactly once per every data row, having always exactly one value in all its input, consuming these and producing exactly one value per every output. Thus, all values (except for the ones in queues of output operations) are consumed, so the assumed situation cannot take place inside of a single partition.

- We have banned any direct contact between any two partitions which may cause this, except for the pair of preamble and epilogue.

- Flow of data from every split leads to a merge which merges according to the same condition, which means that the two data flows are merged in such manner that the quantity of the data rows is preserved (and their order too, if there were multiple data rows present in the graph). Thus, the branching scheme is correct.

- Apparently, epilogue in the loop merge scheme receives exactly one value unless there is an infinite loop in semantics of O, which is a situation which we do not assume in this text.

- The loop condition operation in the last scheme apparently manages the split/merge nodes in consistent manner, so the epilogue of this scheme again receives exactly one value.

□

Observation 2 (Scheme nesting)**.** *Let $G(V, E)$ with O be a consistent flow graph generated by finite repetition of steps described in observations 1 to 3. Let C be a connected subgraph of G composed only of regular nodes, i.e., of vertices annotated by regular operations. Then, create a new graph by nesting a graph H corresponding to either of the provided example schemes into C. Do so by means of connecting of vertices of C and H by edges and by changing some regular operations into other regular operations. Then the new graph is consistent, presuming that the following conditions are satisfied:*

- *The nested scheme satisfies the conditions from Observation 1.*

- *The nested scheme is connected to C only by its preamble and epilogue.*

- *The nested scheme does not contain any input operations.*

- *We have not introduced any new cycle except for cycles containing second input of any loop node.*

- *Semantics of O are preserved.*

Proof. This follows directly from the proof of the previous observation and the structure of G. □

Observation 3 (Scheme sequencing)**.** *Let $G(V, E)$, O and C be defined as in the previous observation. Now we take one of the example schemes and embed it multiple times into C by means of Observation 2. If we use the same condition for all inserted instances of the chosen scheme, then we may join the corresponding bodies of the embedded schemes without voiding consistency of the new graph.*

Proof. By structure of G using an argument analogical to the proof of the *consistency of schemes.* □

The provided schemes apparently suffice for realisation of the standard `if {} else {}` and `while(condition) {}` constructs. However, we will still try to provide solutions for general (consistent) flow graphs. The following example shows a consistent flow graph which cannot be generated by means of observations 1-3. Notice that consistent input streams for the merge operations can be constructed using *(pure)* regular operations. It suffices to compute the merge condition before we apply any split to it.

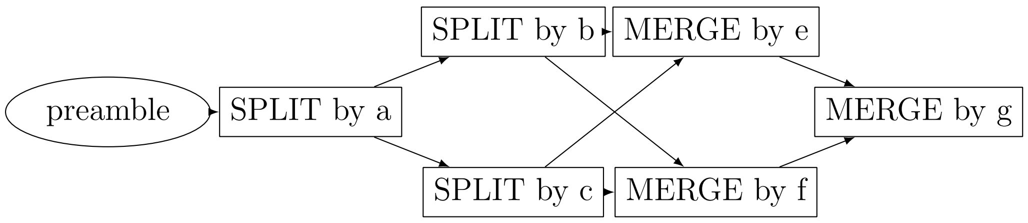


Figure 2.12: Control flow example which is consistent despite its inconsistent look.

a, b, c are bools known in preamble.

e, f are streams of the predicate `(a)` split by the predicate `((a&&c)||(!a&&b))`.

g is a stream of the predicate `((a&&c)||(!a&&b))`.

2.3.1 Graph partitioning

In the previous sections, we have shown how flow graphs composed of regular operations may be processed. This approach does not suffice in graphs containing control flow since different nodes may need to process data at different speeds.

Our strategy is to cut a flow graph into partitions such that every partition consists only of regular operations. These partitions are then connected by buffers. Finally, the data are processed by a crawler which goes through our graph and processes its partitions one at a time. Every single partition is processed by a fragment of code generated by the generators we have already presented.

First, we need some definitions:

Definition 18 (Edge layer)**.** *Let there be a directed graph (possibly multigraph) $G(V, E)$. We will assign one more numeric annotation to every edge. We will call this annotation* layer*. Thus, we have $Layer : E \to \mathbb{N}_0$. If $Layer(e) = n$ we will say that e is on layer n.*

Layer will denote a type of an edge. It will allow us to talk about subsets of edges in a more intuitive manner and also to introduce more complex graph structure. We show our visual convention in Figure 2.13.

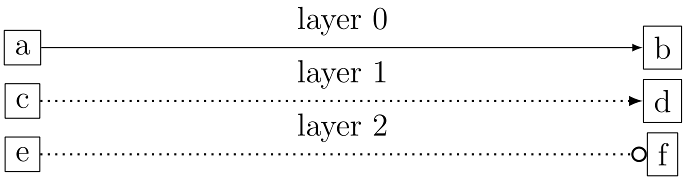


Figure 2.13: Our convention of marking edge layers

Definition 19 (Factor graph)**.** *Let $G(V, E)$ be a directed multigraph and $G_f(V_f \subseteq \wp(V), E_f)$ be a directed graph. We will say that G_f is a* factor graph *of G if there exists a function $f : V \to V_f$ such that:*

- $(\forall u \in V)(\forall c \in V_f)(u \in c \iff c = f(u))$
- *$f(u) = f(v)$ if and only if there exists an undirected path $P_{u,v}$ in G which uses only edges on layer 0*
- $(\exists e \in E)(\exists u, v \in V)(e \sim (u, v) \wedge layer(e) = 1) \iff \exists(e_f \in E_f)(e_f \sim (f(u), f(v)))$
- *The edge e_f in the previous condition is unique.*

We will call vertices of G_f components *since they are connected components in a graph defined by $(V, \{e \in E \mid layer(e) = 0\})$. We will also use the term* component *for partitions of G which correspond to a vertex in the factor graph. I.e., we will call a partition P of G a component if and only if there $(\exists w \in V_f)$ s.t. $(x \in P) \iff (f(x) = w)$ for every $x \in V$.*

Remark. We should warn the reader that due to this definition the terms *component*, *partition* and *class (of equivalence relation)* cohere.

Definition 20 (Input and output vertex of a component)**.** *Let C be a component of a graph $G(V, E)$. We will use the term* input vertex of a component *for a vertex $v \in C$ s.t. $(\exists w \in V)(\exists e \in E)(e \sim (w, v) \wedge Layer(e) \geq 1)$. We will also use the term* output vertex of a component *defined analogically.*

Note that this definition overrides another meaning of this term. Also, note that a factor graph exists for *every* graph and that it is unique (up to isomorphism). This is true since factor graphs are determined by components of connectedness.

Our partitioning consists of the following steps:

1. Control flow node expansion.
2. Removal of cycles in factor graph.

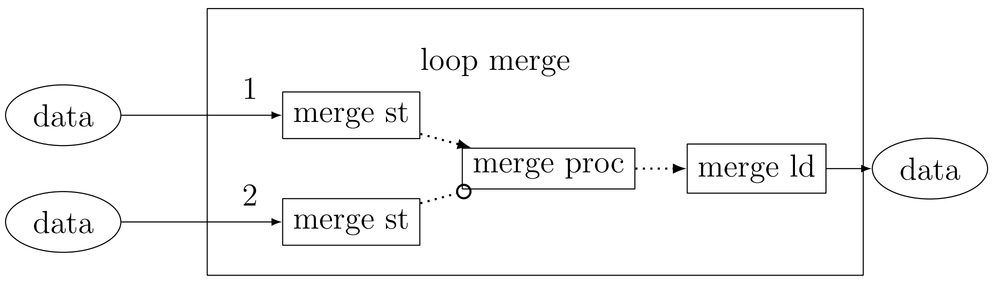
Cycles may be introduced into factor graphs by node expansion, even into factor graphs of acyclic flow graphs. Any path in a factor graph represents a path of buffers where data wait for processing. Thus, a cycle in a factor graph means that some computation path (let us say P) containing some control flow nodes leaves a component (let us say C) and later joins it. This prevents the partition from being processed by a single basic block of code. Therefore, we need to remove all cycles from the factor graph. We do so by splitting the component C into two new components C_1 and C_2 such that $C_1 \leq P \leq C_2$.

In other words, we need to achieve a factor graph with a topological ordering in its semantical meaning. That is, we need to achieve a situation in which partition may be processed in some order and we need this order to be realisable. We do not prohibit cycles entirely, since a cycle whose first iteration does not depend on data from cyclical edges does not create a dependency. If we wished to be overly formal, we could introduce a construct which would understand every node as a set of its instances. These instances would be induced by every iteration of this algorithm. Apparently, every graph created by such construction is acyclic and thus has a topological ordering in its standard, formal meaning.

Node expansion

In this step, we replace all control flow nodes by new vertices which are connected by layer 1 and 2 edges. By this, we insert splits at places where data need to be reordered (where the corresponding streams need to be either split or merged). We formalise this by Definition 21.

Definition 21 (Control flow node expansions)**.** *We shall define expansions of control flow nodes introduced in Definition 16 by Figure 2.14. I.e., we define node expansion as a process which substitutes nodes in a graph by subgraphs defined by Figure 2.14. Edges incident to an expand node are preserved during this process.*



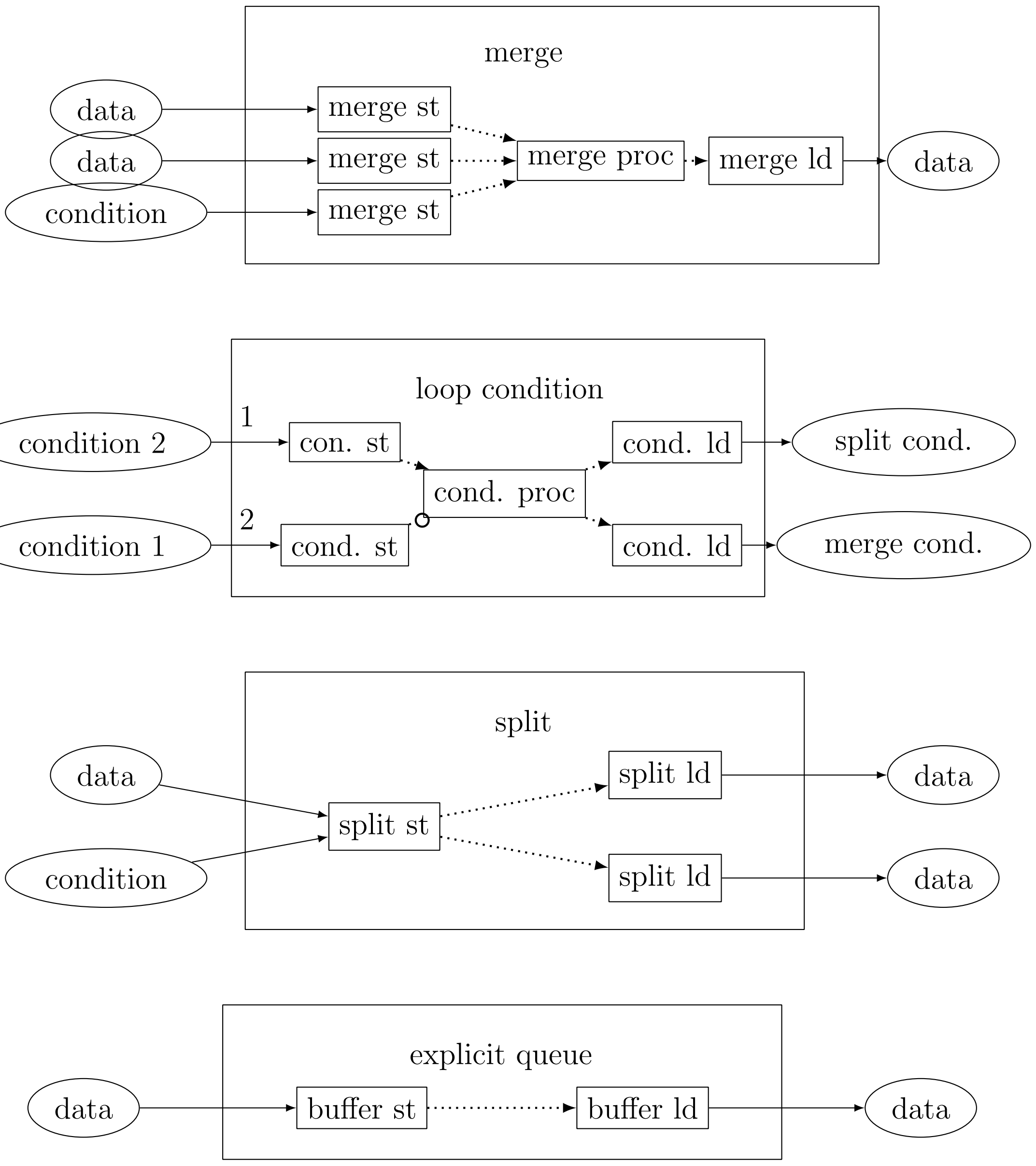


Figure 2.14: Diagrams defining node expansion (Definition 21).

Cycle removal

Let $G(V, E)$ be a flow graph. In order to process data, we need the factor graph H of G to be acyclic. Unfortunately, H may contain cycles and loops in spite of G (restricted to layer 0 and 1 edges) being acyclic.

We propose Algorithm 5 to split components so that cycles get removed.

Ignoring some details, this algorithm proceeds as follows:

1. Take a queue which is on any cycle.
2. Colour its predecessors green.

3. Colour its successors red.
4. Colour other vertices somehow.
5. Cut the problematic component along the green-red borderline.
6. Repeat.

Algorithm 5 (Cycle removal algorithm)**.**

```
  H = G
while ( cycle in H exists )
{ //loop i
  colour vertices of G black
  C = any component on cycle in a factor graph of H
  for v in all outputs of C
  { //loop j
    for w s.t. there a is path from v to w
    (using edges on layers 0 and 1)
    {
      colour w red;
    }

    colour v and all non-red inputs of C green;
    for w non-red s.t. there is a path from w to a green vertex
    (using edges on layers 0 and 1)
    {
      colour w green;
    }

    for w non red s.t. there is an undirected path from w to v
    (using edges on layers 0 not containing red vertices)
    { //loop k
      colour w green;
    }

    bool changed = false;
    for edge (w,x) s.t. w is green and x is red
    {
      put and expand an explicit queue on (w,x) in H;
      changed = true;
    }
    if(changed)
      break;
  }
}
```

Claim 1 (Functionality of the cycle removal algorithm)**.** *Let G be a flow graph consisting of node types defined so far with control flow nodes expanded. Perform Algorithm 5 on G. We claim that:*

1. *The algorithm terminates.*

2. *The algorithm yields a graph H whose factor graph contains neither loops nor cycles.*

Proof. Let K be a cycle or a loop in a factor graph of G and let $C \in K$ (C from algorithm).

- There exists an oriented path P on layers 0 and 1 from some a to some b s.t. $a \neq b$ are two distinct vertices in the same component. P exists due to the existence of K. Non-equal a and b exist because G (restricted to layers 0 and 1) is acyclic and so P cannot lead from a to a.
- If we choose $v \in P$ s.t. $v \neq b$ (v from algorithm), then b will be coloured red. Otherwise a will be chosen as v in another iteration since a and b are in the same component. Thus, b will be coloured red in some iteration of loop j. Moreover, a will be coloured green in the same iteration since there is no path from b to a.
- Since a and b are in the same component, every path from a to b will be disconnected. Thus, the algorithm works if it terminates.
- Putting a queue on an edge does not introduce any new path on level 0 between vertices of G in H. It also decreases the number of paths on level 0 in H between vertices of G. There are finitely many such paths in G. Thus, the algorithm will terminate.

□

Remark. The algorithm will not terminate if there is a cycle composed of 0 and 1 layer edges. But we have defined node expansion so that all backwards edges are on layer 2.

This algorithm is not optimal. The loop marked as `k` may be replaced by a minimum cut algorithm, by which the size of the first cut may be minimised. Unfortunately, there is no guarantee that such optimisation does not increase the minimum costs of cuts performed later. If we wanted an optimal solution, all cuts would have to be optimised at the same time.

As an example of the entire partitioning process, we show a simple computation (Figure 2.15) which performs some arithmetic operations. The reader will surely excuse empty branch bodies (direct edges from the split node to merge the merge node). We begin by a simple representation of a graph which contains split and merge control flow nodes and one explicitly inserted queue (the *buffer* node).

Node expansion produces a new graph (Figure 2.16) with some level 1 edges. This graph already consists of four components, but there are some edges which bypass the control flow. These edges introduce a data dependency in the partition 0 which cannot be met without leaving the code dedicated to processing of the partition 0. This problem may be seen in the included factor graph, which contains an edge going from component 3 to component 0. There is also a loop on partition 0 caused by the edges which bypass the buffer node.

Finally, the cycle removal inserts the *auto buffer* nodes, producing a graph (Figure 2.17) with neither cycles nor loops in its factor graph.

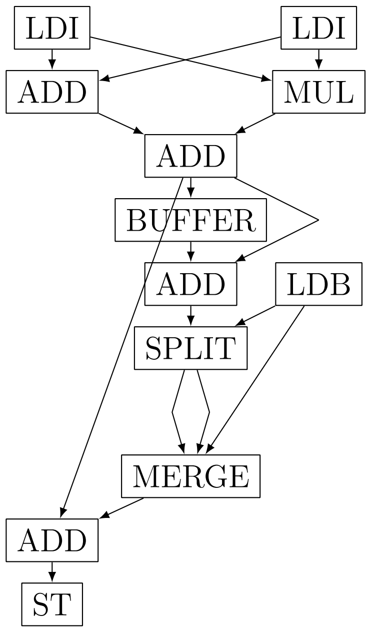


Figure 2.15: An example of control flow prior to any transformation.

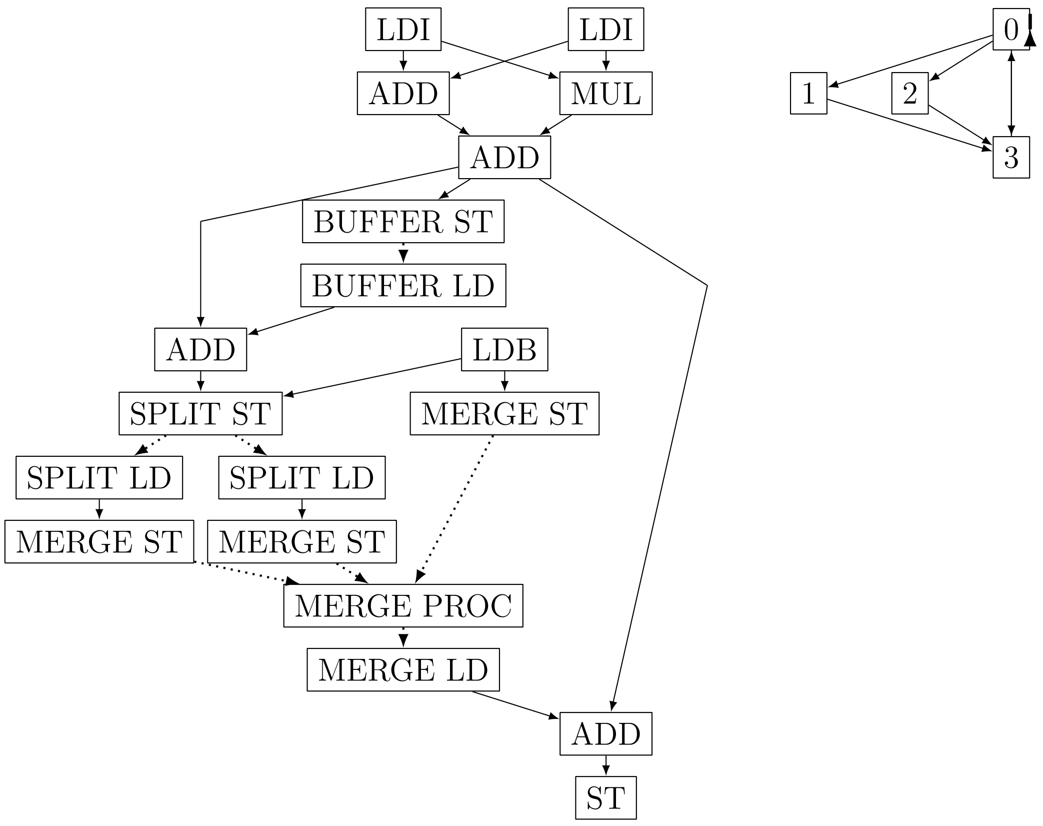


Figure 2.16: The graph from Figure 2.15 after node expansion.

