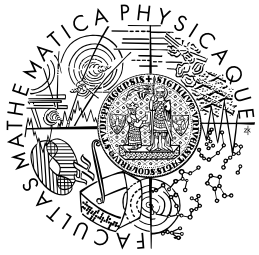

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Mgr. Michal Brabec

Procedural code integration in streaming environments

Department of Software Engineering

Supervisor of the doctoral thesis:	David Bednárek, Ph.D.
Study programme:	4I2, Softwarové systémy
Study branch:	Compilers

Prague 2017

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 12.12.2017 Mgr. Michal Brabec

Title: Procedural code integration in streaming environments

Author: Mgr. Michal Brabec

Department: Department of Software Engineering

Supervisor: David Bednárek, Ph.D.

Abstract: Streaming environments and similar parallel platforms are widely used in image, signal, or general data processing as means of achieving high performance. Unfortunately, they are often associated with domain specific programming languages, and thus hardly accessible for non-experts. In this work, we present a framework for transformation of a procedural code to a streaming application. We selected a restricted version of the C# language as the interface for our system, because it is widely taught and many programmers are familiar with it. This approach will allow creating streaming applications or their parts using a widely known imperative language instead of the intricate languages specific to streaming.

The transformation process is based on the Hybrid Flow Graph – a novel intermediate code which employs the streaming paradigm and can be further converted into streaming applications. The intermediate code shares the features and limitations of the streaming environments, while representing the applications without platform specific technical details, which allows us to use well known graph algorithms to work with the code efficiently.

In this work, we present the entire transformation process from the C# code the complete streaming application. This includes the management of the control flow, data flow, processing of arrays, method integration and optimizations necessary to produce efficient applications. Control flow represents the main difference between procedural code, driven by control flow constructs, and streaming environments, driven by data. We transform the code directly into the Hybrid Flow Graph and then we optimize the graph to introduce a structure better suited for the streaming environments. Finally we transform the graph into a streaming application.

We use procedurally generated code to verify the framework's correctness, where we test methods containing all combinations of loops, branches and serial code nested in each other. We also evaluate the performance of the produced applications and since the use of a streaming platform automatically enables parallelism and vectorization, we were able to demonstrate that the streaming applications generated by our method may outperform their original C# implementation.

Keywords: code transformation; intermediate code; parallel programming; vectorization; streaming systems;

I would like to give many thanks to all the people who helped me with this work and all the research leading up to it. First of all, I would like to thank my advisor David Bednárek, Ph.D. who helped me tremendously with my research, and his advice was indispensable for all my publications. I would also like to thank all my other colleagues from the university, who provided technical expertise and support.

Next, I would like to thank my family for understanding and support. Special thanks to my wife Klára, who believed in me all the way through.

Finally, I would like to thank my coworkers for creative environment, support and important feedback.

Contents

1. Introduction

Streaming environments form an important niche of parallel computing, originally designed for efficient implementation of image or digital signal processing. As these application domains interacted with other areas, particularly with databases, the idea of streaming environments developed from simple fixed-rate pipelines towards systems allowing arbitrary networks of operators with variable data rates. In addition, many parallel software systems resemble streaming environment, although they are not categorized as streaming.

Gradually, streaming environments became almost universal parallel computing platforms; however, no widely accepted standard emerged yet. Thus, each streaming platform often consists of a specialized language, its compiler, and one or more run-time environments, targeted at special hardware (like FPGA), GPUs, and/or general CPUs. An application is either constructed completely in a stream-specific language or it incorporates parts (*kernels*) written in a general language like C or VHDL.

With respect to performance, the specialized streaming languages may have important advantages including enforcing a particular (i.e. streaming) programming style and the absence of optimization disablers (like unrestricted memory access and aliasing) known from general-purpose procedural languages. On the other hand, programming streaming systems is a kind of rare expert knowledge. Moreover, the interaction between streaming and traditional code is often difficult; therefore, it might be advantageous to introduce a compiler able to create streaming applications from a procedural code.

Thus, allowing the use of a well-known general programming language for the design of streaming applications would allow the participation of non-experts who do not posses the knowledge of the required streaming language or the intricate details of the system interfaces. It would also make existing code available for the use in streaming systems.

In our research, we investigate whether modern general programming languages like C# may be used in streaming systems and, in particular, whether a compiler may convert non-parallel C# code into a network of operators which could be executed by a streaming runtime environment. Our research is targeted at streaming platforms implemented on general purpose CPUs and allowing variable data rates using a dynamic scheduler.

We wanted a language widely known among current programmers and researchers, which ruled out older languages like Fortran. We wanted a language with a comprehensive intermediate code so we could avoid the necessity to implement parser of the input language and this narrowed the choice to either Java or C#. We selected C#, because its intermediate language and entire environment is standardized and C# supports structures – object types with value semantics. Another advantage is the Cecil library which provides great tools for work with the C# intermediate code (CIL).

In shared-memory implementations of streaming systems, the overhead of data communication may be as low as writing and reading memory or cache. Thus, parallelism may be improved by decomposing an application into a large number of very small components, each comprising of only a handful or few

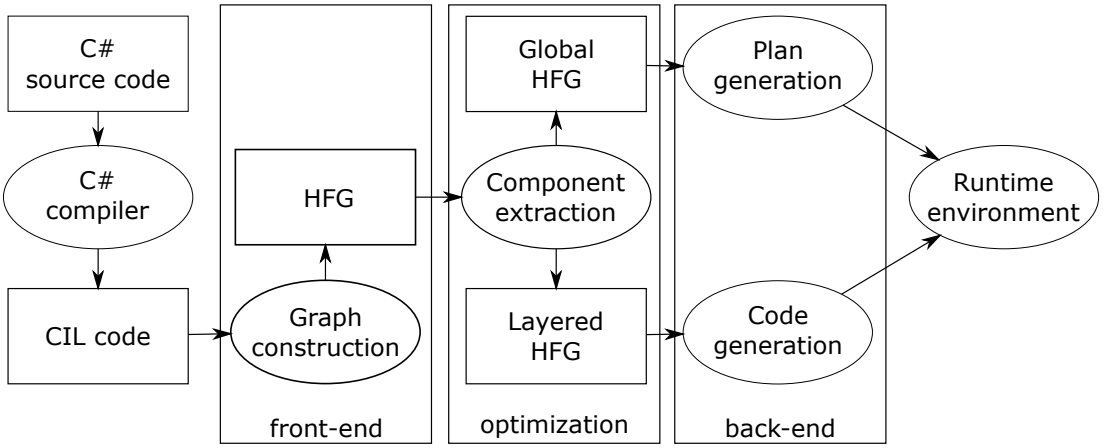

Figure 1.1: Compilation and code generation

dozens of arithmetic operations. This arrangement also naturally supports the use of vector operations.

Our target architecture is illustrated in Figure 1.1. In this case, a source code in C# is compiled (by a standard C# compiler) into the CIL intermediate code and then converted into the *Hybrid Flow Graph*(*HFG*) intermediate code [3]. While the CIL code is a classical intermediate code based on sequences of instructions, the HFG code assumes the form of a network of communicating operators; thus, it corresponds to the architecture of the run-time system, albeit in much finer granularity.

At this stage, the operators are equivalents of individual CIL instructions; therefore they are typically too small to become an efficient streaming operator. Consequently, in the next stage, the optimization technique called *component extraction* rearranges the original operators into small groups, producing a *layered HFG* containing a subgraph for each such group and a global HFG describing the interaction between the groups. For each component HFG, executable code corresponding to a streaming operator is generated. The global HFG is converted to a plan which defines the run-time connections (i.e. streams) between the operators.

In this work, we present theoretical basis of the *Hybrid Flow Graph* formalism, followed by the description of the *graph extraction* phase that converts a sequential intermediate code into the HFG. Next, we present the graph optimization – *component extraction* and its transformation into an application for the Bobox streaming system. Since the HFG follows the data-flow paradigm of the streaming systems, branches and loops must be converted from their sequential form into flows of control data.

Note that function calls have to be eliminated from the converted C# program by means of procedure integration. Of course, procedure integration is not possible in the presence of non-tail recursion and it may lead to unacceptable code expansion in some other cases; however, the same limitations typically apply also in streaming languages. In this work, we use a restricted version of the C# language, where we limit the use of referential data types (classes), assuming that value types (including structures) and arrays are predominantly used in typical streaming applications. The restrictions are described and explained in detail in Section 5.3. Consequently, while our approach is obviously not applicable as a

method of converting arbitrary C# programs for execution by streaming systems, it is applicable as a new interface for the streaming systems.

In order to implement a compiler capable of producing the HFG code, we had to implement multiple algorithms similar to those used in standard compilers, but modified for the HFG intermediate language. In most cases it was necessary to slightly modify the algorithms so that they reflect the HFG structure. We point out the algorithms in the sections focusing on related work, where we identify the original algorithm and the changes necessary for it to work in our environment.

To estimate the performance of the applications produced by our compiler, we measured the performance of several algorithms, including the Levenshtein distance, in their original C# implementation as well as when converted into streaming applications. Our measurements show that even the current prototype implementation of our compiler is able to generate streaming code, which is significantly faster than the original C# code. Additional optimizations might further improve the performance of the produced applications.

1.1 Motivation

Many streaming environments are well suited for execution of database-like queries. However; creating applications by mixing declarative languages, like SQL, and procedural components, like user-defined functions, is not easy in such environments. The following example shows where our proposed system could improve the situation:

```
select * from T1, T2 where T1.region = T2.region and
    DISTANCE( T1.name, T2.name) < 10
```

Listing 1.1: Sql query with a user-defined function

When implemented in a traditional database system, the query in Figure 1.1 contains a custom (user-defined) function $DISTANCE$, usually programmed in a platform-specific procedural language, like PL/SQL in Oracle or PL/pgSQL in PostgreSQL. In the database systems, the SQL part of the query is transformed into a highly optimized plan, where the $T1$ and $T2$ are initially joined based on their values and then the system applies the $DISTANCE$ function.

The SQL plan is usually executed in a specialized environment provided by the database engine, which contains optimized operators for all the standard SQL constructs. However; the execution of the custom functions requires a different environment, because they are implemented in a language with vastly different properties, which brings significant communication overhead. The matter is further complicated when the system tries to parallelize the custom functions, because they behave differently from the SQL operators and both can compete for resource.