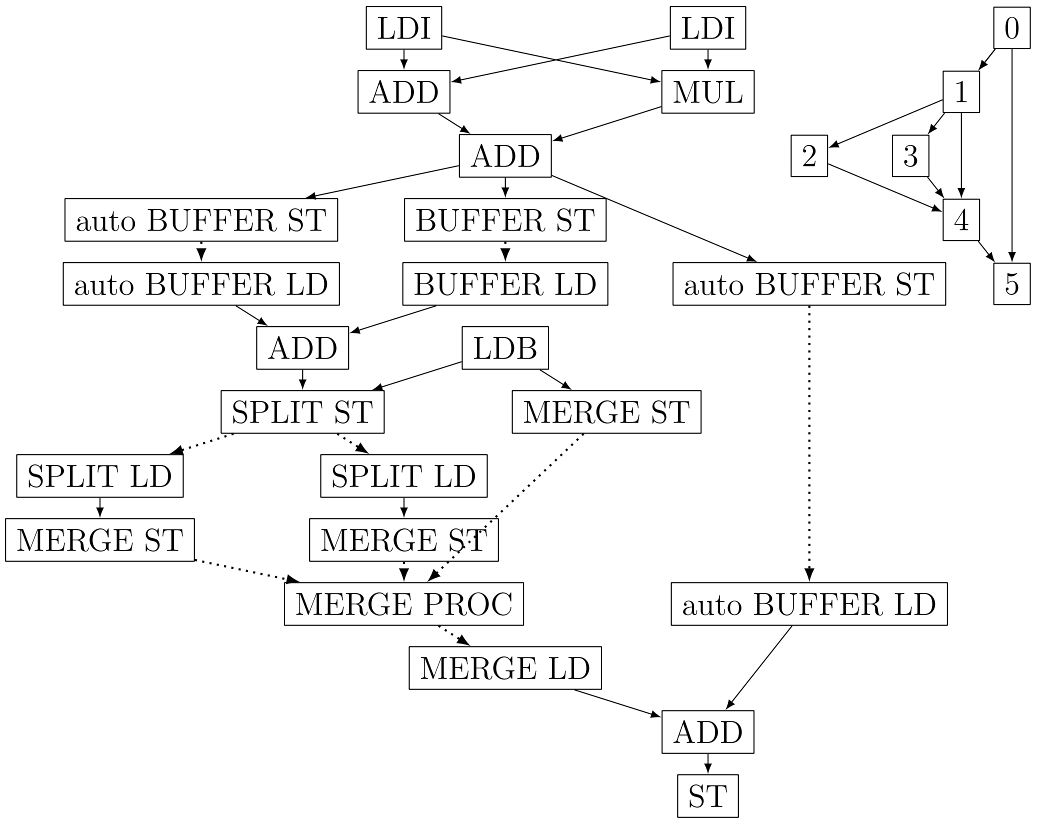


Figure 2.17: The graph from Figure 2.16 after cycle removal.

2.3.2 Order-preserving evaluation

Consider the following problem:

Problem 3 (Ordered realisation of control flow)**.** *Let $G(V, E)$ be a consistent flow graph with O and I defined as in the* vectorised code generation. *Let G consist only of regular nodes (including input and output nodes), splits and merges. Also, let control flow nodes of G be expanded according to the previous section and let the factor graph of G be acyclic. We wish to generate a code fragment which realises the graph G on an arbitrary number of data rows, utilising SIMD instructions whenever possible. We also wish the order of data rows to be kept in the output.*

Remark. Note that by node expansion, we have obtained a graph which contains *only* regular operations. Also, note that the graph is acyclic since we did not include any node type representing loops in the problem definition. We add loops into this algorithm later.

We already know that components of G can be processed as basic blocks. We propose components of G to be connected by constant-sized register-mapped buffers in place of layer 1 and 2 edges. The reason for our decision to make buffers register-mapped is that we wish to minimise the amount of transfers between CPU and RAM. The reason to make our buffers constant-sized is apparently the fact that register space is limited. We again choose w as the number of data rows processed in one invocation of a vectorised version of a partition. We also wish to preserve the order of data rows in all merges in this proposal.

To illustrate the problem, we present the following algorithm.

Algorithm 6 (Naive crawling algorithm)**.**

```
enter the first component,
  effectively pushing w data records into the pipeline.
DFS( component C in factor of G )
{
  if (C has >= w records in every input)
    while (C has >= w records in every input)
      process C;
  else
    abandon current branch;
}
```

There are two problems:

- If we wanted to process all data available at a partition in one go, the size of buffers would have to increase exponentially with every merge, since every merge may have both its buffers nearly full and therefore output $2 * w$ records.
- Since we wish merges to preserve the order of all data, there may arise a situation in which a single element prevents merge. If we waited with the processing of this branch until there were w input data rows available, the other branch may overfill its buffers.

As a solution, we propose Algorithm 7. This algorithm addresses the first problem by pulling new data on demand. The second problem is solved by introduction of pull semantics which causes a branch to process less than w records if needed. We generate code similar to the following diagram for every component i.

Algorithm 7 (Ordered crawler algorithm)**.** *Let G, O, I be defined as in Problem 3. Let w be a positive integer. Generate code corresponding to the following diagram for every component of G and do so in topological order.*

partition 0 *For simplicity, we assume that all input sources are placed in partition 0. This is w.l.o.g. since we may use a separate counter for each partition.*

`terminating` **variable** *When we exhaust all input, we set this variable to zero to indicate that the first partition should process all its remaining data and terminate. Upon termination of any component, this variable is incremented and the process repeated with the next partition.*

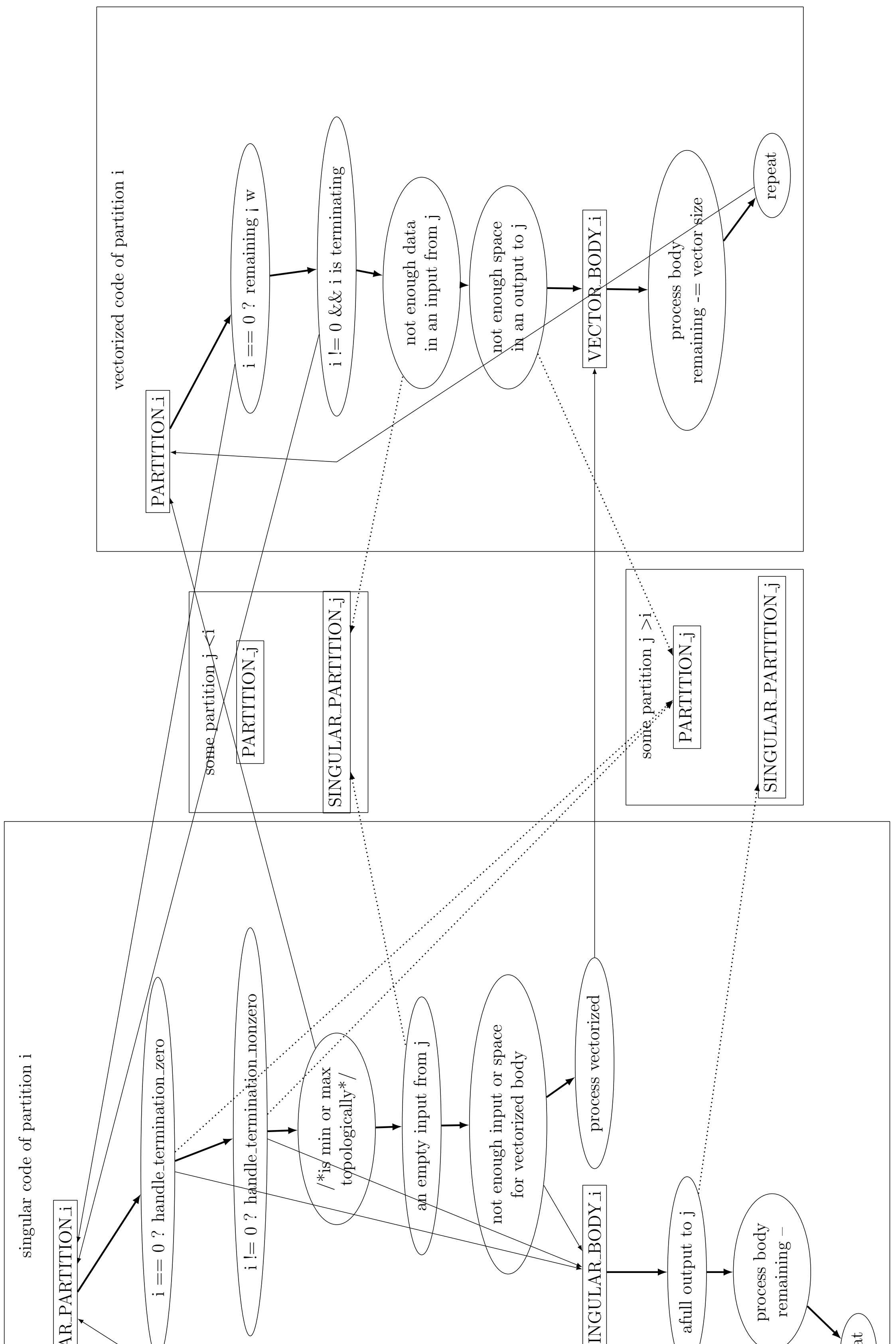

vectorized code of partition i
PARTITION_i
i == 0 ? remaining j w
i != 0 && i is terminating
not enough data in an input from j
not enough space in an output to j
VECTOR_BODY_i
process body remaining -= vector size
repeat
some partition j <i
PARTITION_j
SINGULAR_PARTITION_j
some partition j >i
PARTITION_j
SINGULAR_PARTITION_j
singular code of partition i
SINGULAR_PARTITION_i
i == 0 ? handle_termination_zero
i != 0 ? handle_termination_nonzero
/*is min or max topologically*/
an empty input from j
not enough input or space for vectorized body
process vectorized
SINGULAR_BODY_i
afull output to j
process body remaining –
repeat

Figure 2.18: Algorithm 7.

`remaining` **variable** – *We use this variable to indicate the count of data rows which are yet to be retrieved from all data sources.*

bold line *denotes implicit continuation, i.e., the next operation in case that related condition does not hold.*

solid line *denotes the next operation in case that related condition holds.*

dotted line *denotes jumps to different partitions (in case that related condition holds).*

rectangle *denotes a goto label.*

empty inputs – *When we say* empty input, *we mean an input which is supposed to provide data. I.e., on split, we do not jump into a branch which is not supposed to provide input.*

space in buffers – *When we say enough space in outputs (in the context of vectorised body), we mean a requirement that the capacities of all output buffers of the component in question allow insertion of at least w elements. When we say enough data (in the context of vectorised body), we mean a requirement that every input buffer of the component in question contains at least w elements. When we specify input from/to* j, *we mean that j denotes the partition to/from which the unsatisfied buffer leads, i.e., we are quantifying over* j.

i *is an index of the current partition. These indices are topologically ordered.*

j *is an index of any other partition depending on context.*

handle termination zero

```
if( i == 0 && remaining < w )
{
  terminating = 0;
  if(remaining == 0)
  {
    terminating = 1;
    goto PARTITION_1;
  }
  goto SINGULAR_BODY_0;
}
```

handle termination nonzero

```
if ( i == terminating )
{
  if(an input of i is exhausted)
  {
    terminating++;
    goto PARTITION_i+1;
  }
  goto SINGULAR_PARTITION_i;
}
```

Claim 2 (Functionality of the order-preserving solution)**.** *We claim that Algorithm 7 solves Problem 3 under the stated conditions if all buffers have positive capacity. That is:*

1. *Algorithm produces correct results (if it terminates).*
2. *Algorithm does not enter an infinite loop during the pre-termination phase unless there is an infinite loop contained in the semantics of G with respect to the data processed.*
3. *Algorithm terminates correctly.*

Proof of 1. Our algorithm never retrieves data from an empty buffer and never pushes data into a full buffer. Also, we require G to be consistent with the semantics of the defined control flow node types.

Consistency of G ensures that all data of any single data row will get processed without leaving unprocessable elements in buffers. This means that as long as the elements of all data rows pass through all operations in order, there is no way how two elements from different data rows could get processed as if they were from the same data row. Thus, all results are correct. $\square$

Proof of 2. Let there be a flow graph $G(V, E)$ and a sequence of data rows such that the described algorithm enters an infinite loop which does not process any data. Let C be this loop (a subgraph of a factor graph H of G)[9]. Let t be a topological ordering of H. Also, let there be a description of contents of buffers at the moment when algorithm iterates over C.

- Apparently, the algorithm iterates only over singular code of partitions. This means that every partition in C has either entirely full output or entirely empty input. Also, no partition is contained in C more than once.
- Let $M = \{v \in C \mid \neg(\exists u \in C)(u <_t v \wedge (u, v) \in C)\}$ (informally all minimal vertices in C with respect to topological order of components restricted to edges in C)(shown in Figure 2.19. Let n be the largest index of a data row processed by all components. Such n does exist since every component in M has a full output queue (otherwise our algorithm would not continue downwards).
- Let $a \in C$ be any partition in which the nth data row is the last processed one. Then there is an output buffer of a which contains data from the nth row. Let $e \in C$ be an edge which corresponds to this buffer. Let $b, c \in C$ such that paths $P_{ab}, P_{cb} \in C \wedge e \in P_{ab} \wedge c \in M$ axist.
- Let m be the highest index such that an element from the mth row is still present on the path P_{ab}.

[9]This is a directed acyclic graph with all degrees equal to two, not a directed cycle!

- Partitions a, c have processed the mth row since $a, c \in M$ and $m \geq n$. Partition b has not processed the mth row since an element from the mth row is still on the path P_{ab}. That means that there is an element from the mth row on the path P_{cb}. (This holds since our algorithm never chooses a branch in which no data from the required data row are to be found, and since the required data row is exactly the mth one. That is also the reason why we do not use the nth row.) That is a contradiction because one of the two paths is empty.

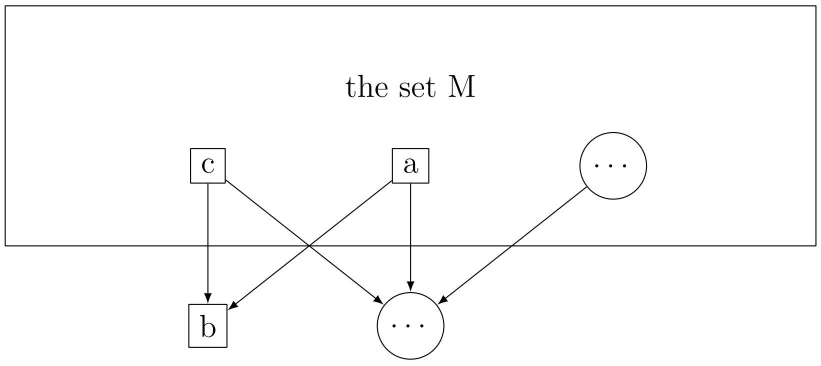


Figure 2.19: An illustration showing cycle C and set M from the second point of proof of Claim 2.

$\square$

Proof of 3. It suffices to check that the algorithm will process all data of the terminating component and increment the *terminating* variable. $\square$

Thus, we have shown that general consistent flow graphs consisting of regular nodes, merges and splits may be realised by this approach.

Loops

It is easy to see that the proposed *loop merge* operation is well defined and that it behaves as expected in the definitorical sense given in *realisation of a flow graph on a single data row* (Definition 10). It may seem that the previous algorithm still provides correct results on a graph with *loop merge* nodes, since the second point of the proof still holds. (The point which fails is the first one.) The real problem comes with data ordering as the following ideas illustrate:

1. Moving data over a backwards edge may result in insertion of these data after data from a data row with higher index.
2. The point 1 does interfere with the first point of the proof (of Claim 2) since the merge operation may have data from different data rows in its inputs and since it ignores their order.

3. We may decide to let multiple data rows enter the loop and then postpone exit of every row with non-minimal index (with respect to being in the loop in question). We would then pull the minimal-index row through the loop using the non-vectorised pull-semantics[10] until it exited the loop. This does not work due to point 1. We would have to implement multiple 'waiting' buffers at every *loop merge* and we would still have to pull a significant amount of data by the non-vectorised bodies. Moreover, efficiency of this approach would decrease with increasing w due to increasing probability that a non-minimal row will need to exit the loop prior to the minimal row.

4. We may improve semantics of the *loop merge* node so that elements of data rows remain synchronised inside of loops. Then, we would add a 'waiting' buffer at the loop exit. This buffer would reorder data rows back into the initial order. This does not work since the row which is waited for may remain in the loop for a very long time. (We would need to use unbounded queues instead of constant-size buffers.)

5. We may employ the previous point the other way around. We may make all other data wait for data coming from loops. This approach does work except for the problem that the rest of the network is performed in non-initial order. This is approximately the way the second proposal copes with the problem. We cannot use it here since we required data to be processed and outputted in order. For this reason, there is no algorithm which at the same time processes arbitrary loops in fully vectorised manner, outputs data in-order and uses only unbounded queues.

6. We may ensure that at most one data row enters the loop at the same time. This makes sense since the theoretical single data row apparently works.

The approach suggested by 6. and 3. — point 3 is basically a more complicated version of point 6 — may be implemented and would work in a *single row realisation* environment on general flow graphs consisting only of regular, split, merge and loop merge nodes. Figure 2.20 shows that Algorithm 7 does not work with flow graphs with arbitrarily placed *loop merge* nodes. The code generated by Algorithm 7 for the following example may change the order of some data rows, despite consistency of the flow graph.

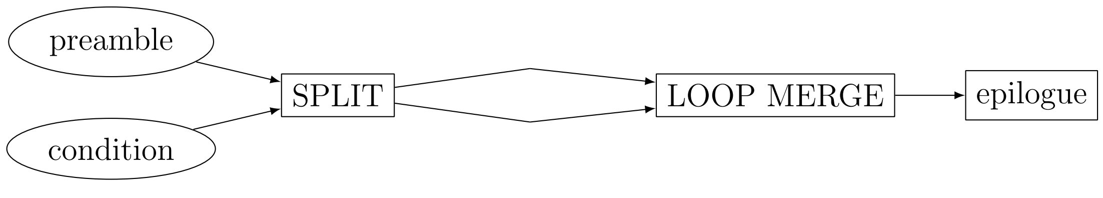


Figure 2.20: A counter-example showing that Algorithm 7 does not work for graphs containing arbitrarily placed *loop merge* nodes.

[10]By pull semantics we mean the mechanism which allows processing of data in non-vectorised manner

Restricting Algorithm 7 with allowed *loop merge* nodes to flow graphs generated by means of Observations 1 – 3 solves the problem if we ensure that data remain in order.

Definition 22 (Implementation of the loop merge operation)**.** *Let every loop merge operation have a Boolean flag (`state`) as its internal state, initialised to false. Modify Algorithm 7 to set this flag to `true` every time any partition processes data*[11]*. Furthermore, modify the same algorithm to use only the singular body of the loop merge node (this is no problem since loop merge node is always the only one in a partition). Let the code of the partition of the loop merge node be defined by the code shown in Figure 2.21.*

```
SINGULAR_PARTITION_i:
PARTITION_i:
if(state)
{
  state = false;
  //k denotes partition of the second input
  goto SINGULAR_PARTITION_k;
}
if(output to l is full)
  goto SINGULAR_PARTITION_j;

if(second input is nonempty)
{
  process the second input;
}
else
{
  if(input from j is empty)
  {
    if(terminating == i)
    {
      terminating ++;
      goto SINGULAR_PARTITION_i+1;
    }
    //j denotes partition of the first input
    goto SINGULAR_PARTITION_j;
  }
  process the first input;
}
repeat
```

Figure 2.21: Body of a loop merge partition for Definition 22.