One solution would be to transform the custom $DISTANCE$ function into an execution plan similar to that produced from SQL, connect both the plans together and execute them in the same environment. This solution would not be ideal, because both parts may have significantly different granularity. Therefore; parallelism will not be optimal due to different complexity and performance of components, but this approach eliminates communication overhead.

More advanced solution must incorporate decomposition of both the custom functions and the SQL operators and then it has to provide a mechanism to group smaller components into bigger blocks. This way the decomposed operators and functions can be grouped according to the structure of the entire query, thus reducing the communication overhead and providing more efficient parallelism. This approach can improve the overall structure of the query, because it bypasses the strict separation of custom functions and SQL operators and processes them all together.

The following two cases illustrate further benefits of the proposed solution:

1. The custom function can call other SQL commands. In this case, the advanced solution might be able to merge both the SQL plans and the custom function into a single query and optimize all its components together.

2. The custom function can be used to generate data. This might severely change the behavior of the query, because the custom functions are treated differently. However; the proposed approach decomposes custom functions and SQL operators and treats them the same, thus eliminating this problem.

It is important to note that in most cases, the custom functions are relatively small, without advanced constructs, like objects or exceptions. This allows us to restrict the input language while still providing sufficient tools for design of these functions.

This work provides the basis for the advanced solution presented in this section. The ParallaX compiler produced as the main part of this work is able to transform custom functions written in restricted C# to an intermediate language similar to the execution plans. Further work will be necessary to efficiently merge and optimize the produced plans, but the difficult transformation from the procedural code to the execution plan is solved in this work.

1.2 Objectives

Our main objective is to design an intermediate representation with properties similar to the streaming environments, which would allow us to work with the applications efficiently but without platform-specific limitations. This representation would also allow us to transform the applications for multiple environments using the same input language. To achieve this objective, we designed the Hybrid Flow Graph that is the main representation of the code in the compiler.

Our second objective is to facilitate the design of streaming applications to programmers without the domain specific knowledge of the streaming environment or its associated languages. To achieve this objective, we designed a compiler that transforms C# code, compiled to CIL, into applications for the Bobox streaming system [2].

1.3 Contributions

- We have designed the Hybrid Flow Graph, which can be used to represent applications in a way that they can be executed in streaming environments

or on other similar platforms. The Hybrid Flow Graph could be used as a common representation of applications for multiple different streaming environments.

- We have designed the compiler front-end capable of transforming a C# code, compiled to CIL, into the Hybrid Flow Graph. The compiler is able to transform code between two different programming paradigms and produce a representation suitable for streaming environments.
- We have designed a series of optimizations for the Hybrid Flow Graph, that improve its structure and behavior during execution.
- We have designed a compiler that transforms the Hybrid Flow Graph into an application for the Bobox system. The HFG structure is transformed into the domain-specific language Bobolang, while the separate kernels are transformed into C++. The entire application is then compiled with the Bobox runtime.
- We have modified algorithms and analyses, used in traditional compilers, for the HFG intermediate language. The algorithms rely on the same theoretical basis, but reflect the structural specifics of a graph-based intermediate language.

1.4 Text Structure

The rest of this work is organized as follows: After reviewing the general concepts of the streaming environments in Chapter 2, we present the well-known techniques and algorithms that form the basis of our work in Chapter 3. Next we define the structure of the Hybrid Flow Graphs in Section 4, especially their components related to control flow handling.

The next chapters define parts of the compiler for streaming environment that is the main contribution of this work. Chapter 5 sums the compiler's general concepts and structure, Chapter 6 explains the transformation of the C# source to the intermediate language – the Hybrid Flow Graph, Chapter 7 defines the HFG optimizations and Chapter 8 focuses on the HFG transformation into the application for the Bobox streaming system.

After all the compiler details are explained, we present alternative target environments supported by the compiler in Chapter 9. Chapter 10 presents our main case studies used to evaluate the performance of the applications produced by the compiler – the Levenshtein edit distance. Finally, section 11 illustrates the performance reachable using the complete compilation chain.

2. Streaming Environments

Stream processing is a programming paradigm that simplifies parallelization of some applications by restricting their structure[1]. Limiting the application architecture leads to better defined and more predictable behavior, which significantly enhances available parallelism and can help with scheduling, distribution and load balancing. This also means that the application can automatically use multiple CPUs and even heterogeneous processing units, such as graphical processors (GPUs) or field-programmable gate arrays (FPGAs), without explicitly managing memory or synchronization.

Streaming processing paradigm defines the application general architecture. An application designed according to this paradigm is constructed from the following components:

- *Stream* – a sequence of data.
- *Kernel*[2] – a series of operations (a function) applied to each element of its input streams.
- *Plan*[3] – an oriented graph, with nodes representing kernels and streams passed along edges.

The most important aspect of the streaming processing architecture is the fact that the kernels are able to communicate only by passing data to the streams connecting them. Kernels cannot communicate through the memory, network or external storage. This principle is crucial to exposing the available parallelism. It is possible to allow some kernels to read or write data to external storage to provide data for the application, but the locations should be independent, so the kernels do not influence one another.