Claim 3 (First solution with loops)**.** *We claim that Algorithm 7 implementing* loop merge *nodes as in Definition 22 solves Problem 3 with allowed* loop merge

[11]Since we need just 1 bit of information per every such node, it is possible to put all these flags into one integer[12]so that this operation remains cheap.

[12]Provided that there is not more of these nodes than integer's length.

nodes if G was created by means of composition of the branching *and* loop merge *scheme (Definition 17) by means of Observations 1 – 3.*

Proof. We need to check that the three points of the proof of Claim 2 still hold.

1. The `state` ensures that as long as there is a data row present in the loop, no new row is retrieved from the first input. Data obtained from the second input must have been earlier processed by the first input. Thus, data remain in order and valid.
2. Note that if no data had been processed second time a *loop merge* partition was entered, the algorithm continues to the partition of the first input. This means that the assumption from the original proof that $C \subseteq H$ holds. Furthermore, it means that either:
    - some data were processed. In this case, a data-processing infinite loop is a correct behaviour; or
    - no data were processed. In this case, the cycle does not use the second input. Thus, the original proof holds.
3. This partition apparently terminates correctly (unless there is an infinite loop in the semantics of O with respect to the data which are processed).

$\square$

Analysis

We provide some observations relating to efficiency of the provided solution.

Observation 4. *Let there be a computation (i.e. a realisation of a consistent flow graph). Let therein be two parallel branches leading from a common predecessor partition to a common successor partition such that both branches receive data from the same data rows (i.e., the data are not being split in the root node). Let the sum of buffer capacities of one of the branches be significantly higher than the corresponding sum on the other branch. This means that one branch has significantly higher average density of data per buffer. Note that now the output buffer of the branch with smaller capacity is likely to cause the content of the other branch to be processed. It causes the content to be processed even though majority of buffers on this branch does not provide enough data for vectorised processing.*

We believe that the reader has some intuitive understanding of this observation. We do not provide exact definitions because we do not draw any exact conclusions and because these definitions would be truly tedious.

We propose assigning buffer capacities according to the following algorithm. We do so even though we have no proof of feasibility which would have realistic assumptions about input data.

Algorithm 8 (Buffer capacity calculator). *Let H be an acyclic factor graph of a flow graph G with expanded control flow nodes. Let w be (again) a chosen size of vectors and let $c \in \mathbb{N}$ be an arbitrary coefficient.*

```
fun l(partition d) { an integer annotation of components of H}

fun balance_sizes(H, w, c)
{
  foreach(d in H)
    l(d) = 0;
  while( exists edge (a,b) in H s.t. l(a) + 1 > l(b))
    l(b) = l(a)+1;
  foreach(buffer (from a to b) in H)
  {
    size(buffer) = (l(b) - l(a))*w*c;
  }
}
```

Observation 5 (vectorised processing of data sources and sinks). *Suppose that all output buffers of the topologically minimal partitions and all input buffers of the topologically maximal partitions (in a factor graph of G w.r.t. connectedness) have size at least w. We claim that if we uncomment the commented node, i.e., if we force vectorised processing of those partitions, then the algorithm still works.*

Proof. vectorised and singular processing are equivalent in these partitions as long as buffers have sufficient capacity. This holds since these partitions will never be required to forward data from a predecessor to a successor. □

Remark. Note, that if the buffer capacity is w, forcing of vectorised processing of partitions may cause their data providers to process data singularly as well. For this reason, we propose that the coefficient `c` from Algorithm 8 should be at least 2.

Remark. Although this observation may not seem interesting for theoretical analysis, it is vital for efficient implementation of vectorised load and store operations. (This is due to typical alignment requirements of fast load and store operations.)

2.3.3 Out-of-order evaluation

Since the cause of ineffectiveness of the previous proposal is the requirement stating that data should be processed in order, we will now investigate the same problem without this condition.

If we simply dropped the condition, i.e., by letting merges merge in any order, another problem would arise — the data may get reordered differently in different branches. This is shown in Figure 2.22.

This problem may be addressed by ensuring that data rows are always handled as one unit, meaning that every split will be performed on all data except data that are not used any further.

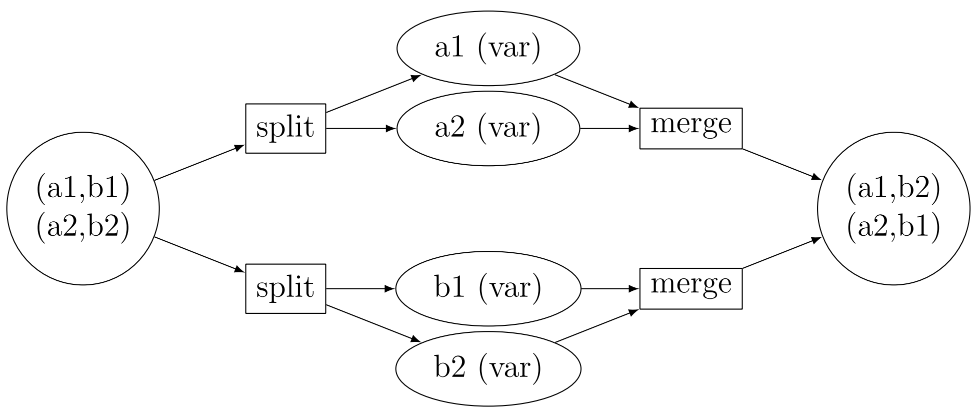


Figure 2.22: Computation which mixes data of different data rows together.

Definition 23 (Sequential form of control flow). *We will say that a control flow of a flow graph $G(V, E)$ with control-flow nodes expanded (Definition 21) and cycles removed (Algorithm 5) is in* sequential form *if the following conditions hold:*

- *All inputs of G are in the same component.*
- *At least one of the following conditions holds for every component C of G:*
  - *C does not have any input from another component of G.*
  - *All inputs of C are unordered merge nodes such that corresponding inputs of any two input merge nodes of C lead into the same partition of G.*
  - *All inputs of C are standard merge nodes such that corresponding inputs of any two input merge nodes of C lead into the same partition of G and the at the same time all these merges use the same condition.*
  - *All inputs of C are explicit queues (all) leading from another component D of G.*
- *At least one of the following conditions holds for every component C of G:*
  - *C does not have any output going into another component of G.*
  - *All outputs of C are split nodes such that corresponding outputs of any two output split nodes of C lead into the same partition of G and at the same time all these splits use the same condition.*
  - *All outputs of C are explicit queues (all) leading into another component D of G.*

Definition 24 (Unordered merge).

Unordered merge operation *will have the same semantical meaning as the standard merge or loop merge operation except it will not take any condition input. This operation may merge streams in an arbitrary but deterministic manner.*

It is not clear whether every consistent flow graph can be transformed into a flow graph with control flow in sequential form. What is clear is that there exists a nontrivial class of flow graphs which are in this form.

Observation 6 (Consistency of schemes with control flow in sequential form)**.** *Let $G(V, E)$ be a flow graph s.t. the following conditions hold.*

- *G is consistent according to the* consistency of schemes *(Observation 1).*
- *G uses only unordered versions of merge nodes in place of standard merge or loop merge nodes.*
- *All input nodes of G are in the preamble.*

Then G may be transformed into a consistent flow graph with control flow in sequential form. This may be achieved by copying the scheme in question onto every path between preamble *and* epilogue *and joining all newly created branch or loop bodies by means of the* scheme sequencing *observation. We may join these by adding any regular operation which does not alter the semantics of G.*

Proof. Sequentiality of the resulting control flow follows directly from definitions. □

Observation 7 (Unordered scheme nesting and sequencing)**.** *Both* scheme nesting *(Observation 2) and* scheme sequencing *(Observation 3) will produce graphs with control flow in sequential form as long as:*

- *All input graphs have control flow in sequential form and all paths from* preambles *to* epilogues *are transformed in the means of the previous observation.*
- *The condition requiring all inputs to be in the same component (w.r.t. regularity and connectedness) is not voided.*

Proof. The key observation is that the previous transformation necessarily splits *preamble* from *epilogue*[13] (with respect to regularity of nodes). Thus, nesting of a new scheme into a component will never create two parallel schemes except for schemes which comply with the *scheme sequencing* observation (Observation 3. □

Consider the following problem:

[13]As shown in Definition 17 of schemes.

Problem 4 (Unordered realisation of control flow)**.** *Let $G(V, E)$ be a consistent flow graph with O and I defined as in the* vectorised code generation *problem (Problem 2). Let G use only the following types of nodes: regular nodes (including input and output nodes), split, unordered merge and loop condition. Let all loop conditions be used by means of the* loop condition *scheme defined by Definition 17 (of schemes) and let the control flow of G be in sequential form (Definition 23). Let all inputs of G be in one component. Also, let control flow nodes of G be expanded and let the factor graph of G be acyclic. We wish to generate code which realises the graph G on an arbitrary number of data rows, utilising SIMD instructions whenever possible.*

We are almost ready to show that flow graphs with control flow in sequential form may be realised very easily without any need of pull semantics. The only problem we need to address is the problem of buffer sizes. Apparently, buffer size $2 * w - 1$ suffices to accept one (full) vector of data rows in a situation in which there is not enough data for vectorised processing of the next component. One problem lays in loops. Consider a loop with nested branching (as illustrated by Figure 2.23).

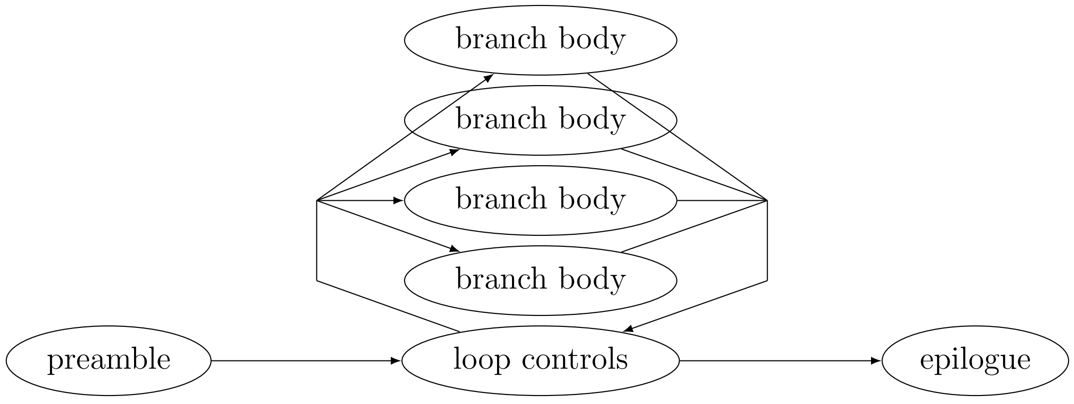


Figure 2.23: Illustration of a loop which may cause a problem if the buffer capacities were not chosen cautiously.

It may easily happen that all branching bodies[14] get filled at some point. The problem comes when these bodies get processed. The problem is that data from all bodies may be directed into the same branch next time they enter the body of the loop (or more generally may be directed into any already-full branch). Thus, some path (or more precisely an entire cycle) may get *entirely* filled. This problem may be solved by limiting the amount of data present in the loop to the capacity of the shortest cycle of the loop. We have already presented the limiting functionality in the definition of the loop condition node (Definition 16). It remains to compute the limit.

Since we now limit the amount of data in loops, there may arise another problem. The problem is that all data rows may get distributed in such way that no buffer contains at least w data rows. This could be solved by employment

[14]More precisely their input and output buffers.

of pull semantics and by enabling the third input of merge nodes again[15]. This third input would then be used solely for the purpose of *finding* data. Another way is ensuring that sufficient amount of data fits into the loop. This may be achieved by increasing the size of some buffer which is shared by all paths.

In other words, we need to determine two functions – *BufferCapacity*, determining buffer sizes, and *Limit*, determining maximum number of records present in a loop. We do so by finding a *SafeCapacity* function, which determines how many records some subgraph guarantees to accept without having to output data. Firstly, the *Limit* function of a loop has to be at least *SafeCapacity* of the loop. Secondly, the *Limit* has to be a number sufficient to populate all buffers of the loop to such extent that there cannot be a loop with all buffers insufficiently populated for vectorised processing. Thirdly, *SafeCapacity* and therefore also *Limit* may be increased by increasing of buffer capacity of a node which is shared by all paths of a loop. We show such a construction in Definition 26 (below). (We add notes on its interpretation, written in upright font.)

Definition 25 (Buffer size-allocation graph). *Let G be a flow graph. Let G' be a flow graph created from G by mapping of all its layer 2 edges to layer 1. We will use the term* buffer size-allocation graph *of G for a factor graph of G'.*

Definition 26 (Construction of capacities). *Let $H(V, E)$ be a* buffer size-allocation graph *of a flow graph $G(V', E')$ defined as in the* unordered realisation of control flow *problem, let $w \in \mathbb{N}$ be a number representing vector size. We shall define the following functions:*

- *MultiPath* $: V \times V \to \wp(H)$ *as* $\mathit{MultiPath}(u, v) = \bigcup_{\substack{P \subseteq H \\ \text{path from } u \text{ to } v}} P$[16].
- *BufferCapacity* $: E \to \mathbb{N}$ *as a function denoting the actual capacity assigned to all buffers between components connected by an edge of E.*
- *ApparentCapacity* $: E \to \mathbb{N}$ *as a function denoting the apparent capacity assigned to all buffers between components connected by an edge of E.*
- *SafeCapacity* $: E \to \mathbb{N}$ *as a function denoting the number of data rows a buffer is guaranteed to accept.*
- *SafeCapacity* $: V \times V \to \mathbb{N}$ *as a function denoting the number of data rows $S = \mathit{MultiPath}(u, v)$ guarantees to accept. More precisely, it is the number of data rows the component u can push into S in vectorised manner without overfilling any buffer and without the component v having to process data.*
- *Capacity* $: \wp(H) \to \mathbb{N}$ *as a function denoting the actual capacity of a subgraph of H.*
- *Limit* $: V' \to \mathbb{N}$ *as a partial function denoting the number of data rows allowed into a loop which is managed by a loop condition, as defined in Definition 16.*

[15]The input may be used to determine which branch contains data.

[16]We understand a path to be a connected subgraph with degrees less or equal two, therefore, we have just implicitly required that this path does not enter a single vertex twice.

Let the values be assigned according to the following rules:

- *Let $e \in E$ be an edge which does not connect components managed by the same loop condition (i.e., this is not an edge from a merge node to a split node such that these two nodes take condition input from the same loop condition node).*
  - $\mathit{BufferCapacity}(e) := 2 * w$
  - $\mathit{ApparentCapacity}(e) := \mathit{BufferCapacity}(e)$
  - $\mathit{SafeCapacity}(e) := \mathit{BufferCapacity}(e) - w + 1$

    This is due to the fact that vectorised version of Algorithm 7 requires at least w empty slots in its data sinks.
- $\mathit{SafeCapacity}(u, v) := \min_{\substack{P \subseteq H \\ \text{a path from } u \text{ to } v}} \sum_{e \in E(P)} \mathit{SafeCapacity}(e)$
- $\mathit{Capacity}(S) := \sum_{e \in S} \mathit{ApparentCapacity}(e)$
- *Let s be a loop condition managing partitions u and v such that u consists of merge nodes and v consists of split nodes. Let $e \sim (u, v)$. We will refer to every such edge as to a* balancing buffer *of a loop.*
  - $\mathit{Limit}(s) := \mathit{Capacity}(\mathit{Multipath}(v, u)) - \mathit{Capacity}(P) + (|P| + 1) * (w - 1) + 1$

    Where P is a path from u to v such that $\mathit{Capacity}(P)$ is minimal.

    Assume that the loop defined by node s contains $\mathit{Limit}(s)$ data rows. Then even the shortest cycle on the loop contains a buffer which contains at least w elements. The multipath corresponds to the 'loop body'.

    The first difference of capacities 'fills' all buffers except for buffers on the path with the lowest capacity.

    The $|P + 1| * (w - 1)$ 'fills' every buffer by $w - 1$ data rows.

    The $+1$ in $|P + 1|$ is for the buffer between v and u.

    The last $+1$ is to make at least one buffer contain w data rows.
  - $\mathit{BufferCapacity}(e) := \quad \mathit{Limit}(s) - \mathit{SafeCapacity}(v, u) + (w - 1) + w$

    This ensures that safeCapacity of the entire loop is at least $\mathit{Limit}(s) + w$.
  - $\mathit{ApparentCapacity}(e) := \mathit{Limit}(s)$

    This stands for the capacity of the entire loop since the edge e is the one which will be counted.
  - $\mathit{SafeCapacity}(e) := \mathit{Limit}(s) - w + 1$

Claim 4 (Consistency of buffer capacities)**.** *We claim that the BufferCapacity function is well defined, i.e., is defined for any buffer in G and the value is determined unambiguously.*

Proof. Apparently, assignment of capacity of the edges from the loop scheme depends on assignment of capacities of all buffers in the body of the loop. Loop nesting may be represented by an oriented tree. Every oriented tree has a topological ordering, which may be used to assign these functions. □

Although we have (approximately) doubled the actual capacity of buffers of every loop, the apparent capacities have remained in $O(n)$ with respect to sizes of bodies of the loops. This means that the memory consumption should still remain reasonable. If we assume that the size of bodies decreases with quotient q with respect to nesting, then the resulting memory consumption is $n * (1 + \sum_{i=0}^{\infty} q^i)$. With $q = 1/2$, which is a reasonable expectation, this means that memory consumption trebles. Unfortunately, the new worst-case memory bound is $O(n^2)$.

Claim 5 (Complexity of buffer size allocation)**.** *Let G, H and BufferCapacity be defined as in Definition 26. Let $n = |V(G)|$. We claim that the capacity required by G is in $O(n^2)$.*

Proof.

- Capacity of the balancing buffer of every loop is at most the capacity of the loop body. This holds since the limit is at most the capacity of the body.
- Capacity of the balancing buffer does not affect any other buffers since the apparent capacity of the loop remains the same.
- Thus, every loop contained in G may allocate at most $O(n)$ memory for its balancing buffer.
- The number of loops present in G is also bounded by $O(n)$.

□

This upper bound seems to be tight. Consider $O(n/2)$[17] loop schemes nested alternatingly with the branching scheme in such manner that always only one branch contains a loop. Let the innermost body be of size $n/2$. This is shown in Figure 2.24. This setup causes all loops to allocate an amount of space corresponding to the capacity of the innermost body.

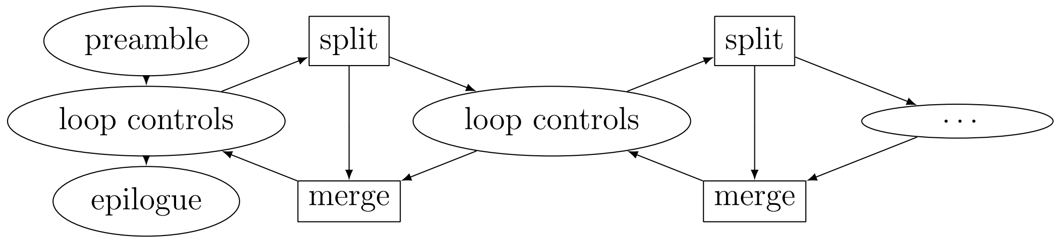


Figure 2.24: A worst-case memory-consumption scenario.

[17]Which is equivalent to $O(n)$ except for the semantical information it carries.

Note that a simple nesting of loops would not suffice, since the only path through the loop body would lead through the next loop body and therefore would get subtracted in the computation of *Limit*. The fact that we cannot subtract the *Limit* in the scenario with branching is caused exactly by the two points discussed earlier:

- The crawler does not know where data are located.
- There has to be a sufficient capacity in case that the inner-most loop produces data which will be directed to the second branch in the next iteration.

It is possible that this upper-bound may be reduced by introduction of mechanism which would purposefully plan the path of the crawler (Algorithm 7). We do not have any proposal at the moment.

Remark. Note that buffer capacities (as defined by Definition 26) may be easily computed by an inverse topological pass of a factor graph of G.

Claim 6 (Unordered solution)**.** *We claim that Algorithm 7 solves this problem if we make the following adjustments. In addition, we claim that its output (a realisation of G) will always use the vectorised version of loop body until the termination phase is entered.*

- *All jumps lead to vectorised labels except for jumps in termination handlers.*
- *Let a and b be processing bodies of merge nodes. Let C be a component such that both a and b output data into C. Then we require a and b to be generated into the same (algorithm) partition.*
- *All buffers have their capacity assigned to BufferCapacity as defined by Definition 26.*
- *Limits of all loop condition nodes are set according to Limit as defined by Definition 26.*
- *Each time a partition receives data input, the terminating variable is set to $Min(terminating, i)$ with i equal to the index of the partition which received data.*

We will show the following points (as we did in the proof of Claim 2):

1. *The algorithm produces correct results (if it terminates).*
2. *The algorithm does not enter an infinite loop during the pre-termination phase unless there is an infinite loop contained in the semantics of G.*
3. *The algorithm terminates correctly.*

Proof of point 1. All data rows are handled as consistent packs, so each data row is handled correctly. □

Proof of point 2. Let there be a flow graph $G(V, E)$ and a sequence of data rows such that the described algorithm enters an infinite loop which does not process any data. Let C be this loop. Let t be topological ordering of a factor graph of G.

Again let $M = \{v \in C | \neg(\exists u \in C)(u <_t v \wedge (u, v) \in C)\}$. M is empty, since every component has either all inputs sufficiently populated or no input sufficiently populated.

This means that the algorithm is either only descending or only ascending in the graph structure, which implies that it traces a loop. Thus, either all buffers on C contain an insufficient amount of data or no buffer on C has a sufficient amount of space. But we have assigned buffer capacities in such manner that this cannot happen. $\square$

Proof of point 3. This is again a simple procedure of checking that every partition processes its data and increments the termination counter. $\square$

Remark. The partitioning algorithm may be easily adjusted for conversion of some flow graphs into their sequential equivalents. The simplest version will handle graphs which contain simple edges which bypass some control flow schemes. This algorithm may also be further refined to handle situations like parallel branching. The point of the adjustment lays in choosing a 'native' outgoing control flow node which will then be pasted on any bypassing edges or before any other control flow which does not match the 'native' control flow node. A question is how to solve some implementation details in order to prevent entering of infinite loops. Another question is how to optimise the resulting graphs since a naive implementation may produce highly inefficient results.

2.3.4 Refinement of out-of-order evaluation

The out-of-order solution seems to be superior to the ordered one in many aspects. An optimal situation would, of course, be if we could return data rows in an arbitrary order. Unfortunately, the data receiver may need the data ordered. We, of course, may add row indices into the data rows for identification, but that does not solve the problem entirely, since either we will have to store the results one by one or someone else will have to retrieve them one by one. Either version would increase the amount of data transfers between CPU and RAM significantly.

Ideally, we would like to reorder the data back using as few RAM stores and loads as possible. For this purpose, we may use the information about performed shuffling of the data. We propose the following approach:

1. We process all the data and store the results into the RAM. Also, we store a sequence of Boolean values per every split or merge node. This sequence will contain a single value per every data row which has been processed by the node in question. This value will represent either left or right branch of the split or merge.

2. We construct a new pipeline which will be an exact copy of the processing pipeline but will be built backwards and will not contain any data processing.

3. Then we pipe the data from RAM into the new pipeline in inverse order. Also, we provide every split and merge node with the values stored by the corresponding merge or split as a control.

Remark. Note that we cannot pipe the original pipeline directly to the one built backwards. Imagine that the first data row to enter the processing pipeline is the last one to come out. If we piped the first pipeline directly into the sorting one, we would have to store *all* the data in CPU registers until the first data row was processed and could be piped into the sorting pipeline.

This approach surely has its drawbacks, but it may, for some graphs, reduce the amount of RAM transfers needed for data reordering.

3. Architectural overview

3.1 Implemented functionality

We implement a rather general framework with the following features and properties:

- The input is provided by means of two text files (or text streams) — a graph and an instruction table. The graph file describes the structure of a graph similar to the definition of a *flow graph* (Definition 4). The instruction table describes general properties of instructions and provides *instruction patterns* which are used to generate the actual code. The patterns are to be in a dollar shell-like expansion format which is described in Section 4.4.
- We implement a vectorised generator of regular instructions as described by Algorithm 3. Indirect width conversions (already discussed in Section 2.2.3) are solved by search for the shortest paths with respect to the set of already generated paths. Our implementation provides output in C language, but may be also easily refined to other languages and environments.
- Furthermore, we implement the order-preserving solution of branching graphs as described by Algorithm 7. *Node expansion* and *cycle removal* (both discussed in Section 2.3.1) are implemented via a general mechanism of graph transformers. Buffer sizes are assigned according to Algorithm 8.
- Example instruction table covering the C language via Intel's Streaming SIMD Extensions is provided. This table has been satisfyingly tested, using degenerated testing graphs and a testing environment.
- We also provide testing features as described in Section 4.7.
- There are 16 test scenarios which are designed to ensure correctness of the implementation and also to serve as examples of use.
- Command-line frontend executable is provided. This frontend may be simply extended by custom commands, graph transformations etc.
- We provide implementation of four different output environments, i.e., four different forms of the resulting code fragment. Three provide support only for graphs consisting of regular instructions. Two of these are intended for testing purposes and one provides a Bobox-specific interface. The last one implements control flow and provides an output with an environment-independent interface. The control flow ('cf') environment is also meant to be further composed with other environments since it contains functionality which may partially be seen as 'core' functionality.

3.2 Brief introduction of components

Our framework, abbreviated as *ctb*, is implemented as a header library consisting of multiple classes composed via templates. It may be used directly from a C++ code as well as a standalone executable. We understand the data processing to be a sequence of actions which are described by a sequence of commands.

First, we would like to present all components of our framework on high level of abstraction. We provide more detailed API descriptions in Chapter 4.

Command environment

The *ctb* class (`ctb.h`) acts as a frontend of our generator. It contains:

- Methods representing commands.
- Interpreter of commands.
- Tables containing various access methods of other templated components. These are specifically:
  - Table of loaders.
  - Table of alias environments which act as separate text processors.
  - Table of alias environments which represent an output environment.
  - Table of commands.
  - Table of graph transformations.
- Methods which allow registration of new commands and components. This allows custom classes to be used without actual changes to our code.
- A single instance of a generator.
- A single instance of an instruction table.

Instruction Table

Instruction table (`instructions.h`) is designed to store definitions of operations, instructions, data types and width conversions. Instruction table supplements the code generator with all information it needs. We provide more detailed semantics at the end of this chapter. For the time being it is more than sufficient to mention that instructions, types and conversions are resolved into string patterns which are later used by the generator. (We refer to these as to *instruction patterns.*)

Graph

We provide an implementation of a graph template (`graph.h`, `graph_factor.h`) with the following features:

- Graph may act as both directed and undirected graph.
- Vertices and edges are implemented as separate objects. Both vertices and edges contain an instance of a user-defined type.
- Support for distance calculation.
- Crawling methods.
- Cycle detection.
- The template contains data fields for all general annotations specified by our definition of a flow graph. This mainly includes semantics of edge layers and graph factorisation.
- Support for computation of a factor graph. The factor is then accessible as a member object.
- Export of both the graph and its factor graph into dot representation.

Generator

Generator (`generator.h`) may be viewed as a special case of the Graph structure. It is parametrised by the type of an instruction table and by the type of a graph. An instance of the generator class generates some instruction code using a provided instruction table (where annotations of graph elements refer to structures contained in the instruction table).

Loaders

Loaders (`loader_*.h` files) are classes which provide methods which import and export representations of graphs and instruction tables. These are required to contain a string identifier and four methods which perform the mentioned operations. Every loader is supposed to be registered within the ctb class. The identifier is then used to specify which loader should be used in commands.

Exact definition of formats is provided in sections 4.5 and 4.6, altogether with the semantics of the generator-related functionality.

Preprocessing environment and the Writer class

As we have mentioned before, our generator uses a complicated text processing environment to generate code. The key class is the Writer class parametrised by a tree structure of alias environments.

Writer

Writer (found in `writer.h`) may be simply viewed as containers of generated code or as a string. Writer was designed to act as a data sink for the Generator class. Besides various io and formatting related functionality, the Writer provides a variadic[1] print method. This method takes a format string, which resembles the `printf`[2] format but looks and acts more like the text preprocessor of Bourne shell. The print method takes its first argument and recursively performs a sequence of expansions. These expansions may, for instance, perform:

- Simple substitution of arguments into the format string.
- Evaluation of arithmetic expressions.
- Evaluation of *aliases*. When a variable with a textual name is encountered, it is passed to the root alias environment which is then responsible for resolving of the alias into a result.

Expansion formats are further described in Section 4.4.

Alias environments

Aliases are string constants standing for other string constants.

Alias environments (`aliasenv_*.h`) are *static* classes which define and resolve aliases for writer objects. Alias environments are invoked via a simple method which takes a string (a name of an alias) and returns another string. An alias environment may simply look up the required alias in a map, but may also perform arbitrary computations or even effects. To allow text-processing contexts combining multiple environments, alias classes are supposed to employ some sort of inheritance, therefore creating a tree-like structure for evaluation.

Alias environments (altogether with the Writer class) represent a generic mechanism which we use in different manners in often unrelated contexts.

For instance, the following alias environments may be found in our framework:

- An empty environment.

[1]Taking a variable number of arguments.
[2]C print method which prints formatted text

- A bind environment, which takes two other environments as parameters and *combines* them into a single environment.
- An aliasenv maker, which is a general environment whose content may be dynamically managed. This is used for purposes which do not have substantial semantics, e.g. various auxiliary usages.
- The control flow macro environment (`aliasenv_cfmacros.h`), which implements buffer macro expansions. This is an example of a more involved usage.
- The generator alias environment (`aliasenv_generator.h`) is used for resolving of generator-related aliases which can be typically found in instruction patterns.
- Environments specific for output environments, further described in Section 4.8. (`aliasenv_simple.h`, `aliasenv_cf.h`, `aliasenv_simu.h` and `aliasenv_bobox.h`)).

Alias environments as drivers of the generation

Some alias environments also represent output environments — for instance, a specific language or a specific way how the basic blocks, generated by a generator, should be composed. These environments, registered in an instance of the ctb class, are the actual top-level drivers of the generation. When code generation is invoked, one of these environments is provided with a generator object and is supposed to use the generator (or more accurately its output) to produce some useful code. These environments are required to have a string identifier assigned.

Preprocessors

A preprocessor is an alias environment which was registered within an instance of the ctb class as a text preprocessor. A preprocessor may later be used for processing of arbitrary input, meaning that an input file is taken, printed using a writer parametrised by this environment and outputted. This functionality makes sense since nontrivial text transformations may be implemented by means of alias environments.

Transformers

A transformer class is a class which provides some graph transformation. This allows arbitrary graph transformations to take place during various phases of computation of our framework. Transformation classes may be registered and used in the same manner as loaders. Again, transformers are identified by string names.

Currently, the only transformer is to be found in file `cf_transofm.h`.

Other structures

Cartesian multiplier

Cartesian multiplier (`cartesian_multiplier.h`) is a simple (but yet interesting) utility which helps to generate cartesian products of containers. The cartesian multiplier is basically an iterator which dereferences to a list of iterators. Values of iterators in this list change with every iteration of the top-level iterator so that all combinations may be observed.

Parser

The Parser class (`parser.h`) is a simple push down automaton which evaluates arithmetic expressions. This parser was handwritten but its working principle resembles standard LR parsers. [3]

Languages

Besides other features, the Writer class provides code formatting and indentation. Language classes (`languages.h`) contain sets of callbacks which are used to determine indentation and line breaking. These are passed to an instance of the Writer class as member typedefs of alias environments.

Implicit containers

There are two container templates (in `multicont.h`) which basically add one dimension to the template parameter. We use these at places where we needed to replace a single (always present) element by a list of optional elements without invalidation of already existing code. These also serve as a syntactic sugar, since we do not require any container semantics unless the user explicitly needs to use multiple instances of the object in question.

Neglecting the fact that `int` may not be used in place of a class, implicit container of integers could work as follows:

```
impl_contA<int> a = 3;
int b = a + 5; //8
a[3] = 4;
int c = a + a[3]; // 7
int d = a[0] + a[3]; // 7

impl_contB<int> a = 3;
a["c"] = 1;
int e = a + a["default"] + a["c"]; //7
```

Figure 3.1: An example of use of implicit containers.

[3]More on the topic of languages and automatons may be found in [15].

Tag handlers

Among other characteristics, any instruction may have a list of tags specified. These tags are used to decide whether an instruction may be used or not. The tag handlers (`taghandler.h`, `tagmaster.h`) are objects decide of feasibility of a supplemented list of tags. In other words, these objects provide a single, arbitrary function from a set of strings to Boolean.

Our implementation of this structure provides more involved features and serves for translation of user-provided flags into masks which are later used for determining whether an instruction may be used.

Proxies

During the initial phase of development, we used to use the Proxy class (`proxy.h`) to make some class members read-only for public. This seemed to be a conceptually good idea, until the management of templated inheritance turned this concept into a nightmare.

3.3 Graph algorithms

In this section, we introduce the employed graph-related algorithms.

We provide code samples in a partially functional pseudo language inspired by C++11. We omit all unnecessary keywords, signatures and parts of signatures which are not necessary for semantical understanding. We do so to simplify and shorten the code.

Code samples shown in this chapter aim to explain the main ideas without all implementation details. The actual implementation may be found in files `graph.h`, `graph_factor.h` and `cf_transformer.h`.

Graph crawler

We implement most of our algorithms using a relatively simple crawling template. This template is designed to be usable for wide variety of tasks. Its semantics roughly corresponds to the following pseudo-code (exact implementation may be found at the very end of the `graph.h` header file). This template was designed to crawl graphs in topological order.

```
fun crawler<recurse, inverse>(node* n, fun f, fun g, list l = {0})
{
  if(!g(n))
    return
  if(recurse)
    foreach( u : parent of n)
      crawler(u, f, g)
  if(f(node))
    foreach( u : child of n)
      crawler(u, f, g)
}
```

Figure 3.2: Our callback-driven crawling template.

`l` list of edge layers to be used.

`Child (parent)` of vertex v is a vertex u such that there exists an edge (v, u) ((u,v)).

`f,g` are possibly effectful functions of type $V \to \mathit{Boolean}$. The function `f` is designed to perform some work while the `g` function is designed to control which nodes are to be crawled.

`Inverse` switches direction of this algorithm, i.e., makes it consider edges in an inversed direction.

`Recurse` turns on and off the search of parents.

We implement the second recursive call using an out-of-scope queue which is passed implicitly to prevent some loops on call stack. This precaution reduces the maximum number of nested calls to the longest directed *stroke*[4] which is present. This functionality (if used properly) allows algorithms to be written so that the maximum number of nested calls does not exceed the length of the longest directed *path*. The actual implementation also handles crawling on a defined subset of graph layers passed by the argument `l`.

Topological search

We implement topological search of connected partitions using the following snippet. This function is very useful since it stands for `foreach(v in topological order) f(v);`. If there is no topological ordering for the component in question, then this code is equivalent to an unordered `foreach`.

```
fun crawl_partition_topological(node* n, fun f)
{
  crawl<true, inverted>( n
    , (node* n){ f(n); n->lastpass = passid; return true;}
    , (node* n){ return n->lastpass != passid;}
  );
}
fun crawl_topological(fun f)
{
  passid++;
  foreach(vertex v)
    if(v->lastpass != passid)
      crawl_partition_topological(v, f);
}
```

Figure 3.3: Topological search algorithm.

There are only three things to note:

- The `lastpass` variable is used in combination with the (out-of-scope) `passid` variable as an auxiliary mark which denotes whether a node has been visited. The `passid` ensures that a new mark which has not yet been used is used.
- This function is not re-entrant as shown. In real implementation, we use a bit more complex call which uses an out-of-scope set instead of the `passid` variable. (This change was made for user convenience when we made the API public.)
- We could write this in an even simpler manner, using only the `g` callback. Unfortunately, such (simplified) solution would enter an infinite loop if there was a cycle in the presumably acyclic graph. The shown solution does not enter an infinite loop, which is the preferred behaviour.

[4]A path which may contain vertices multiple times.

Cycle detection

We detect cycles in two steps:

1. We construct some ordering of nodes using the topological search.
2. We check that the ordering is a correct topological ordering.

Both these steps may be done by simple calls to the topological search algorithm.

```
fun find_cycle()
{
  int order = 0;
  node* node_on_cycle = NULL;
  crawl_topological((node* n){n->order = order++;});
  crawl_topological((node* n){
    forall(e output edge of n)
      if(e->from->order > e->to->order)
        node_on_cycle = n;
  };
  return node_on_cycle;
}
```

Figure 3.4: Cycle detection algorithm.

Shortest path search

For the purpose of shortest-path routeing, we maintain a map of size $|G|$ in every vertex. This map contains one record per every vertex of G. This record contains:

- distance
- next node on a shortest path

This solution, despite being quite minimalist, requires $O(n^2)$ memory, which may be a problem for big graphs. Since we use minimal path search only for the purpose of finding shortest width-conversion path, this limitation does not concern us much. An alternative would be running Dijkstra's algorithm on every request [2].

We fill these routing structures using the well known Bellman-Ford Algorithm [2] implemented using our callback machinery. Equivalent pseudo code may be:

```
foreach(vertex v)
{
  crawl<false,false>( v
    , (node* n){ recalculate n; return updated || first pass;}
    , (node* n){ return true }
  );
}
```

Figure 3.5: Pseudo code of the Bellman-Ford algorithm.

Factor graph construction

Construction of a factor graph is again a simple process. The only interesting part is the second step in which we try to sort the components (pseudo) topologically even if no topological ordering exists. We do so for aesthetic reasons.

We should remind the reader that we use two instances of the standard graph to represent the graph and its factor graph. The actual implementation just uses inheritance to allow the factor graph to be a member of the graph (`class G: public graph_t<...> { graph_t<...> factor; };`). For now, we will simply use the names `G` and `factorgraph`.

1. The crawling algorithm is used to assign component ids[5] to all vertices. We show this in Figure 3.6.
2. In the next step, we try to remap component ids so that the components are in an order resembling our subjective feeling of a correct (pseudo) topological ordering.
    - This task is simple if topological ordering exists - a single pass by the topological search in the factor graph yields a map which is then used to reassign the ids. We show this in Figure 3.7
    - If no topological ordering of components exists, we still try to return nicely-looking results. This makes sense since the underlying graph will usually have a topological ordering.

      First, we assign some ordering to *edges* on levels 0 and 1. We do this by the topological search, so we assume to get the edges ordered (mostly) topologically[6]. Our observation is that some feasible ordering of components is an ordering induced by the order of input edges. Specifically, we pick the maximum input edge which is less than every output edge of a component. We show this in Figure 3.9.
3. Finally, the vertices are gathered into components represented by vertices in the new graph structure (as shown in Figure 3.8).

[5]We use the term *class id* in the actual implementation. This is in accordance with standard algebraic terminology.