Kernels can be either stateless or stateful depending on their semantics. The stateless kernels do not store any state information and their output always depends solely on the actual inputs. Stateful kernels maintain internal state, which can influence their output.

Figure 2.1 shows a *plan* of a simple *join* that merges two sets of values read from a data source (database) by the kernels *source*1 and *source*2. The join condition is implemented in the *merge* kernel. The plan declares kernels and their connections via streams, but it does not define the kernel implementation. Kernels are implemented separately as functions or objects, usually in a standard procedural language, and together with the plan they comprise the application.

The *start* kernel in Figure 2.1 initiates the application execution, since it is the only kernel without input. We use the start kernel to simplify the plan structure. The *source* kernels read the input from a data source (database or file), while the *print* kernel prints the result to the output (console).

[1] This chapter does not provide the analysis of all the systems that can be possibly categorized as streaming environments. Instead, we focus only on the systems best suited for the implementation of data-intensive application, like database queries.

[2] Alternatively called *Operator* or *Box*

[3] Alternatively called *Execution plan*

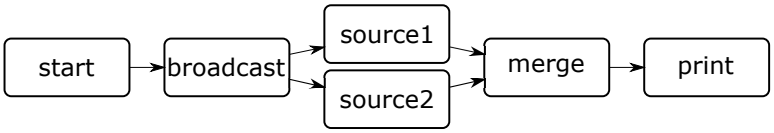

Figure 2.1: Simple join programmed for the Bobox streaming environment

A *streaming environment* is a system which is able to execute applications designed according to the streaming processing paradigm, also called *streaming applications*. The environment defines supported languages for programming the separate parts of the application. This usually means that the kernels are created in some general programming language, while the plan is defined in some special declarative language or constructed during runtime by explicit system calls.

Our selected target platform is the *Bobox* [2] streaming environment. It supports kernels implemented as C++ classes and for the execution plan, Bobox uses a special declarative language called Bobolang [3].

The rest of this chapter will discuss important aspects of streaming environments, with main focus on the Bobox system. We will explain how compatible applications are designed in Section 2.2. In Section 2.3, we will examine how these systems expose and exploit available parallelism and in Section 2.4, we will focus on factors that influence performance.

2.1 Related Work – Streaming Environments

This section presents other research related to the general concepts of this work, while the works related to the specific parts of our research are presented in later sections alongside the appropriate technique or algorithm. This approach means that the related works are discussed when relevant instead of being collected at the beginning without connection to the actual algorithm or concept.

The definition of a *streaming environment* or system is not strict, the label has been applied to a broad range of systems. The lower tier is occupied with systems designed for tight integration with hardware, often accompanied with circuit synthesis or specialized hardware.

The Brook system adds simple parallelization constructs to the C language, allowing the programmers to use the GPU as a streaming co-processor [4]. The Brook system provides extensions for the C language and a special runtime library. The applications are designed in the extended C, transformed into a code for the target environment (originally designed for DirectX and OpenGL) and compiled along with the runtime library.

Esterel is a language for programming synchronous parallel reactive systems [5]. A system designed in the Esterel language consists of multiple components (modules), which communicate through signals broadcast through the entire system. Each component defines the signals it recognizes and actions it can perform. The components can react to the signals by initiating or terminating actions or producing other signals. The language is the main programming environment for the experimental Kiel Esterel Processor [6].

The StreaMIT language was designed as a tool for efficient development of streaming applications [7]. The language adds streaming constructs to a restricted

version of the Java language and the programmers use the constructs to define the structure of the streaming applications (their plan, kernels and streams). The StreaMIT code is then compiled by a specialized compiler to a code for the target environment. The compiler is designed as an extension of the Kopi Java Compiler.

All these streaming systems rely on their own platform-specific languages, which contain special constructs necessary to define the structure of the streaming applications. This approach requires that the programmers construct the application structure by hand, which means that they have to know the properties of the constructs and target platform.

The field-programmable gate array (FPGA) can be considered a special case of low-level streaming hardware, where the separate programmable gates represent kernels. The FPGAs are traditionally programmed in a low-level hardware description language (HDL), but there are purely streaming technologies developed over it, like the Folding Streams algorithm [8], which generates the HDL code from a declarative implementation.

The upper tier of streaming systems consists of environments typically implemented using traditional CPUs which allow dynamic scheduling and variable data rates. The need for variable data rates often stems from their use related to databases, as in the case of Aurora [9], a system designed for data processing in monitoring applications. Aurora defines its own query algebra called SQuAl, where the structure of the query is declarative, while the operators are implemented in a procedural language similar to the Oracle PL/SQL.

STREAM [10] is a general streaming system with extensions focusing on data processing and databases. The STREAM system is programmed using a superset of the SQL language called CQL. This language provides additional constructs for the design of more efficient streaming applications, but it is heavily focused on database environment, with limited general application.

Streaming environments also involve wide-spread systems for big data processing, including application real-time database management [11] or image recognition [12]. The most influential technology of this field are MapReduce [13] and Apache Hadoop [14].