[6]Note that our use case is exactly that of graphs which are acyclic on levels 0 and 1.

```
crawl_topological((node* n){n->classid = -1;});
int classcount = 0;
foreach(node n in G)
  if(n->classid == -1)
  {
    crawl_partition_topological(n, (node* n){n->classid=classcount;});
    classcount++;
  }
```

Figure 3.6: Factorisation – class assignment.

```
map<int, int> translation;
int i = 0;
factorgraph->crawl_topological((node* n){
  translation[n->classid] = i++;
});
foreach(node n in G)
  n->classid = translation[n->classid];
```

Figure 3.7: Factorisation – construction of a translation map for acyclic graphs.

```
for( i in 0 .. classcount-1)
  create component i in factograph (i.e. a standard vertex)
foreach(node n in G)
  assign n to component n->classid of factorgraph;
```

Figure 3.8: Creation of the factor graph itself.

```
int order = 0;
factorgraph->crawl_topological(
  (node* n){
    foreach(e output edge of n)
      e->order = order++;
  }
);
map<int, int> minouts;
map<int, int> maxins;
for(i in 0 .. classcount-1)
{
  //this ensures that minouts values are always unique
  minouts[i] = MAXINT - classcount + i;
  maxins[i]= -1;
}
factorgraph->crawl_topological(
  (node* n){
    foreach(e is output edge of n)
      minouts[n->classid] = min(e->order, minouts[n->classid]);
    foreach(e is input edge of n)
      maxins[n->classid] = max(e->order, maxins[n->classid]);
  }
);
for(i in 0 .. classcount-1)
  maxins[i] = min(maxins[i], minouts[i]);
vector<int> translation = keys of maxins sorted by value;
foreach(node n in G)
  n->classid = translation[n->classid];
```

Figure 3.9: Factorisation – construction of a translation map for possibly cyclic graphs.

Buffer size assignment

This algorithm is implemented according to Algorithm 8.

```
map<classid, int> order;
G.factor.crawl_topological(
  (node* p){
    foreach(edge e : input of p)
      order[p->id] = max(order[e->from->id]+1, order[p->id]);
  }
);
G.crawl_topological(
  (node* n){
    foreach(e : output layer 1 edge of n ) {
      coef = order[e->to->classid] - order[e->from->classid];
      e->from->data.set_param("buffercoefout"+(e->from_pos), coef);
      e->to->data.set_param("buffercoefin"+(e->to_pos), coef);
    }
  }
);
```

Figure 3.10: Buffer size assignment algorithm.

The mechanism of setting buffer sizes via parameters is further described in Section 5.4.

Cycle removal

We implement this algorithm as described by Algorithm 5. We believe that the reader can imagine how our crawlers can be used to perform the mathematically described steps.

Remark. The graph transformer provides a step-by-step visualisation of this transformation. It may be invoked by an additional `show` parameter passed to the transformation command.

3.4 Instruction representation

We would like to shed some light on the structure of the instruction table to make the internal workings of the generator clearer. These structures may be found in (`instructions.h`). We shall postpone exact field semantics into Chapter 4.5, which is dedicated to API description. The architecture may be visualised by the following diagram:

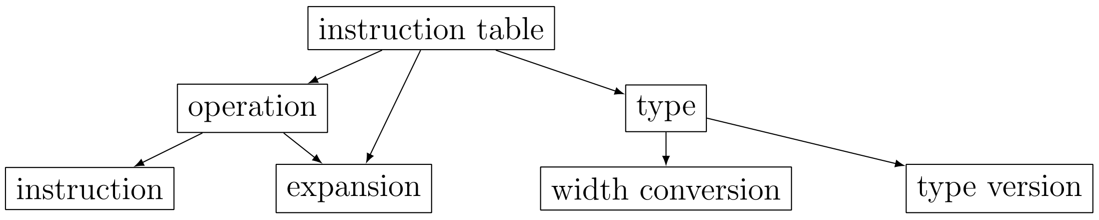


Figure 3.11: Architectural diagram of the instruction table.

Nodes of this graph correspond to either `struct`s or `class`es.

Leaf node structures represent actual *instruction patterns* except for expansions.

Edges may be read as 'a manages b' or 'a contains b' or 'a is responsible for b'.

Instruction table contains:

- Containers of *types*, *operations*.
- An accumulated search table of *expansions*.
- An API which provides mapping from identifiers to these structures.
- *Tag handlers* which are used to restrict the set of employed instructions and their access API.

Operation class represents either logical operations or expansions. In our implementation every operation is allowed to have at most one output. This is w.l.o.g.. Operations contain/provide:

- An *identifier*.
- A container of *instructions* realising this operation.
- A container of *expansions* which are identified by operation id of the operation.
- Operation semantics such as *width*, a list of *input types* and a list of *output types*.
- Publicly available references to *return types*.
- An API for retrieval of *instruction patterns*. The generator communicates directly with an instance of the operation class, asking for results suitable for a specific width. The operation then returns patterns of the most suitable instruction which realises the operation.

Instruction is a small structure which represents a target instruction. This structure contains:

- *Width* information.
- A list of *tags*.
- An *indicator* which indicates whether the instruction is suitable for the current pass of the generator.

- *Instruction patterns.* This encompasses some predefined, specially handled fields and a map of user-named fields, which get outputted into different layers of the supplemented writer[7].

Expansion may be seen as a special case of an operation or as a placeholder which provides some information about what should be done with some part of a graph. All expansion nodes are supposed to be removed by some graph transformation prior to the generation and possibly replaced by new nodes or even by new subgraphs. The motivation for these nodes are exactly graph transformations, due to clash of the following two requirements:

- When generating code, we need to work only with specific type-aware instructions.
- The graph transformations and node types which we have introduced are type independent.

These expansions present a way of binding specific instructions to general transformations by providing a list of instruction identifiers identifying specific target instructions for specific type signatures. These expansions are hence allowed (and assumed) to be present in multiple instances with different type signatures. These contain the following fields:

- A *transformation name* and a *transformer identifier*, which define the graph transformer that should process them and also how it should process them. We should note that graph transformers are free to ignore these identifiers.
- A *transformation identifier*, which also acts as the operation id.
- A *type signature*, which is used to choose specific expansion from the set of expansions identified by the same identifier.[8]
- A list of *instruction identifiers* which specify some target instructions for the expansion.

For instance, consider explicit buffers, defined in Definition 16. Buffers may be represented by single nodes in the input graph. This identifier may have two expansions defined. When the 'cf' transformer is run, it should choose one of the expansions depending on types of neighbouring nodes and expand it to two new nodes.

identifier	inputs	outputs	name	transformer	list of inst.
BUFF	INT	INT	buffer	cf	BUFF_ST_INT, BUFF_LD_INT
BUFF	BOOL	BOOL	buffer	cf	BUFF_ST_BOOL, BUFF_LD_BOOL

Figure 3.12: An example of expansion semantics.

[7]This is implemented by means of one of our implicit containers.

[8]This mechanism allows very simple introduction of type inference. This would be implemented as a graph transformation and its full function would still remain under control of the user via definition of expansions.

Type represents logical data type of the target architecture. It contains:

- Container of *type versions.*
- Container of *type conversions.*
- *API* which resolves requests for instruction patterns of type versions and conversions.

Type version is a variation of a data type specific for a width. It contains:

- *Instruction pattern* representation of this type in the target language.
- *Width.*

For instance, the type *bool* may have *bool/1*, *int/16*, *int/32* and *_m128i/128* as its type versions. (*type pattern/width*)

Width conversion is a structure which represents a width conversion between two widths of a data type. These contain:

- *Input width* and *Output width.*
- *Tags.*
- *Flag* indicating whether tag requirements are satisfied.
- *Instruction patterns.*

Consider again code fragment given in Figure 2.5 (fragment of the vectorised generator). Lines 1, 2, 4, 7, 8 were generated from instruction patterns of *instructions.* The lines 3, 5 and 6 were generated from instruction patterns of two *width conversions* (the lines 5 and 6 are two patterns defined by a common width conversion). All type specifiers present in the figure were resolved from input and output widths and types of instructions and of width conversions. The patterns of instructions and the types were composed together by internal mechanics of the writer. (However, the form of this composition was defined by the form of the pattern. The writer itself is an independent text processor which has *no* information about internal needs of the generator.)

The example in Figure 3.13 shows how a single line of the resulting code may be composed from various patterns. We show a single call to the writer as it may be issued by the generator. Some context is provided via string declarations. The real situation is more complicated since this example does not consider vectorisation. Assume the target environment to be represented by a fictive 'clang' alias environment which inherits the 'generator' alias environment.

This example illustrates the context of the generator, of instruction patterns and of instruction-related structures. However, the main point of this example lays in the principle of the text processing environment, e.g., of cooperation between a writer and alias environments. Understanding of the principle of alias environments is crucial for understanding the entire system.

```
//determined by the node type
string pattern_type = "$opexpr";
//retrieved from the instruction table
string type_code = "int";
string ins_code = "$arg1 + $arg2";
//memoized names of variables containing
//  results of the operations of vertices a and b
string arg1 = "var_a";
string arg2 = "var_b";
//a new variable name constructed from id of vertex c
string name = "var_c";
//the actual call
w.print("$opexpr", type_code, ins_code, width, arg1, arg2, name);
//the process of evaluation
//    ("$opexpr" -> "$type $name = $operation;") //by clang aliasenv
//"$type $name = $operation;"              //intermediate result
//    ("$type" -> "$1")                    //by generator aliasenv
//    ("$1" -> type_code)                  //by writer
//"int $name = $operation;"
//    ("$name" -> "$6")                    //by generator aliasenv
//    ("$6" -> name)                       //by writer
//"int var_c = $arg1 + $arg2;"
//    ("$arg1" -> "$4")                    //by generator aliasenv
//    ("$4" -> arg1)                       //by writer
//"int var_c = var_a + $arg2;"
//    ("$arg2" -> "$5")                    //by generator aliasenv
//    ("$5" -> arg2)                       //by writer
//"int var_c = var_a + var_b;"             //the final result
```

Figure 3.13: Example of instruction pattern evaluation.

`type_code`, `ins_code`, `pattern_type`, `arg1`, `arg2` – some string variables which were retrieved from contextual information, i.e., either directly or indirectly from the structure of the processed graph or from instruction tables

`w.print...` – the actual code which prints an instruction pattern into the writer which acts as data sink for the generator

`comments` – below the call, the comments show the intermediate results of the writer and the flow of evaluation

3.5 Actual implementation of the generator

- The generator is implemented according to the scheme described by Algorithm 3. (The main body is contained in `generator::data_t::generate` method in `generator.h`.)

- The generator provides the writer with a consistent context retrieved from the input graph and the instruction table as described in Section 3.4.
- The problem of indirect width conversions is solved by generation of shortest paths from the set of already generated widths. (Width conversions are handled by the `generator::data_t::get_access` method which is to be found in `generator.h`)
- Types of operations are strictly checked. Expansions are chosen according to the type signature of incoming edges of a node and are also type-checked. The exact expansion is determined by types of incoming edges. The implicit type inference is not required to know all types to decide upon exact expansion due to possible problems with looped dependencies. In such case, a unique matching expansion is found and used. If there is an ambiguity found, a warning is thrown and an arbitrary expansion chosen. The type inference works by assigning types to edges in one topological pass upon graph load and is recalculated upon every transformation. This way transformations have full information about types (otherwise, multiple executions of transformations would be required). Type inference and check takes place in `generator::data_t::infer_type` (`generator.h`). Expansion selection is implemented by means of `instruction_table::find_expansion` methods of instruction table (`instructions.h`).
- All other functionality lays either directly or indirectly in alias environments and instruction patterns (or is an implementation detail).

We should explain one more thing, related to the API of the generator. The generator is actually parametrised by a multilayer writer (created by means of one of the previously described implicit containers) and also by a configuration which specifies how the layers should behave. There are options which allow some layers to receive input only from the first instance of an instruction or to receive only the first string written. This allows the generator to output an arbitrary number of code fragments per every single basic block. The former option may be used for simple accumulation of conditions related to control flow semantics of control flow nodes. The latter may be used for indication of presence of specific instructions or for indication of conditions whose value is not operation-specific.

3.6 Summary

We have already described all significant functionality of our framework. The next chapters describe only implementation details. The reader should now think over all introduced concepts and the architecture of our framework. Indeed, we have designed a framework which can perform all needed steps to output results equivalent to Algorithm 7:

- Our framework can generate vectorised basic blocks as proposed by Algorithm 3.

- The framework can partition the graph as described in Section 2.3.1:
  - Node expansion may be performed by use of a graph transformer whose input is described by expansions. (This is implemented in `cf_transform.h`.)
  - Cycle removal is implemented according to Algorithm 5 by means of a graph transformer. (This is implemented in `cf_transform.h`.)
- Code of a partition, as described by Algorithm 7, may be generated by a custom alias environment. This environment would contain a static template of this partition with placeholders in place of all conditional jumps specified by the algorithm. As proposed, the control flow nodes may accumulate their control-flow related conditions into named layers of the writer. Substituting these accumulated values into the template is then a simple procedure consisting of a single call to a new writer. Our implementation of such environment is to be found in `aliasenv_cf.h`.
- Other generic procedures, such as buffers, may be implemented either externally in the C preprocessor or by means of alias environments. Either implementation is then interfaced via records in an instruction table. We employ the latter way due to stubborn recursion-rejectiveness of the C preprocessor[9]. Specifically, we implement an in-register SSE buffer in the 'cf_macros' environment (`aliasenv_cfmacros.h`).

[9]Recursion may actually be achieved by means of various workarounds. Unfortunately, the implemented algorithms tend to fail for obscure reasons.

4. Technical details and API description

4.1 Usage

Our framework was developed under Linux and therefore it assumes standard Unix tools for development. The source files themselves are supposed to be platform-independent.

- C++11 compliant compiler. Tested compilers are GCC (4.9.3) and CLang (3.7.1).
- TinyXML2 library.
- Make. (Optional - for compilation of executables, testing, development)
- Dot, gpicview. (Optional - for graph visualisation.)
- Standard shell tools. (Optional - for test scenarios and development.)
- Functional bobox environment with aligned memory allocator (optional - for scenario 6)
- CPU implementing SSE 4.2. (Optional - for test scenarios which generate SSE instructions.)

The project may be built by make invoked from the root directory:

```
> make
```

By default, make compiles the ctb executable and unit tests and runs basic test scenarios. The following make targets are available:

basictest compiles and runs basic regress tests.

fulltest runs full check of the sse instruction table and a test of bobox environment.

ctb compiles the ctb executable.

all defaults to ‘ctb’ and ‘basictest’.

Ctb may be used in three ways:

- As a separate executable (`ctb`) controlled by text commands. See the help (`ctb -h`) for list of identifiers identifying available environments and transformers.
- As a header C++11 library. In this case, the `"ctb.h"` file has to be included in the project. The `ctb::ctb` class serves as the library frontend. This frontend may be used in two ways, both of them requiring creation of a single object of the ctb instance:
  - using text commands - see the `int parse_command(string)` method of the ctb class (in `ctb.h`).
  - using raw templated calls to commands - see the rest of public methods defined in `ctb.h`.

Basic usage reference of the provided executable may be obtained by:

```
> ctb -h
```

The text command controls of this library may be very easily interfaced by raw C commands, which should allow direct language-independent integration.

The simplest way to invoke ctb is by:

```
> ctb -f programme
```

Here `programme` is a text file containing a sequence of commands. Examples may be found in test scenarios.

Use from some programming environment is meant to be done by means of creation of a single instance of the ctb class. For instance:

```
ctb::ctb c;
c.load_instab<xml_loader, ifstream>(ifstream("table.xml"));
```

4.2 CTB commands

The input programme has to conform to the following rules:

- Every line represents exactly one whole command unless this line is empty or commented.
- Any line can be commented by insertion of `#` at its beginning.
- Command consists of non-empty space-separated strings. Every sequence of one or more spaces is considered as one delimiter.

- The first string in the list identifies the command to be run. The rest of the list is passed as arguments to the command.

The available commands are:

- `help` shows a standard UNIX-like help.
- `source <input file>` – interprets contents of the file as its argument.
- Import and export commands. These take the identifier of a loader class as the first argument and a file name as the second argument. The structures are loaded to/exported from the same object of a graph or of an instruction table. The structure is always cleared before load. Note that not all loaders have to implement all import end export functions.
  - `exportgraph <loader> <output file>` – available for the 'dot' loader
  - `exportinstab <loader> <output file>` – available for the 'csv' loader
  - `loadgraph <loader> <input file>` – available for the 'xml' loader
  - `loadinstab <loader> <input file>` – available for the 'xml' and 'csv' loaders
- `generate <output aliasenv> <output file>` – the `aliasenv` is invoked with the loaded generator and instruction table. The produced string is written into the `output file`.
- `testgraph` – testgraph is a custom command which acts as a special version of loadgraph. This loader function constructs a graph from the loaded instructions, trying to utilise all available operations.
- `adddebug [depth=1 [<list of vertex ids>]]` – This command hooks a debug node[1] to every vertex specified in the space separated list of vertex ids as well as to predecessors of these nodes up to `depth`. Debug nodes are automatically selected from operations of type *debug*. If the list is empty, debug nodes are hooked to all vertices.
- `transform <transformer>` – the graph transformed identified by the first argument is invoked. This transformer is free to edit the loaded graph as it pleases.
- `visualise` – writes the currently loaded graph into a temporary file and invokes the dot (graphviz) with gpicview to show the graph to the user. We use this mechanism for error reporting as well.
- `preprocess <aliasenv> <filein> <fileout>` – processes the `filein` by the aliasenv identified by `aliasenv` and writes the results to `fileout`. The `<aliasenv>` string may be found in a list provided by the `ctb -h` help.

[1]These are special output nodes which are supposed to output values directly. These are not subject to width conversions.

4.3 Directory structure

Framework source codes

`ctb/aliasenv_*.h` files contain definitions of alias environments.

`ctb/cartesian_multiplier.h` contains the cartesian multiplier.

`ctb/cf_transform.h` contains graph transformations which perform cycle removal, node expansion and buffer size allocation.

`ctb/context.h` contains definition of a sample structure interfacing the 'cf' aliasenv output.

`ctb/{conversions.h,datatypes.h,defines.h,errorhandling.h}` contain auxiliary structures.

`ctb/{graph.h,graph_factor.h}` contain implementation of graph structures.

`ctb/generator.h` contains implementation of the generator.

`ctb/instructions.h` contains implementation of structures storing instruction tables.

`ctb/loader*.h` contain definitions of structures for import, export and possibly dynamic creation of graphs and instruction tables.

`ctb/multicont.h` contains our implicit containers.

`ctb/parser.h` contains an evaluator of arithmetic expressions.

`ctb/{proxy.h,ptrglue.h}` contain auxiliary structures serving for syntax-related constructs.

`ctb/split.h` contains string-processing utilities.

`ctb/{taghandler.h,tagmaster.h}` contain definitions of tag-handling structures.

`ctb/writer.h` contains definitions of the Writer class.

Other framework files

`ctb/main.cpp` constructs the ctb executable.

`ctb/test.cpp` constructs an executable containing all unit tests.

`ctb/Makefile` contains (machine) description of the building process.

`ctb/{ctb,test}` are the produced executable files.

`ctb/control-flow-notes/` contains the first draft of partition-handling code.

`ctb/sse_cf_macros/buffer.h` contains a set of C macros implementing a SSE in-register buffer. This set is not fully functional.

`ctb/sse_cf_macros/pp_macros.h` contains macro support for the buffer. These are taken from a third-party repository [16].

`ctb/sse_set/src*.csv` contain definitions of SSE instructions. These are split into multiple files to prevent duplicities in different load/store sets.

`ctb/sse_set/C_table*` are tables concatenated from the partial source tables. There is a deprecated version used by the 'bobox' environment and a refined version which expects specific handling provided by the 'cf' environment.

`ctb/templates/*` are template files used by some alias environments.

Test related files

`ctb/tests_table` contains brief description of test scenarios.

`ctb/test*/` directories contain test scenarios.

`ctb/test*/Makefile` are used to execute tests. This may be done by invocation of make.

`ctb/test*/*.expected` are files which are diff-checked against actual results.

`ctb/test*/graph.xml` contain description of input graphs.

`ctb/test*/instab.{xml,csv}` contain description instruction sets for scenarios.

`ctb/test*/{program,program_visual}` contain task description for ctb. The visual version usually contains added visualisation commands and arguments. The visual version may be invoked by `make visual`.

`ctb/test*/check_output.sh` contains a sequence of commands which check that results obtained from the tests are correct. (If not present, then this is done by Makefile).

`ctb/test*/astable.sh` is an awk script which formats streamed tabular information into a tabular view.

`ctb/test*/run.sh` present in tests which encompass control flow invokes the test executable altogether with the `astable` script. This results in a nice visualisation of content of buffers between any two iterations of the crawler algorithm.

`ctb/test*/macros.h` contain test scenarios in case of scenarios which encompass control flow (12, 13, 16, 17).

`ctb/test*/*.{h,cpp}` also contain various test-related functionality such as programmed result-checking, macros for dumping of register content, etc..

Thesis files

`ctb/thesis/*` are tex template files provided by Martin Mareš. Some of these files have been edited.

`ctb/thesis/contents/*` are content files.

`ctb/thesis/graphs/*.dot` contain graph figures of this thesis. These graphs are written in the dot format, i.e., the same format we use for visualisation of graphs in ctb.

`ctb/thesis/graphs/*.sh` are scripts which preprocess and postprocess the dot files. The primary purpose of postprocessing is making graphs more compact.

`ctb/thesis/graphs/fixed/*.tex` are graphs whose tex code needed to be edited by hand. These files are used instead of those of same name generated by dot.

`ctb/thesis/{Make,contents/{Make,graphs/Make}}` take care of building process of this thesis.

4.4 Text preprocessor

This section explains how the Writer class (called also *the preprocessor*) works. This may be of interest since all text is handled using the writer class. Also the instruction tables specified in csv are preprocessed by the writer.

As we have already indicated, the writer class is basically a string container which allows appending of expressions via a printf like function. Unlike printf, the print method uses shell-like dollar expressions. This allows reuse of arguments and does not restrict the order of arguments. A very simple example follows:

```
writer w;
w.print("There were $1 $2", 117, "dogs");
w.write_str()
```

yields

```
"There were 117 dogs"
```

Figure 4.1: A simple example of usage of writer's print method.

There is a number of expansions which take place, most of them recursively, when the string is being written into a writer object.

Expansions

The order of expansions is the following:

1. Cartesian expansion. This expansion is performed only once.
2. All other types of expansions. The `${}` and `$[]` expansions are carried out by a new writer with the same alias environment.

Argument expansion

`$<n>` (e.g., `"$2"`) is resolved to the *n*th argument. This argument is converted to a string and printed using a recursive call which again performs all expansions except for the cartesian expansion.

```
writer().print("a[$1] = a[$1] $ $2", i, j).write_str()
```

yields

```
"a[i] = a[i] $ j"
```

Figure 4.2: Another simple example of usage of writer's print method.

Alias expansion

Patterns resolved by the alias expansions take the form `$<[a-zA-Z]+>`. The identifier is resolved using an alias environment. We should remind the reader that environments are static classes, which are passed to the writer by means of a template parameter.

Cartesian expansion

Cartesian expansion takes place only once at the beginning of evaluation of the print call. First, the format string is scanned for all expressions in the form of
`${ <identifier> -> <value1>, <value2> ...}` or
`${ <identifier1>, <identifier2> ...  -> <value1>, <value2> ...}`
This expression is not allowed to contain nested curly braces. Every such expression is transformed into a list of tuples and replaced by `$<identifier1>`. Then, a cartesian multiplier is constructed from all these lists. Finally, the writer iterates over the product. Per every iteration a new auxiliary environment (which inherits the original environment) is constructed, filled by the currently observed result of the cartesian product and used to print the entire format pattern.

This means that per every line of input, multiple lines of output are emitted, featuring all combinations of variables specified in the lists.

The left side of the expression, i.e., the identifiers, declares a *tuple* of aliases (constant variables). The first $|tuple|$ values from the right side are then assigned to this tuple and printed. This is repeated as long as there are any values remaining on the right side.

```
writer().print(
  "${ colour -> red, blue } $animal; nice ${ animal -> dog,cat };"
).write_str()
```

yields

```
"red dog; nice dog"
"red cat; nice cat"
"blue dog; nice dog"
"blue cat; nice cat"
```

Figure 4.3: An example of the cartesian expansion.

```
writer().print(
  "${ colour,animal ->blue,dog,red,cat}ish $animal"
).write_str()
```

yields

```
"blueish dog"
"redish cat"
```

Figure 4.4: An example of tuples in the cartesian expansion.

Recursive expansion

Expressions of the form `"${ whatever that does not contain '->' }"` are evaluated in two steps:

1. The content of braces is expanded.
2. The result is evaluated as a standard alias. That means that the result is evaluated by some alias environment and then printed into the current writer. Of course, the final print is again performed with all expansions.

This expansion allows correctly paired nested curly braces.

Arithmetic expression expansion

Expressions of the form `$[ <expression expression> ]` denote arithmetic expressions. First the content is evaluated by a new writer object. Then, the resulting arithmetic expression is evaluated by the Parser class.

Supported operators are: (,), $-$, $+$, $*$, /, %

Import and export modes

Remark. This subsection describes more advanced features of the preprocessing environment. Knowledge of these is not required for basic use of ctb.

If the writer is used to import or export expressions which are to be later processed by another writer object, careful handling of dollar expressions is crucial. For this purpose, we introduce four import/export modes. The writer takes two modes as template parameters. The first mode specifies how the writer should behave when writing data into its private storage. The second mode describes how it should behave when writing them out.

The life-cycle of an expression is the following:

1. Print is called.
2. Cartesian expansion is performed, resulting in a sequence of calls into an internal print method.
3. Expression is recursively evaluated - single dollar expressions may get expanded or not, depending on the first import/export parameter. During this step, double (and more) dollar expressions are handled according to the first import/export parameter.
4. Expression is appended to the internal data container. Also, the expression is broken into lines using the member language type of the root alias environment.
5. Expression remains in the internal storage until data output is requested.
6. Content of the writer is outputted. During this phase, all lines are indented and dollar expressions are *again* processed in order to either double, ignore or reduce the multiplicity of dollar signs. This time, the second mode argument is used.

We show the semantics of import/export modes in the following table:

	Ignore	Eat	Let	Expand
$ expression	$	is expanded	is expanded	$$
$$ expression	$$	$	$$	$$$$

Figure 4.5: Semantics of import/export modes of the writer class.

Hence, typical combination will be *Let+Eat* for import operations and *Let+Expand* or *Ignore+Expand* for export.

Formatting and language

Formatting is driven by some specialisation of the language structure which was already mentioned. Language structure is supposed to provide the following two methods. These methods are called on every character of the printed string.

-
  ```
  void shouldindent(
  string line, int& outindent,
  int& indent, int& nobreak
  )
  ```
  [2]

  line is the line to be indented.

  outindent is the actual indentation of the line.

  indent is a persistent indent context.

  nobreak allows suppressing of the next *nobreak* line breaks.

-
  ```
  void shouldbreak(
  int pos, string line,
  bool& break_before, bool& break_after
  )
  ```

  pos and line provide an exact position and its context.

  break_before and break_after are returns which indicate whether the string should be broken before or after the position identified by the *pos* argument in *line*.

4.5 Format specifications

Our basic input method is input in form of xml files. Besides xml file, we implement also a plain text input interface for instruction tables since flat files turned out to be much easier to create.

Regarding output methods, our framework creates neither graphs nor instruction tables, so we do not have any motivation to provide any implicit output interface. All output interfaces we do implement we implement for sake of development convenience.

[2]The ampersand denotes references. In this context, it means that ampersanded arguments act as return values. Besides that, in the actual implementation, we, of course, use constant references to pass the potentially long strings.

4.5.1 XML

We use standard XML format, as parsed by the TinyXML2 library. Any field of any structure may be represented either by an attribute or by a separate child element. Of course, the fields which are present in multiple instances have to be present as child elements.

The structure is described by figures 4.7 and 4.6. The attributes which are not supposed to be present in multiple instances are specified as the lists following their parent elements.

```
root
├── instruction_list
│   └── operation - input, output, debug, opid, out_type, in_type
│       └── instruction - widths_in, width_out, code, note, tags,
│           rating
│           └── code_custom
├── expansion_list
│   └── expansion - opid, name, transformer_name, arguments, note
└── type_list
    └── type - typeid
        ├── type_version - width, bitwidth, code, note
        └── type_conversion - width_in, width_out, code1, code2,
            code_custom, code_generic, note, tags, rating
```

Figure 4.6: The structure of xml instruction table files.

```
graph_list
└── graph
    ├── vertex - params, vid, opid
    └── edge - from, to, from_pos, to_pos
```

Figure 4.7: The structure of xml graph files.

The following structures are expected to be present at most once:
`root, instruction_list, expansion_list, type_list, graph_list, graph`

The exact meaning of the fields will be given in one of the following subsections.

4.5.2 CSV

Our format has the following general properties:

- Every line stands for one record unless this line is empty, commented or joined with the following line.

- Every record consists of tab-separated fields.
- A line can be commented by # put at the very beginning of the line.
- Any line that has the \ at its end is joined with the following line.
- Our use of this format has the following semantics:
  - Fields which represent lists are comma separated.
  - This file is preprocessed on load by a text preprocessor[3] when the file is loaded.

Although we use the term CSV, the reality is that our flat plain-text format has almost nothing in common with the RFC 4180[4].

The reason why we have decided to add a flat representation of instruction tables is that flat plain-text formats are much easier to process by general text-processing tools. Specifically, our problem is that a single instruction table may contain thousands of instructions, which typically share some patterns in some fields and mainly the fact that someone has to write them. For this reason, we have decided to create a flat format which would be preprocessed by our own custom preprocessor. Specifically the cartesian expansion was designed specially for the purpose of simple description of instructions.

This approach, of course, presents one difficulty — we wish to encode a tree structure into a flat file. We flatten the structure by using one line per every leaf of the tree while specifying significant information of the entire path. This way, some information may be present duplicitly. In reality, we often build instruction tables bottom-up, using a single scheme per multiple leaf instructions, therefore generating multiple instructions of (typically) *different* operations by one line.

The motivation presented in the previous paragraph may need an example. Imagine that we want to generate multiple vectorised versions of the bitwise *or* operation. We will most likely use the same instruction pattern (the same intrinsic function) and just specify it for all integer types which we have defined (signed and unsigned integers of varying bit lengths). This way, we generate one instruction per every integer *or* operation. Note that these are different operations since they operate on different types.

We distinguish multiple different types of lines. We do so by the second column, which contains a type of record. The first column serves for user-defined note, which serves for user's orientation in the file, development notes, debug, etc..

We show the structure in Figure 4.8. In this tree, every path between the root node and any leaf list specifies one possible structure of a line. The fields found on this path must be present in their order.

[3]An instance of the Writer class parametrised by some suitable parameters.

[4]RFC stands for Request For Comment and is a standard means of publishing internet standards.

```
note
 ├─ type=instruction
 │   └─ (operation definition) out_type, in_types, opid, flags
 │      (instruction definition) widths_in, width_out, code, tags,
 │      rating, code_custom, code_custom, code_custom...
 ├─ type=expansion
 │   └─ (operation definition) out_types, in_types , opid
 │      (expansion definition) name, transformer_name, args
 ├─ type=type_version
 │   └─ (logical type definition) type_id, bitwidth
 │      (version definition) width, code
 └─ type=type_conversion
     └─ (logical type definition) type_id
        (conversion definition) width_in, width_out, code_1,
        code_2, code_custom, code_generic, tags, rating
```

Figure 4.8: The structure of csv-formatted instruction tables.

4.5.3 Detailed field semantics

Common

Identifiers - identifiers are supposed to be strings consisting of alphanumeric characters and underscores. This is not a restriction, just a recommendation. Note that the identifiers starting by an underscore are reserved for internal use.

`note` is an arbitrary user-defined string. This is not used by the framework at all, but is preserved for potential debug purposes or for export.

`tags` is a comma separated list of user-defined strings. These serve for restriction of the instruction set. These flags may be restricted through the command line interface or directly from an alias environment.

`rating` is an integer which allows definition of precedence. The instruction with the highest rating is used if there are more feasible instructions for the required width.

Operations and instructions

`flags` – a comma separated list consisting of flags from the following list. These identify special types of operations. These nodes may be treated specially by graph transformations and also affect how the default `code` field is used.

- `input` – an operation which acts as a data source. Control flow operations, e.g., operations which carry data via effects over layer 1 or 2 edges are not supposed to be marked as `input` or `output`, but as `effectinput` or `effectoutput`. Type inference is not applied over layer 1 or 2 edges.

`output` – an operation which acts as a data sink. Analogical to `input`.

`effectoutput` – an operation which is similar to output but serves rather for an effectful transport of data. In other words, operations flagged by this flag are not provided with any target storage. These operations may serve for storing values into some temporary place. Also, this flag indicates that output edges should not be type checked (these edges are created by graph transformations, so no check should be necessary). This means that the `out_types` field is ignored. Both `effectoutput` and `effectinput` are handled as `input` and `output` operations during code generation. However, neither `effectinput` nor `effectoutput` operations are marked as inputs and outputs for the purpose of graph transformations, i.e., these edges are not present in lists containing input and output nodes.

`effectinput` – an operation analogical to `effectoutput`.

`debug` – an operation which acts as a special data sink for the `adddebug` command. These nodes have to be specified for all types and all widths if `adddebug` is to be used. Their type is chosen automatically if they are added via the `adddebug` command.

`noop` – a special operation which stands for no operation. Such operation is completely ignored. No type checks, no generation, no structure checks. The only requirement is that these nodes have no output edges. This mechanism may be used to bind multiple nodes into the same component. One operation of this type is always (automatically) generated - its identifier is `_noop`. The user should never need to use this flag.

`expansion` – an internal operation type which denotes an operation which stands for expansions. The user should never need to use this flag.

`opid` – operation identifier.

`out_type` – `type_id` of the output type. The types of operations are strictly checked upon generation with respect to 0 layer edges. Only the first type is considered during generation. Output edges of operations with `output` flag are checked to be empty regardless of this value. Output types are also not checked for nodes flagged by `effectoutput`.

`in_types` – list of `type_ids`. Type checking conditions are analogical to `out_type`. Input types are not checked for nodes flagged by `effectin-put`.

`width_in` – input width of a type conversion (i.e., the number of independent data rows processed by a single instruction).

`widths_in` – comma separated list of input widths of an instruction. Analogical to `width_in`.

`width_out` – output width of an instruction. Every instruction has to have some width defined (otherwise some control flow nodes would be without any width specification). For this reason, the `width_out` serves as the primary width identifier and is always legal. This field may be omitted for output nodes — in that case the value is determined as maximum of the `widths_in` list. In most cases, all values of `widths_in` and `width_out` are supposed to be equal for instructions (not for width conversions). The fields have to be specified because operationsmay exist which have properties of both data processing operations and of width conversions.

`code` – an instruction pattern for the instruction. This field is used with respect to the node type, which means that it is substituted into one of three predefined patterns. For instance `code` may be `"$arg1 + $arg2"`. Note that the dollar signs must be doubled in case of the csv format since the csv loader expands single-dollar expressions. Also note that argument expressions are indexed from 1 while edge positions are indexed from 0.

`code_custom` – taking the format of `"<identifier>: <instruction pattern >"`. The pattern will be printed into a layer of writer which is identified by `<identifier>` — for this reason it may be present multiple times. Custom code does not use a predefined code template, so a full code must be present. For example `"$type $name = $arg1 + $arg2;"`.

Types, type versions

`type_id` – an identifier of a type.

`width` – width operations which operate on this type version. Or length of this vector.

`bitwidth` – a hardware bit length of this type per one record. This allows non-trivial extension-specific computations in instruction patterns. This may be 1 or 8 for a Boolean type independently of the `width` argument.

`code` instruction pattern for this type. For instance `"int"` for an integer.

Type conversions

`width_in` – width of input vectors.

`width_out` – width of output vectors.

`code1, code2` if *Width_in*/*Width_out* = 2, these two fields act as selection expressions of the first and second half of the input vector. If *Width_out*/*Width_in* = 2, then the second field is ignored and the first one acts as a merge expression merging two vectors into one. We show this in Figures 4.9 and 4.10:

```
width_in=1
width_out=2
code1="$arg0 | ($arg1 << 16);"
```

Figure 4.9: A width conversion corresponding to line 3 of Figure 2.5.

```
width_in=2
width_out=1
code1="$arg1 & 0xFFFF"
code2="($arg1 & 0xFFFF0000) >> 16"
```

Figure 4.10: A width conversion corresponding to lines 5 and 6 of Figure 2.5.

`code_custom` is a non-managed version of the field. This means that all names, types and conversions have to be present in this single field, if this field is to be used on its own.

`code_generic` allows iterated select or insert operation. This means that we may use this field if we can describe all expressions by the same pattern. For instance, if we want to join four elements into a single vector, we may declare the target variable by the `code_custom` field and then write a generic select which will modify the variable once per every call. For this reason, the `code_generic` is guaranteed to be generated after `code_custom`. This field is non-managed (i.e., you have to provide a full expression including all possibly needed declarations, etc.).

```
code_custom="$type $name[4];"
code_generic="$name[$vindex] = $arg;"
```

Figure 4.11: An example of use of the `code_generic` field showing join of four values into one vector.

Graph fields

`params` – comma separated list of assignment such as `"var1=dog,var2=cat"`. Hereby specified values are available during code generation. In other words these values may be used in instruction patterns.

`vid` – identifier of a vertex.

`opid` – operation identifier.

`from` – vertex id of an edge.

`to` – vertex id of an edge.

`to_pos` – the *To* annotation identifying arguments of operations. This is indexed from zero.

`from_pos` – the *From* annotation.

4.5.4 Instruction pattern fields semantics

This subsection describes aliases provided by the generator, i.e., 'variables' which may be used in instruction patterns. Of course, any other aliases may be defined in custom alias environments and used in instruction patterns.

Generator-specific aliases

`type` – a pattern of the type of the result for the correct width.

`name` – a name generated for the variable.

`name1, name2...` – names generated for the variables in the `code_custom` field of split operation. These are also generated for standard instructions if the widths are not equal.

`classid` – index of the current partition. These are in topological order with respect to the graph structure.

`basename` – name of the first variable in vector.

`widthin, widthout` – the corresponding vector widths.

`packsize` – the number of values processed in one iteration of the vectorised body. Also called w in this text.

`operation` – the instruction pattern which is being processed.

`arg1, arg2, arg3...` – arguments of the operation in question. These are names memoized from some previous iteration of the generator, generated either from an instruction or from a width conversion.

`arg1n1, arg1n2...` – allow access to the name1, name2, etc. of arguments in case of instructions with different widths.

`iindex` – input index – index denoting position of the first data row of the vector in the currently processed pack of data.

For instance, for standard instruction of width 4 and pack size 16 the pattern will be expanded four times with values 0, 4, 8 and 12.

`oindex` – output index – output analogy of the `iindex`. These two values will differ in width conversions.

`vindex` – vector index – index denoting position of the currently processed subvector in a vector processed during width conversion. In other words, this is always the difference between `iindex` and `oindex`.

This value is meant for use with generic joins and splits. For instance, assume width conversion from 8 to 2 at pack size 16. Then the `code_generic` pattern will be expanded with values 0, 2, 4, 6, 0, 2, 4, 6. The `iindex`es will be 0, 0, 0, 0, 8, 8, 8, 8 while `oindex`es will be 0, 2, 4, 6, 8, 10, 12, 14.

Other default aliases

The following patterns are specified as constants in the aliasenv of the generator as default C-specific patterns. Since these are language specific, one may need to override them. This may be done by their specification a custom alias environment. Those are exactly the patterns which are chosen based on the `flags` field and affect the (only) `code` field (and `code1`, `code2`).

```
"$innercode" -> "$type $name = ($type)($operation);"

"$declcode" -> "$type $name;"

"$outputcode" -> "$operation /*$name*/;"

"$inputcode" -> "$type $name = ($type)($operation);"

"$conversioncode" -> "$type $name = ($type)($operation);"
```

Figure 4.12: Aliases defined as default patterns in the generator environment.

We also propose the use of the following three aliases which lay entirely in user space.

`ioindex` – this field is meant to identify input or output data for the purpose of effects. We propose this index to be set as a parameter in vertices of input and output operations. We also propose this index to be set to a numeric value so that input and output data streams may be processed in both dynamic and static manner by a generic algorithm.