MapReduce is a distributed streaming system. The applications are constructed from the map and reduce operators, implemented in a well-known procedural or object-oriented language, like Java or C++ [15]. The applications must define how the data is allocated between the map and reduce operators, this is done by the partition and compare functions that assign and deliver the data from the map to the reduce operator.

Hadoop is a system for distributed scalable data processing and in essence it is a coarse-grained streaming system [16]. Hadoop applications are typically programmed in the Java language and the system supports multiple programming techniques, like MapReduce. Aside from Java programming, there is also the Pig Latin high-level programming language designed as an extension of the Appache Hadoop that provides more flexible tools for data processing [17].

For our experiments, we have chosen the Bobox system [2] which is well suited for database applications, but it is not limited to them. Bobox stream layouts (denoted *execution plans*) are defined using the Bobolang language [18], while the underlying operators are programmed in C++. The low-level details of the

```
operator main()->() {
  bobox::broadcast()->()[to_odd],()[to_even] broadcast;
  Source()->(int) odd(odd=true, length=100);
  Source()->(int) even(odd=false, length=100);
  Merge(int),(int)->(int) merge;
  Filter(int)->(int) filter;
  Print(int)->() print;

  input -> broadcast;
  broadcast[to_odd] -> odd;
  broadcast[to_even] -> even;
  odd -> [left]merge;
  even -> [right]merge;
  merge -> filter -> print -> output;
}
```

Listing 2.1: Boboland representation of the application introduced in Figure 2.1

system are hidden by our system and the restricted version of the C# language without any special streaming constructs.

2.2 Configuration and Programming

Bobox applications follow the architecture defined by the Streaming processing paradigm, they are constructed from kernels connected by streams. The structure is defined as an execution plan in *Bobolang* and separate kernels are implemented in C++. We will explain the basic principles on an application that joins values read from an external data source (database).

A Bobox application requires three main parts:

- *Plan* – an execution plan written in Bobolang.
- *Kernels* – an implementation of all kernels in C++.
- *Environment* – an object connecting the plan and the kernels.

The application plan defines its structure and available parallelism. The plan is defined in a special declarative language *Bobolang*, which is described in much detail in the following works [3], [19]. Bobolang has many advanced features, but its basic structure is intuitive and we will explain it on the example. A graphical representation of our example plan is in Figure 2.1 and its Bobolang representation is in Listing 2.1.

The Bobolang code defines the application as an operator called *main*. All the used kernels are declared in the first section, where each kernel is identified by a type, parameters and a name with parameters. For example, the merge kernel is of type *Merge*, it has two integer inputs and a single integer output and it is called *merge* in the rest of the code. The streams are defined in the second section, where each stream is represented by an arrow between the kernels it connects, along with a parameter name is connects. For example the stream

```
class merge_box : public basic_box {
public:
  typedef generic_model<merge_box> model;

  BOBOX_BOX_INPUTS_LIST(left, 0, right, 1);
  BOBOX_BOX_OUTPUTS_LIST(main, 0);

  merge_box(const box_parameters_pack &box_params)
    : basic_box(box_params)
  {}

  virtual void sync_body() override
  {
    input_stream<int> left(this, inputs::left());
    input_stream<int> right(this, inputs::right());
    output_stream<int> output(this, outputs::main());

    while (!left.eof() && !right.eof()) {
      int l = left.current()->get<0>();
      int r = right.current()->get<0>();
      if (l <= r) {
        output.next()->get<0>() = l;
        left.move_next();
      } else {
        output.next()->get<0>() = r;
        right.move_next();
      }
    }
  }
};
```

Listing 2.2: Merge kernel implemented in C++

between the *odd* and *merge* kernels is connected to the *left* input of the merge kernel.

Aside from the execution plan, we have to implement the necessary kernels. Some common kernels are already provided by the Bobox system, like the *bobox::broadcast* kernel that copies every single input value into all its inputs. Bobox kernels, also called *boxes*, are implemented as a C++ class that inherits the *basic_box* class provided by the Bobox system.

Listing 2.2 contains the implementation of the *Merge* kernel that sorts and merges the values from two input streams. The class first declares its inputs and outputs, constructor and model (a type used to reference the kernel from Bobolang). The main functionality is in method *sync_body*, that first links the inputs and outputs to objects and then iterates while the inputs contain data, merging the values to its output.

Last part of the application the *environment* object based on the class *runtime* provided again by the Bobox system. This environment links the plan with the kernels via a register, where all the kernel classes and data types must be

```
virtual void init_impl() override
{
  register_box<source_box::model>(box_tid_type("Source"));
  register_box<print_box::model>(box_tid_type("Print"));
  register_box<merge_box::model>(box_tid_type("Merge"));
  register_box<filter_box::model>(box_tid_type("Filter"));

  register_type<int>(type_tid_type("int"));
}
```

Listing 2.3: Kernel regisrtation section for the application

registered. The register is then used to execute the plan. The registration is done in the method *init_impl*, which is called prior to the application execution. The initialization for our join sort example is in Listing 2.3, where kernels and data types are registered with the name used in the Bobolang code.

2.3 Available Parallelism

The streaming processing paradigm exposes available parallelism by limiting the application structure to a graph of kernels able to communicate only through streams of data. All the kernels are therefore completely independent and can be executed in parallel with the streams as the only points of synchronization.

There are two sources of parallelism in streaming environments: multiple kernels executed in parallel and vectorization of the kernel code. Both approaches can be combined.

Kernels can be executed completely independently, because they communicate only through streams and their execution has no side effects. This parallelism is limited only by stream connections which represent synchronization and also introduce communication overhead.

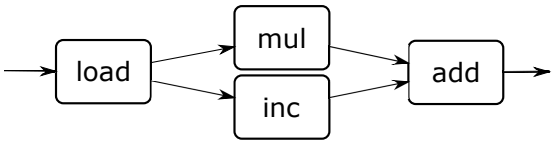

Figure 2.2: Kernel parallelism

Figure 2.2 shows a simple plan, where we can exploit two kinds of kernel parallelism. First, we can always execute kernels *mul* and *inc* in parallel, because they are not connected by a stream. Second, we can execute all kernels in parallel for different values in the input stream. Figure 2.3 shows the kernels executed in parallel for different values of the stream - 1, 2, 3, ...

Kernel code can be vectorized, because a single kernel is executed on a single processing unit, which is usually equipped with a vector unit (or co-processor), for example SSE or AVX in Intel processors. This unit can be used to parallelize code inside a single kernel further improving the application performance. It might be important to account for the actual number of vector units and their distribution between processors, but that is dependent on target platform.

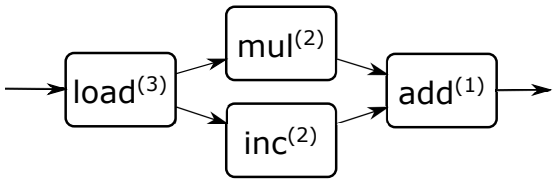

Figure 2.3: Kernels executed in parallel for the stream - 1, 2, 3, ...

Bobox prohibits the use of multi-threading inside a kernel, because the produced threads can compete for resources with the Bobox runtime and hinder the overall performance.

2.4 Performance Concerns

The most important factors that influence the efficiency of a streaming environment are: communication overhead and granularity. Both effects are connected, because the more kernels we have, the more available parallelism we can exploit, but the overhead is growing as well.

The communication overhead is caused by the data passing between kernels through streams. This overhead depends on the type of stream we are using. It is bigger for streams connecting kernels on different servers, but it is present even for kernels on a single multiprocessor, with memory streams. Granularity on the other hand directly influences available kernel parallelism, the more kernels we have, the more we can execute the application in parallel.

The overhead is always present and it is necessary to find the right balance between parallelism and communication. To maximize parallelism, we can make every instruction into a kernel, but then the overhead would overwhelm the benefits of the parallelism. Or we can eliminate the overhead by packing the entire application into a single kernel, which eliminates all kernel parallelism as well.

The right balance is usually different for each application and platform and it has to be individually assessed. This is one of the areas for our future work, because the ParallaX compiler currently supports only static platform-independent optimizations.

3. Common Techniques

In this chapter we provide context on the more complex well known techniques and systems used in this work. We provide basic description, with small examples, and links to other works that offer more detailed information. The chapter describes four important concepts necessary to understand our ParallaX compiler and the examples used in this work: *graph rewriting systems* in Section 3.1, *Kahn process networks* in Section 3.2, basics of the *Common intermediate language* code in Section 3.3 and an introduction to the SIMD instructions in Section 3.4.

3.1 Graph Rewriting Systems

A *graph rewriting system* is a grammar designed to transform graphs [20]. Similar to formal text grammars, the system consists of rules of the form $r : L \to R$, where L is the *pattern graph* and R is the *replacement graph* [21]. Rewriting systems can be defined for any type of graph, but in this work, we focus only on oriented annotated graphs, where the annotation is a string.

The semantics and application of the rewriting rules are defined algebraically using either the double-pushout or single-pushout approach. In the *double-pushout approach* [22], each rule is a pair of graph morphisms, where the first the defines nodes and edges to be deleted and the other defines nodes and edges to be added. In contrast, each rule in the *single-pushout approach* [23] is represented by a single morphism that applies the transformation in a single pushout operation. We use the single-pushout approach, because its rules are simpler and their application is more intuitive.

In the case of single-pushout approach, a rule is defined as a morphism $r : L \to R$, where all the nodes and edges affected by the replacement graph R must be matched by the pattern graph L [24]. The pattern graph can contain *empty nodes*, which do not have specified annotation and can match any node. To make the semantics and examples more readable, we use a simplified notation, where we omit the empty nodes.

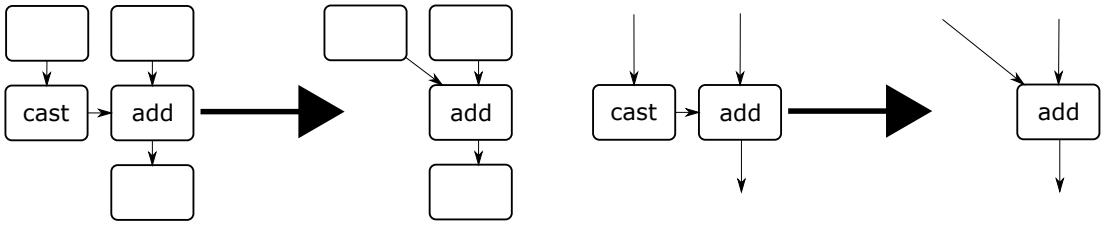

Figure 3.1: Simple graph rewriting rules.