`input`, `output` – we propose these fields to be constant patterns set in an alias environment to expressions representing actual input or output values consistent with semantics of the input and output instructions. These patterns would then be used in instruction tables. This proposal ensures that instruction tables are not polluted by environment-specific side effects.

For instance, `"$input" -> "input_$ioindex[j]"`. This pattern assumes that the context filled in by the alias environment responsible for composition of output prepares data into arrays and then iterates over them in a loop with index j.

4.6 API description

Note that in this section, we will omit all insignificant information such as constant reference markings or template definitions.

4.6.1 Loader API

Loaders are required to implement the following five methods:

```
void load_graph(G& graph, istream&);
void load_instab(IT& instab, istream&);
void export_graph(G& instab, ostream&);
void export_instab(IT& instab, ostream&);
static string get_name();
```

Figure 4.13: API of loader classes.

`G, IT` are a templated types of an instruction table and of a generator.

`istream, ostream` are C++ implementation of input and output streams. This way, direct interfacing with another programme is possible.

`get_name` is required to return a sensible name not colliding with names of other loaders.

We believe that provided the description above, the API is self describing.

When a new loader is implemented, it is supposed to be registered within the ctb object by its templated `register_loader` method. This call may be inserted into the `fill` method of ctb. Or may be inserted into a new instance of the `main.cpp` file. Registration is not necessary if the user does not wish to use the text command controls.

4.6.2 Aliasenv API

Fully functional alias environment which acts as an output environment is supposed to provide the following interface:

```
typedef <typename> language;
static string alias(string a, bool* s = NULL);
static string get_name();
static writer generate(int packsize,  G& generator,
  string name, stringlist args);
```

Figure 4.14: API of alias environments.

`language` is a language structure which is to be used by any writer parametrised by this environment.

`alias` is a method which tries to evaluate the string a. If the second parameter is not null, it is also supposed to report whether the attempt was successful.

`get_name` is again any sensible string identifier.

`generate` is a method which is supposed to generate the resulting code fragment using its arguments:

packsize is the size of the pack for which the body is supposed to be generated, i.e., the w from chapters dedicated to theoretical analysis. The environment is allowed to ignore this parameter and decide its own suitable size.

generator is again a templated generator type. This variable provides direct access to the underlying graph as a simple member variable.

name is a simple identifier meant for identification of the generated code fragment. E.g., the alias environment may output its results as member methods of a class whose name contains this identifier.

args is a list which may contain other arguments.

If an environment is constructed to act just as an alias processor, only the alias method and the language type are obligatory. If an environment is not supposed to be used as the top-level parameter of writer, the language field may be omitted as well.

As other structures, environments which provide the generator method are supposed to be registered within ctb by the register_aliasenv method.

4.6.3 Generator API

Generator provides the following methods for the purpose of code generation.

```
void generate(int packsize,
    imp_contB<W>& w, imp_contB<output_options>& options);
void generate_partition(int partition, int packsize,
    imp_contB<W>& w, imp_contB<output_options>& options);
imp_contB<output_options> option_struct();
int partition_count();
bool partition_is_topo_max(int partition);
bool partition_is_topo_min(int partition);
void update_types();
void update_factor();
graph_t& get_factor();
```

Figure 4.15: API of alias environments.

w is the writer which is supposed to receive the output. This type may be constructed as follows:

```
impl_contB<writer<example_aliasenv>> w;
```

The results in non-default layers are to be retrieved as follows:

```
string result = w["empty_output_indicators"].write_str();
```

`update_types` forces full type inference and structure check. This may be necessary during graph transformations.

`update_factor` updates the factor graph. This has to be called manually because unstable factor graph is not desired during transformations.

`options` specify special behaviour for specific layers. Instance of this structure may be either manually constructed or obtained by the `option_struct` method. Setting options works exactly as in case of the writer. For instance:

```
auto o = generator.option_struct();
o["empty_output_indicators"].unique = true;
```

Currently available options are:

- `once` causes generator to write an instruction into such layer only once per vertex.
- `global_once` causes generator to write the value into such layer only if the corresponding layer of writer does not contain anything yet.
- `layer` causes generator to write the pattern into the layer specified by this field. This applies only if this field is non-empty.
- `order` allows specification of evaluation order of patterns. Patterns of layers with lower `order` field get outputted before those with higher `order`. This defaults to `-1`. This is useful if combined with the `layer` option since this allows the user to specify how patterns of different fields should be interleaved.

For the purpose of graph transformations, the generator provides the following methods. These methods interface the corresponding methods of the member graph object while handling generation-related data specially. Advanced functionality, such as custom graph crawling, is provided by the graph itself.

```
node* get_vert(J id);
node* add_vert(v_id v, op_id op, map<string,string> params);
void add_edge(v_id from, v_id to, int to_pos, int from_pos, int layer = 0);
void rm_vert(J v);
void rm_edge(edge* e);
void connect_as(J v, K as, bool inputs = true, bool outputs = true);
void foreach(function<void(node*)> f);
```

Figure 4.16: Graph API of the generator.

`J, K` are automatically resolved types identifying vertices. These may be pointers or references to the actual vertex structure or just vertex ids.

`v_id, to_pos, from_pos, params` correspond to the described field semantics.

`get_vert, add_vert, rm_vert, add_edge, rm_edge` apparently allow altering of the graph structure.

`connect_as` adds new edges to the vertex `v` imitating the way the `as` node is connected.

`foreach` calls the function `f` once per every vertex in topological order. This function may be nested.

The retrieved nodes then provide the following api:

```
T data; I id;
int classid, colourmark;
imp_cont<vector<edge*>> in, out;
edge* in_at,out_at(int i, bool check_uniqueness = true);
```

Figure 4.17: API of graph nodes.

`id` contains vertex id.

`classid` is id of the component of connectedness if the factor graph is up-to-date. Make sure you have called the `graph.update_factor()` method. This is not done automatically since factor graph and this id is apparently not stable with respect to factorisation.

`colourmark` is and auxiliary identifier which may be used by the user for marking of vertices during graph algorithms.

`data` contain the data type specified by the `T` template parameter. The generator's data type contains a pointer to the corresponding operation (`op`), an operation id (`opid`), parameter access methods (`get_param(string)` and `set_param(string,string)`).

`in, out` are containers containing pointers to incoming and outgoing edges. Layer 0 may be iterated by simple `for(e : in)`, other layers by `for(e : in.get_layer(1))`.

`in_at, out_at` provide direct access to the `i`th edge.

The underlying graph may be accessed directly through the `graph` member. The factor graph may be accessed through the `factor` member of `graph`. The factor graph again provides the same api as shown above. Namely, the `get_vert(int p)` method returns a vertex representing partition `p`. Its `data` member contains a set `vertices` of vertices of this component. Furthermore there are sets `in` and `out` of vertices which connect this component from/to another partition.

4.6.4 Instruction table import API

Instruction tables may be constructed via the following api:

```
operation& add_operation(opid_t op_id, tid_t out_type,
  vector<tid_t>& in_types, flag_t flags);
void operation::add_code(vector<int> widths_in, int width_out, string code,
  stringlist code_custom, string note, string tags, int rating);

type& add_type(typename T::tid_t t, int bitwidth = 0);
void type::add_code_type(int width, string code, string note);
void type::add_code_conversion(int width_in, int width_out,
  string code1, string code2, string code_custom,
  string code_generic, string note, string tags, int rating);

void add_expansion(opid_t op_id, string name, string transformer,
  vector<opid_t> args, string note, vector<tid_t> in_types,
  vector<tid_t> out_types);
```

Figure 4.18: Import API of instruction table.

`add_operation, add_type` look up the requested operation or type. If the object does not exist, it is created. Finally, reference to this object is returned, allowing insertion of child elements.

other fields are exactly the fields described by field semantics.

4.6.5 Graph transformer API

Api required from custom transformers is the following:

```
static string get_name();
void transform(G& generator, stringlist args);
```

Figure 4.19: Api of graph transformers.

Semantics of these fields are again the same. There are no restrictions on content of `args` and no restrictions on what the transformer is allowed to do with the generator or with its graph.

4.6.6 Tag handler API

Tag handlers are supposed to provide a single function:

```
bool is_satisfactory(string tag_list);
```

Figure 4.20: Api of tag handlers.

The tag_list is a string list of tags of an instruction or width conversion. The return value is Boolean which tells whether the tags satisfy *some* conditions. The tag structures may be defined by whoever wants to define them. Currently, we use one tag structure to pass command-line specified tags and another one to restrict instruction sets for the 'bobox' environment. Tag handlers are passed directly to the instruction table via a shared pointer. This is the only class in our framework which needs to use dynamic polymorphous.

4.6.7 Custom commands

Custom commands may be registered within the ctb class by the following method:

```
void register_command( string cmd, function<void(stringlist&&)> f,
  string description);
```

Figure 4.21: Command registration API.

cmd – the command string.

f – a function performing the command.

description – a short reference which will be shown in the '-h' option.

4.7 Testing

Error handling

All abnormalities that are encountered are reported either as errors or as warnings. Errors are thrown in form of a pair of a string and a bool. The former specifying what happened and a the latter specifying whether it was critical. These errors are caught in multiple try/catch handlers and re-thrown with more detailed context until the ctb class catches the error and writes it out. This way, a stack-like trace of ctb actions is printed with relevant context. Warnings are printed directly into the standard error stream. If more detailed information about such a warning is needed, ctb provides the -w option which makes ctb to process all warnings as errors (implying additional stack information).

Graph visualisation is supposed to be used for debug. All generator related errors are accompanied by visualisation of the graph in question unless the -g switch is used. Visualisation may also be invoked by the visualise command. The visualisation shows vertex names, operation identifiers, to_pos and from_pos annotations, order of edges (if assigned) and inferred types of edges.

Framework testing

Unit testing

Every unit of ctb contains a static `self_test` method. This method contains some scenarios which are supposed to ensure that everything works properly. All these methods are included by the `test.cpp` file and compiled as a separate executable.

Test scenarios

We also provide a number of scenarios which should ensure that the composed whole works as expected. Some of these scenarios test graph transformations, some test alias environments, some test implementation of macros, some test instruction tables, etc..

These tests are usually specified by means of a `program` file. Results are usually checked against remembered outputs which were verified by a human being. Since the order of some iteration methods of the C++ language is not stable across different compilers, some outputs may differ.

The test directories also serve as examples of usage of ctb.

Instruction table testing

For the purpose of testing of instruction tables, we provide the `testgraph` command, which generates a new graph such that:

- Multiple instances of every input operation are generated.
- An instance of every data-processing[5] operation is created. Inputs are connected to these operations so that different arguments always receive input from different inputs.
- An output operation is created for every output of a data-processing operation.

When writing a new instruction table, we usually wish to ensure two things:

- The syntactical correctness of regular instructions. ‘item The emantical correctness of regular instructions.

[5]Non-input-non-output operation.

Syntactical correctness may be ensured by means of a compilation of a code generated from the testing graph. However, experience shows that these graphs are of enormous size, and therefore their compilation may take very long time. As a solution, we provide a special option `-c`, which makes the compiler output *only* the first instruction per every vertex. For all other instructions, only a dummy variable is declared.

For testing of the semantic correctness of instruction tables, we have implemented the 'simu' environment. This environment outputs a fragment of code which constructs an input using a pseudo-random generator.[6] When the resulting c file is compiled and run, it repeatedly calls functions realising the graph by different widths and checks that all results are equal to the width 1 results. If a collision is encountered, the problematic fragment is invoked again with a break-point-like indicator set. This causes that a detailed log is generated during the iteration which observed the wrong behaviour. This log contains contents of sse registers generated via *debug* nodes.

The 'simu' environment is supposed to be used with the a graph generated by the `testgraph` command (mentioned above) to ensure correctness of the entire instruction set. Note that this scheme does not automatically ensure testing of all instructions and width conversions — the test needs to be performed with different widths and tags to ensure that all conversions and instructions get eventually generated. Also, input and output operations have to be available for all data types. This functionality resides in `loader_test.h` and `aliasenv_simu.h`.

We do not cover testing of special control-flow-related instructions since these require tailored testing scenarios.

Scenarios

Test directories of our framework are numbered. The following scenarios may be found in the project:

1 consists of the unit tests and some simple hard-coded generation. The input for this generation is based on sse but is not meant to provide functional output. This results are produced by the 'simple' environment.

2, 3 test that loads and exports of tables work properly.

4, 5 are 'simu' environment tests. Test 4 is a full test of the provided SSE table. Test 5 shows the same functionality on a simple graph and, unlike the test 4, is a part of the basic test chain. See the previous *Instruction table testing* subsection for more details.

6 provides an output in the form of Bobox boxes. This environment shows full implementation of preloads with respect to Bobox semantics of envelope usage.

[6]We use the mathematical model described in [17].

7, 8, 11 test the cycle removal algorithm, node expansion and type inference. These tests provide a 'visual' target, which shows what is going on (`>make visual`).

10 tests functionality of provided SSE buffer macros.

11 shows functionality of non-vectorised control flow on a simple C table with hard-coded input-output operations. This scenario uses the buffer macros with simple, non-vectorised variables.

12 shows the same functionality as test 11 with fully vectorised bodies but without full employment of sse related capabilities of the buffer.

13 shows the same functionality as tests 11 and 12, but uses fully vectorised buffers and the actual SSE table.

14 tests that type inference works with incomplete type information and that loops do not cause any problems.

15 tests that less common width conversion patterns work as expected.

16 is a matrix multiplication benchmark.

17 is a split instruction benchmark.

The test 9 was meant to test the C preprocessor implementation of a SSE buffer. This implementation may be found in the `sse_cf_macros` directory. If the reader is interested in advanced usage of the C preprocessor, he is welcome to look around. Macros for recursive computations were taken from [16].

We should also warn that the auxiliary files are often present duplicitly in the test directories. Our reason for that is that we wish to minimise confusion caused by non-explicit linking.

At this point, we would also like to remind the reader that most of these scenarios contain visual make targets as well as the `run.sh` script.

4.8 Available environments

There are four environments with generation capabilities. We will give a detailed description only for the 'Cf' environment since it is the only one which is not self-contained.

Simple

'Simple' environment was the first implemented environment. It prepares random data into arrays created according to the graph specification and runs the result for a fixed number of iterations. This environment handles only graphs consisting of regular operations.

Simu

'Simu' environment is an improvement of the 'simple' environment. This environment was designed for testing of the provided SSE instruction table.

Bobox

'Bobox' environment provides functionality similar to the simple environment except for two things. First, its API is API of the Bobox environment and second, it is tailored to utilise the capabilities of aligned load and store operations. Besides the generator-related functionality, this environment also generates code which manages the envelope system of the Bobox environment. With hindsight, we have to admit that this API layer should rather have been abstracted entirely outside our framework.

Cf

'Cf' environment implements Algorithm 7. This environment already abstracts its API and fully uses the SSE instruction set.

The output file consists of four functions (discussed further in Section 5.2) enclosed into a class with two templated context types (this is shown in Figure 4.22). These four functions take a number specifying how many rows are to be processed and two context structures which provide arrays of data, indices and offsets. The required context structure API is shown in Figure 4.23.

```
#include "macros.h"
template<typename CTXIN, typename CTXOUT>
class box_<name>
{
  void process_single    (int, CTXIN&, CTXOUT&){...};
  void process_aligned   (int, CTXIN&, CTXOUT&){...};
  void process_unaligned (int, CTXIN&, CTXOUT&){...};
  void process_shifted   (int, CTXIN&, CTXOUT&){...};
}
#define IN_INDICES_<name> <values>
#define IN_TYPES_<name> <values>
#define IN_TYPES_IL_<name> <values>
#define OUT_INDICES_<name> <values>
#define OUT_TYPES_<name> <values>
#define OUT_TYPES_IL_<name> <values>
#ifdef RUN_BOX
  RUN_BOX(box_<name>, IN_INDICES_<name>, IN_TYPES_<name>...);
#endif
```

Figure 4.22: API of the resulting code of the 'cf' environment.

```
struct context
{
  <type1>* data_0; <type2> *data_1; ...
  int index_0;     int index_1;  ...
  int offset_0;    int offset_1; ...
}
```

Figure 4.23: API of the context for the 'cf' environment.

The alias `input`, which is meant to represent storage area in instruction patterns, is set to `"ictx.data_$ioindex[ictx.index_$ioindex+$iindex]"`. As we have suggested earlier, the `ioindex` value is supposed to be passed as a vertex parameter of input/output nodes. The `iindex` is provided by generator's context. Indices are supposed to be incremented by the entire packsize at ends of code of partitions — this way we do not introduce new dependencies between load instructions. Indices may be simply incremented by adding the `"$inputinc"` pattern to the `increments` pattern layer. The alias `inputinc` is set to `"ictx.offset_$ioindex+=$packsize;"`. Aliases `output` and `outputinc` are defined analogically.

The following writer layers are used for the purpose of instruction-code generation:

`global` gets outputted into the top-level scope of the processing function. Suitable for injection of testing values.

`default` layer serves for the standard output.

`preload` layer has the `once` flag turned on and is directed to the default layer, meaning that this layer is generated before the `default` layer while the topological ordering of all instructions is preserved.

`shiftacum` and `shiftacumpreload` are analogical to the pair of `default` and `preload`. The difference is that this layer gets outputted into a different layer which is handled specially. Namely, when the `EXPANDSHIFT` alias is encountered in any output layer, the content of the accumulator is printed into a switch pattern which realises shifted loads. The resulting switch gets printed into the default layer. The alias which is used to switch different paths is `offset`. The alias which may be used in instruction patterns for the constant version of the `offset` is `constoffset`.

`increments` has the `once` flag set to true and is outputted after any code of the partition. It is meant to contain index increments and therefore the load and store instructions are supposed to output either the `"$inputinc"` or the `"$outputinc"` pattern into this layer.

Also, there are seven macro constructs which are designed to enable easy automatic management of the resulting box. Namely these contain:

`IN_INDICES_<name>` set to coma separated list of input indices.

`IN_TYPES_<name>` set to coma separated list of input types.

`IN_TYPES_IL_<name>` set to coma separated list of input types interleaved by dummy types in index order. This means that if this list is zipped with the sequence `0, 1, 2, 3...` then types correspond to their `ioindex`es.

`OUT_*` - analogical macros for outputs.

`RUN_BOX` macro expansion is performed if the macro is defined. This macro may be simply included into the resulting code fragment by means of the inclusion of the `macros.h` file. This file is, by the way, supposed to be available but is not required to contain anything specific. In that case, the macro may use the defined index and type lists to construct appropriate context types and to pass these into some processing function. Such function is then free to fill the two contexts with data and use the generated box class as it finds appropriate.

We also provide a sample context template (`context.h`). This template provides storage for the required fields, basic methods for offset calculation and also accumulator lists. The lists allow generic processing of these contexts (such as preparation of input data and retrieval of the processed data). In other words, any processing environment (such as Bobox) can be simply interfaced by a single file containing a single *generic* algorithm. There is no need to generate an environment-specific interface for every outputted fragment.

Regarding the context `struct`, we do realise that the creation of a separate variable per every input is awkward. The context structure may be designed in much more elegant fashion by means of variadic templates. Still, we do not feel appropriate to obscure this structure by overusing the functionality which is often seen as a subdomain of dark magic. In other words, we have decided to provide this structure for its simplicity and easily understandable interface. This decision is supported by the fact that this structure is most likely the first piece of our code which any user of our framework will come into contact with. Still, as we have mentioned above, our template does allow generic processing of data.

An arbitrary context structure may be used with the resulting code fragment, but it is bound to provide the API from Figure 4.23. If the user wishes to define his own context structure with a more elegant API (and without our API), he is welcome to do so. In this case the user will surely want to override the `input` and `output` aliases. He may do so by creating a new alias environment and by forwarding the generation call to this new environment. The necessary steps are the following ones:

1. Create a new alias environment.

2. Define its `get_name` method.

3. Define its `alias` method. Optionally define any aliases whose use in instruction patterns is desired. For instance, `input`, `output`, `inputinc`, `outputinc`, `offset` aliases. Note that this step effectively overrides any aliases which may be defined in any lower layer.