Figure 3.1 shows two rewriting rules, the one on the left is a standard graph rewriting rule and the right one is simplified. The nodes without annotation are the *empty nodes* that can match a node with any annotation. In the simplified rule, we remove the empty nodes and leave only the edges connecting them to signify where the empty nodes are.

The rules are applied to an input graph one at a time while there is at least one that matches. The application can be nondeterministic when multiple rules

are applicable at the same time. The rewriting system must specify what rule should be selected in case nondeterminism, usually according to their order or priority.

The rule application itself is very intuitive, the matched subgraph is replaced by the replacement. The full theoretical basis of this process is described in detail in the following works [22, 20, 24], but an intuitive understanding is sufficient to follow the examples.

3.1.1 Associative Graph Rewriting Systems

The original rewriting systems support empty nodes in their pattern graph (L in rules $r : L \to R$), which can match any node [21]. We had to extend the basic behavior, because our optimization component requires more control over the matching process. The *associative graph rewriting systems* provide enough control to implement optimizations used in our compiler.

An *associative graph rewriting system* is a graph rewriting system, where the pattern graph can contain *group nodes*, *repeat nodes* and *chain nodes* with behavior similar to regular expressions.

A *group node* is defined by a list of allowed annotations, similar to the regular expression "[*abc*]", where the empty node is basically a special version of the group node, equivalent to the expression ".".

A *repeat node*, declared with a + symbol, defines a set of nodes that share the same connections but they do not have to be directly connected among themselves, this construct does not have any regular expression equivalent.

A *chain node* is defined as "a^*", where we require that the repeated nodes are connected as a single string matching the annotation without any outgoing or incoming edges besides the first and last. The rewriting system also allows the use of special nodes $\$N$ in the product graphs, which represent the subgraphs matched by the group or chain nodes. The behavior is best shown on an example.

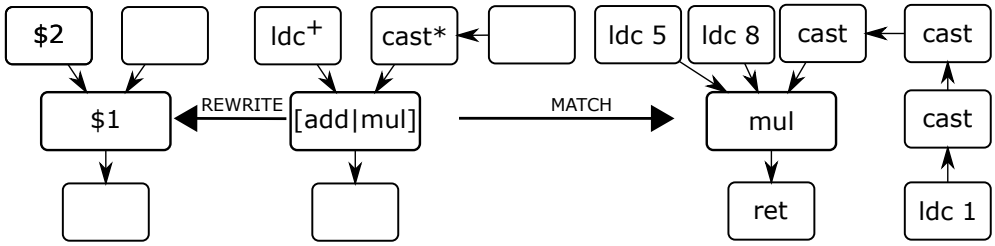

Figure 3.2: Regex pattern graph match

Figure 3.2 shows a pattern graph matching an annotated directed graph. The pattern graph contains a group node [*add*|*mul*], which matches any node with either the *add* or *mul* annotation. The empty nodes match a node with any annotation and the chain node $cast^*$ matches any number od *cast* nodes connected one after another. The chain node would not match the graph, if there was another node connected between the casts. The group and repeat nodes can be combined, like $[add|mul]^*$. The repeat node matches any number of *ldc* nodes connected to the central group node.

Besides matching the pattern, an associative graph rewriting system must also produces an associative collection connecting the matched and pattern nodes.

With it, it is possible to match the pattern and then modify the nodes based on how they were matched.

3.1.2 Graph Rewriting System Use Cases

We use the graph rewriting systems for multiple different purposes that are not necessarily connected, besides their graph-based data structures. The work contains the following use cases:

Intermediate Language Semantics

We use a graph rewriting system to define the semantics of our intermediate language. The system defines all the operations available in the language and it is part of the language definition.

Source Code Emulation

We developed a graph rewriting system that is able to emulate the calculation of the source code (C# code compiled to CIL code) and it is an instrumental part of our data flow analysis. The system is used solely for CIL emulation and has no relation to the intermediate language.

Optimizations

The optimization component of our compiler is implemented as an associative graph rewriting system, where each optimization is represented by a single rule. Although the rewriting system modifies the intermediate code, the optimization component does not change the semantics of the code.

Component Extraction

The component extraction is a special optimization step that groups parts of the intermediate code into bigger, more efficient components. The behavior of each new component is defined by the subgraph it represents, we do not add new rules to the graph rewriting system defining the intermediate language behavior.

3.2 Kahn Process Networks

The *Kahn process networks* is a distributed model of computation, where independent serial processes communicate through unlimited serial queues (FIFO) [25]. The processes read and write atomic data elements (tokens) from and to the queues and they cannot communicate with one another outside of the queues. The entire modeled system is deterministic, despite synchronization and parallel execution of the processes. Figure 3.3 shows a small part of a Kahn process network, where the arrows represent queues transferring data and the circles represent processes.