4. Define its `generate` method. Make this method forward its argument to the corresponding method of the 'cf' environment. Use the new alias environment as a template argument of this call (this will apply the new aliases by specifying the new alias environment as the root environment of alias expansion).

5. Register the new alias environment within ctb. E.g., modify the `fill` method in the `ctb.h` file or duplicate the `main.cpp` file, adding a registration call between initialisation of the ctb object and the processing command. This step may be skipped unless you want to control ctb via textual commands.

6. Use the new environment instead of the 'cf' environment.

4.9 Implementation of new environments

Implementation of an entirely new environment, e.g., providing support for an entirely new language, a SSE extension and a target distributed environment, may require the following steps:

1. Writing of a new instruction table.

2. Implementation of a new alias environment.

3. Implementation of a new language formatter, unless you are fine with non-formatted output files. In that case feel free to use the `language_empty` structure.

4. If a new control flow strategy or a new type of control flow node is to be introduced, a new graph transformer may be also necessary.

The first two things which the user should think about are:

- Representation of vectorised data types. All elements need to retain the correct order when being split or merged.

- Exact strategy of employment of control flow.

5. Implementation of the SSE instruction set

We will skip any broader introduction of the SSE extension, since deeper knowledge of the actual instruction set is not necessary for the purpose of this text. A very brief and very informal introduction to the SSE extension may be found in [18]. SSE reference documentation is provided by [19]. A handy reference is the cheat-sheet available at [20]. Paper which discusses more advanced aspects of efficient use of the SSE (and similar) extensions is [7]. We provide only the necessary conceptual minimum of the SSE type system.

We will omit the `_mm_` prefixes and other non-substantial information contained in function names.

5.1 Data types and conversions

SSE type system

SSE provides a number of 128 bit registers, which may be directly used by means of the `_m128` type. Such register can contain 128/*Length* values of signed or unsigned integers of lengths from 8 to 64, single-precision floating point numbers and double-precision floating point numbers. Boolean values are supposed to be represented by masks whose bits are either all 1 or 0.

The content of these registers is to be interpreted as a right-to-left list of values. Note that semantics of shifting operations make this point important. The reader may ask why is it to be this way, and rightly so — the reason is the clash of our customs of writing lists left-to-right while writing numbers right-to-left. Consistent, increasing order of elements is often desired for the purpose of width conversions and casts.

Our type system

As we have already mentioned, we provide the following data types:

- Signed and unsigned integers of widths 8, 16, 32, 64.
- Single-precision and double-precision floating point numbers.
- Byte bools, which are suitable for use with control flow.
- Bit Boolean vectors, which are suitable for high throughput computations without control flow.

We use the simplest and most obvious way of arrangement of data in SSE registers when a register is to be filled entirely. However, when two vectors of different widths of the stored data types are to be processed by one operation, we need to select only some data from the vector of the narrower data type width. Broadly speaking, we always use positions in their intuitive order which are multiples of $128/n$ where n stands for the number of data elements we want to store into a single register.

Consider the following example of a 16 bit integer width conversion from width 8 to 4 (numbers represent order of elements).

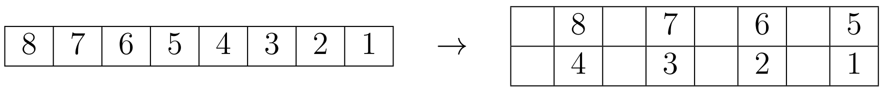


Figure 5.1: Width conversion realised on SSE registers.

This way, elements stay always aligned with other elements from the same data row. Such vector may enter a computation with another vector consisting of four 16-bit integers. But it may as well enter a computation with a vector of four 32-bit integers. This property is especially handy with bools if we edit Boolean splits to copy the value into both fields. Visually this means performing the previous split as follows:

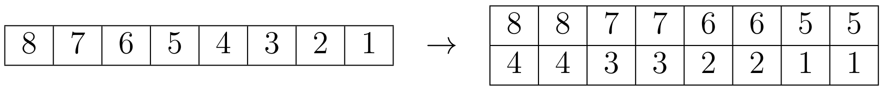


Figure 5.2: Width conversion suitable for bools.

Notice that to perform this split, a nontrivial data reordering was necessary. A simpler approach would simply select every second element by an *and* mask and then put it into a new vector. Showing this by a pseudo-vector C operation, we would do:

```
int result1 = (input & 0x00FF00FF);
int result2 = (input & 0xFF00FF00) >> 8;
```

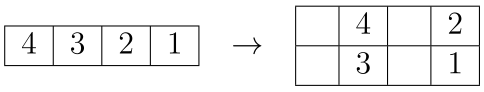


Figure 5.3: Example of a possibly malfunctioning width conversion.

The results are still aligned. The problem with this operation comes when two vectors of different base width come into contact. If we wanted to sum the vectors from Figure 5.3 with a vector of 16-bit integers (the original width of `input` was 8-bit), we would get an element-wise sum of the vector $((4, 2), (3, 1))$ with the vector $((4, 3), (2, 1))$. Of course, this problem may be further addressed by proper setting of a fixed ordering per every data width. The technical difficulty is that all load, store, split and merge operations would have to shuffle data in a rather nontrivial fashion.

Width conversions (as described above) may be implemented by means of the set of `shuffle` instructions provided by SSE. We provide an excerpt dedicated to shuffle macros from the SSE documentation [19] to portray its semantics:

“ The Streaming SIMD Extensions (SSE) provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the SHUFPS instruction. You can view the four integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word. ”[19]

5.2 Considering load and store problems

Most of our implementation assumes to receive its input by means of more-or-less continuous arrays (e.g., by a sequence of long arrays per every input). The SSE extension provides fast aligned load and store operations whose use is indeed desirable for this purpose. The aligned load/store operations load vectors of values from continuous memory areas which are required to begin at multiples of 16 bytes (128 bits).

So, ideally, we would like all these arrays aligned to 16 bytes and be of lengths of multiples of 16 bytes. However, the world is not always ideal, so we dedicate this section to alternative approaches. Besides the fast loads, the extension also provides the `alignr` operation. This operation takes two SSE registers and an *offset*, concatenates the two registers and selects a new (continuous) SSE register shifted by *offset*.

The `alignr` operation allows implementation of an efficient non-aligned load. This works by preloading the first aligned area into an auxiliary variable and then chaining a sequence of load operations through the `alignr` operation (as shown in Figure 5.4). One problem is still present — the last argument of `alignr` has to be a compile-time constant, which means that either all offsets have to be known or that we have to switch over all possible offsets. We implement the latter option per every operation. Stores may be implemented analogically.

```
_m128 aux1, aux2;
_m128 aux1 = load(input[j-offset]);

aux2 = load(input[j-offset+1*packsize]);
_m128 a1 = alignr(aux2, aux1, offset);
aux1 = aux2;

aux2 = load(input[j-offset+2*packsize]);
_m128 a2 = alignr(aux2, aux1, offset);
aux1 = aux2;

...
```

Figure 5.4: Example of an efficient non-aligned load of sequence of SSE registers.

input is an array of input data.

j represents the first index of the currently processed pack of data.

offset is the difference between the actual address of the required data and the preceding aligned address.

As the result, we propose generation of four different instances of the graph-processing code fragment (we have already shown these in Section 4.8). Use of these code fragments would then be switched during the processing in order to achieve an efficient processing of all the data. These fragments would be:

- Non-vectorised realisation.
- Vectorised realisation utilising unaligned load and store operations.
- Vectorised realisation utilising aligned load and store operations.
- Vectorised realisation utilising shifted load and store operations.

The vectorised realisations would be used on sequences of data rows such that all input and output arrays are continuous across these sequences. Aligned version would be used if possible, shifted or unaligned version would be used otherwise.

The non-vectorised version could be used to process preambles and epilogues around transitions of data containers. This means processing the last `offset` rows before an end of any of the containers as well as processing the first `packsize - offset` after beginning of any container. The former processes the last `offset` lines which cannot be loaded by the vectorised algorithm. The latter ensures that the rows processed by vectorised algorithm stay aligned with output containers. However, this depends on the actual algorithm - the 'bobox', 'simple' and 'simu' environment do need this (explicit) processing of preambles and epilogues but the 'cf' environment (Algorithm 7) does not, as long as all alignment-related conditions are satisfied.

5.3 Buffer macros

Working principle

We implement a register-stored SSE buffer by means of macros resolved by the 'cfmacros' alias environment. A buffer created by these macros consists of a number of circularly organised registers and two variables which denote the number of stored elements (`contains`) and the current reading position (`readat`). We also define some constants which allow us to simply retrieve information such as the total capacity (`capacity`) of the buffer.

Stores (pushes) are performed by means of a switch which uses the value `(readat + contains) % capacity` to determine a code fragment which inserts a single element at the correct position. We assume that the target C compiler optimises such switch into a single jump instruction with its address determined by the expression. Loads (pops/peeks) are implemented likewise but provide also vectorised version using the `alignr` instruction.

Of course, this scheme requires quite large numbers of SSE registers even for small graphs, while standard CPUs provide only 16 such registers. We assume this data to be stored mostly in the L1 cache and automatically loaded when needed. We leave spill code[1] optimisations to the target C compiler. The compiler may also be supplemented with profiling statistics for this purpose.

Buffer API

```
BUFFER_DECL(capacity, vsize, inpacksize, outpacksize, id, type)
BUFFER_SIZE(id)

BUFFER_FULL(id)
BUFFER_FULL_GRANULAR(id)

BUFFER_EMPTY_GRANULAR(id)
BUFFER_EMPTY(id)

BUFFER_PUSH_SIMPLE(capacity, vsize, id, val)
BUFFER_PUSH_ONE(capacity, vsize, id, typeabbrev, val)

BUFFER_CONSUME_ONE(id)
BUFFER_CONSUME_VECTOR(id)

BUFFER_PEEK_SIMPLE(capacity, vsize, id, to)
BUFFER_PEEK_ONE(capacity, vsize, id, typeabbrev, to)
BUFFER_PEEK_VECTOR(capacity, vsize, id, typeabbrev, to)
```

Figure 5.5: Buffer creation macros API reference.

[1]The actual machine load and store instructions added after the first iteration of scheduling and register allocation for variables which could not fit into the actual CPU registers.

`id` is an identifier of the buffer.

`capacity` is the total capacity of the buffer.

`vsize` (vector size) is the number of elements which can be stored into a single SSE register.

`typeabbrev` is the SSE type suffix.

`inpacksize and outpacksize` are the sizes of data packs which are processed by a single invocation of the partitions (again the w value of the two partitions the buffer connects). Current implementations assign this number uniformly.

`DECL` produces declarations of all variables realising the buffer. These are to be accumulated into a layer of a writer which goes into global scope (w.r.t. the code of partitions).

`SIZE` returns the number of elements actually stored in this container.

`FULL, EMPTY` resolve to expression which determine whether the buffer is full or empty. The `GRANULAR` versions take the `inpacksize` and `outpacksize` into account, meaning whether an entire vector of values produced by a vectorised partition code can fit into the buffer.

`PUSH` adds a value into the container.

`CONSUME` increases the `readat` counter and decreases the `contains` counter, effectively removing the head element from the buffer.

`PEEK` allows reading of the head element (or vector).

`SIMPLE` versions of the `PUSH` and `PEEK` operations do not take the subvector structure into account. A simple variation of the buffer for simple types (such as `int`) may be constructed using these operations.

5.4 Implementation of splits and merges

As we have already said, the control flow is implemented according to Algorithm 7. For the purpose of generation of all the conditions, we allow operations to have defined an arbitrary number of instruction patterns. Results of these patterns are then accumulated to different writer layers, which are intended for specific chunks of code as shown in the diagram of Algorithm 7. It remains to discuss some details.

Every control flow node gets a unique identifier assigned as its parameter. (This is done during node expansion.) These ids are preserved during node expansion. Every control flow operation uses this id to construct identifiers for all buffers it requires. These buffers are then declared in one of the expanded nodes by the `DECL` macro and further used by other macros.

Buffer sizes are assigned into parameters with respect to edge position. This means that aliases (node parameters) like `buffercoefin1`, `buffercoefin2`, etc. are assigned by the size assignment algorithm (implemented as part of the cf graph transformer in `cf_transform.h`) and then used in macro expansions. This may be seen in Figure 3.10.

Also, every expanded control flow node adds a new label into an accumulator (writer layer) dedicated to labels of the current partition. This way, the expanded nodes maintain links to partitions which contain their data sinks/sources without needing any explicit information about identifiers of these partitions.

Both splits and merges are currently provided only in non-vectorised version – meaning that the actual split or merge of two streams is performed by a simple loop-like mechanism which always handles one record only. vectorised versions of control flow operations may be easily added once a vectorised version of the `BUFFER_PUSH` operation exists.

Despite our brevity, we have described what was worth describing. All other details are technicalities boiling down to the diagram of Algorithm 7. The actual implementation of control flow instructions may be found in `sse_set/src_cf.csv`. Corresponding graph transformations (node expansion, cycle removal and buffer size assignment algorithm) may be found in `cf_transform.h`.

5.5 Summary of the provided instruction table

The C instruction table which we provide may be found in the `sse_set` directory of ctb. It covers:

- Standard C operators in both vectorised and non-vectorised version.
- The discussed data types.
- Width conversions.
- Load and store operations in both aligned and unaligned versions.
- Split and merge operations. These are provided in different variations in directories of the corresponding tests.

The table has been tested by mechanism described in Section 4.7.

5.6 Benchmarks

Matrix multiplication

Scenario (test directory) 16 contains a benchmark of 2-dimensional integer matrix multiplication. This is a graph without any control flow. We used 10000 records. Time was measured by the `clock()` function. Results are shown in Figure 5.6.

The handwritten solution was implemented by nested loops, so its run-time may have been negatively affected by insufficient compiler optimisation. We ran compilation with the `-O3` flag.

The pack size was 4 since there were no instructions which could use higher vectorisation factor. Thus, load and store instructions were vectorised with factor 4 while multiplications were vectorised with factor 2. (The SSE set does not provide multiplication with higher vectorisation factor than 2.) Also note that the shifted test took significantly longer time due to the cost of its offset switch and an additional preload instruction per every input node. The performance of the shifted version should converge towards efficiency of the aligned solution with increasing pack size (again the w number).

	handwritten	non-vectorised	aligned	unaligned	shifted
clang++	145	53	28	29	46
g++	124	56	28	30	39

Figure 5.6: Benchmark of two dimensional matrix multiplication (in CPU ticks).

Split instruction

Scenario 17 contains a benchmark of a graph consisting of a single instruction – a split instruction. This instruction splits a stream of 32-bit integers into two new streams. Since this graph contains bools, the pack size is 16. Other properties remain the same.

We do not provide a vectorised version of the split instruction, so we did not expect good performance. Also, the graph algorithm has to perform two stores and multiple width conversions while the handwritten version just loads and stores the variables without any penalty. Considering this, the results shown in Figure 5.7 are still reasonable ones.

	handwritten	non-vectorised	aligned	unaligned	shifted
clang++	41	150	166	171	166
g++	36	135	128	152	133

Figure 5.7: Benchmark of the non-vectorised split instruction (in CPU ticks).

6. Conclusion

We have provided basic formalism describing our problem (in Section 2.1). Furthermore, we have analysed the problem of flow graphs and provided some relevant observations and proofs. We have shown how basic blocks can be easily vectorised in the domain of our problem (Section 2.2). Moreover, we have shown that order-preserving processing of data is problematic in case of networks with branching (Section 2.3.2). We have also shown that order-preserving processing inhibits vectorisation of loops (Section 2.3.2). Also, we have attempted to provide a solution which would not preserve order of data (2.3.3). This attempt (which we understand as mostly successful) has shown that other problems arise when the ordering condition is dropped. Although not without significant overhead, we have provided an approach which promises fully vectorised processing of networks containing loops. There still remains an uninvestigated question of how exactly the out-of-order solution should be employed, how this solution should be optimised and how networks should be preprocessed.

Besides theory, we have provided a general framework which provides all important features for realisation of these networks. Our implementation handles vectorisation of networks without control flow (as discussed in Section 2.2) and also implements the basic order-preserving branching version of the problem with all necessary transformations (as described by sections 2.3.1 and 2.3.2). We have included many user-convenience and testing-related functionality such as testing commands, debugging reports utilising graph visualisations, generators of testing graphs and a wide variety of testing scenarios which serve also as usage examples. Also, our framework may be easily extended by new functionality and may be adapted to new languages, environments and approaches.

Additionally, we have provided an instruction table covering the C language via intrinsic SSE instructions (as described by Chapter 5). We have managed to test this table satisfyingly using the mechanism described in Section 4.7. We have also developed a family of mechanisms which allow reasonably efficient description and testing of instruction tables. This instruction table, despite being quite complete, is still rather an example of use than a guaranteed, finished and fully-optimised product. Despite thorough testing which employed automatically generated graphs covering the entire table (which currently consists of 1197 records), there still may be bugs. Also, there are optimisation reserves, especially in case of control flow which we have managed to provide only in non-vectorised versions due to reasons discussed in Section 5.3 .

Our conclusion is that this framework has been satisfyingly tested and that it produces reasonable results. (At least according to benchmarks presented in Section 5.6.)

Bibliography

[1] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. The bobox project parallelization framework and server for data processing. *Charles University in Prague, Technical Report*, 1, 2011. ISSN: 1613-0073.

[2] J. Matoušek and J. Nešetřil. *An Invitation to Discrete Mathematics.* 2 edition. Oxford University Press, Oxford, 2008. ISBN: 0198570422.

[3] Michal Brabec and David Bednárek. Procedural code representation in a flow graph. pages 89–100, 2015.

[4] M. Brabec and D. Bednárek. Hybrid flow graphs: Towards the transformation of sequential code into parallel pipelines. *ITAT,CEUR*, 1422, 2015. ISSN: 1613-0073.

[5] M. Brabec, D. Bednárek, and P. Malý. Transformation of pipeline stage algorithms to event-driven code. *CEUR Workshop Proceedings*, 1214, 2014. ISSN: 1613-0073.

[6] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2001. ISBN: 978-1558602861.

[7] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber. Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409 – 425, 2005.

[8] Jacqueline Chame Jaewook Shin, Mary Hall. Superword-level parallelism in the presence of control flow. *Proceedings of the International Symposium on Code Generation and Optimization(CGO'05)*, 2005. ISBN: 0-7695-2298-X/05.

[9] Jaewook Shin, Mary Hall, and Jacqueline Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocessors and Microsystems*, 33:235–243, 2009.

[10] K. Gilles. The semantics of a simple language for parallel programming. *Information Processing: Proceedings of the IFIP Congress*, 74:471–475, 1974.

[11] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, 1:1107–1123, 2008.

[12] Richard Bird. *Thinking Functionally with Haskell.* 2014. ISBN: 9781107452640.

[13] A. Schrijver. *Theory of linear and integer programming.* John Wiley, 1986. ISBN: 978-0-471-98232-6.

[14] Uday Kumar Reddy Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model.* 2008. ISBN: 978-0-549-76796-1.

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 3rd edition. 2006. ISBN: 978-0321462251.

[16] Github - cloak repository. https://github.com/pfultz2/Cloak.

[17] Stack overflow – c algorithm to produce same pseudo random number sequences from same seed on. http://stackoverflow.com/questions/15500621/c-c-algorithm-to-produce-same-pseudo-random-number-sequences-from-same-seed-on.

[18] Félix Abecassis. C++ - getting started with sse. http://felix.abecassis.me/2011/09/cpp-getting-started-with-sse/.

[19] *Intel Architecture Instruction Set Extensions Programming Reference.*

[20] Jan Finis. x86 intrinsics cheat sheet. http://db.in.tum.de/ finis/x86-intrin-cheatsheet-v2.2.pdf?lang=en.

List of Figures

Attachments

- Source code of our framework with examples of instruction tables and example implementations of alias environments should be distributed along with electronic version of this thesis. Plain git repository may be found at `http://www.ktweb.cz/ctb`. Browsable mirror is to be found at `http://www.github.com/kareltucek/ctb`. Description of directory structure may be found in Section 4.3.