The Kahn process networks were originally designed as a model of computation for distributed systems [26], but they are also used to model embedded

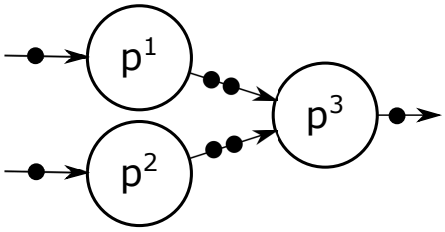

Figure 3.3: Part of a Kahn process network

system [27], signal processing [28] and they can be even used to replace streaming environments for some applications [29].

The Kahn process networks are similar to the streaming environments, presented in Chapter 2, where processes are an analogy to the kernels and queues to the streams. Kahn process networks, though purely theoretical, can be considered a subset of streaming environments, because they have similar semantics. The main difference is that the Kahn process networks require processes with deterministic behavior[1], which is not necessary, in the streaming environments.

We do not use the Kahn process networks directly in this work, but the intermediate language we designed as part of our compiler satisfies the requirements and can be considered a KPN. This means that its calculation is deterministic and it does not produce any unpredictable results.

3.3 CIL Basics

We selected the *C# programming language* as the interface for our ParallaX compiler, because it is widely known, the language and its entire ecosystem (the Common language runtime) is standardized [30, 31] and there are advanced utilities available for it. Our system processes C# applications compiled to CIL code, which is the intermediate language of the *Common language runtime*, the process is shown in Figure 3.4. Therefore, a basic understanding of the CIL instructions and inner workings is useful, because most of the examples used in this work use the CIL notation or rely on its structure or invariants.

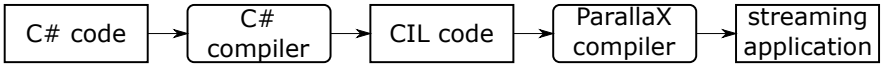

Figure 3.4: ParallaX compiler interface overview

3.3.1 Common Language Runtime

The *Common language runtime*, or *CLR* for short, is and application ecosystem, which allows multiple high level programming languages to be used interchangeably and platform independently. The infrastructure is standardized under ISO and ECMA [31] and specifies the execution environment, libraries and structure necessary for applications to work. Figure 3.5 shows the CLR architecture overview.

[1]The process always produces the same results in the same order for the same inputs.

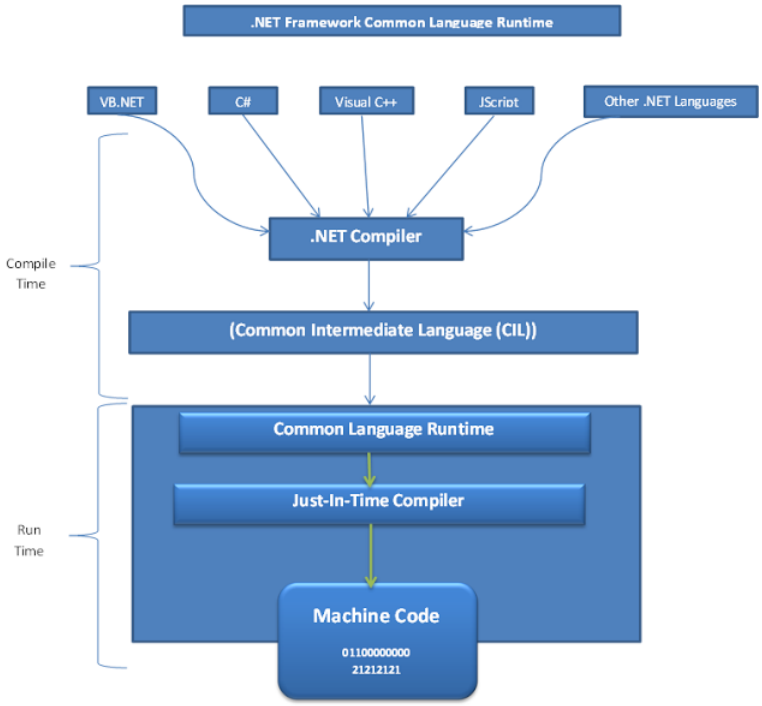

Figure 3.5: Common language runtime architecture

The Common language runtime defines a single intermediate language called the *Common intermediate language*, or *CIL* for short, along with a virtual machine able to execute it. CLR also specifies multiple higher level languages, like C#, F#, Visual Basic and others, which are all compiled into the Common intermediate language and can therefore be combined. The CLR standard is platform independent and there are already multiple systems compliant with it, like the Microsoft .NET [32], Mono [33] and the newly released .NET Core.

3.3.2 Common Intermediate Language

The *Common intermediate language* (*CIL*) is a stack-based, object-oriented assembly language. Each statement in the language is a single instruction and the instructions communicate only through a common stack, instead of registers used in other assembly languages. The CIL stack can store only numbers and references, and the instructions must interpret the values according to their semantics. Besides the stack, the CIL machine has access to local variables and method arguments, which can be changed and read repeatedly. Figure 3.6 shows a schema of the CIL virtual machine, which reads code and stores the values to the variables and stack.

To understand the examples, it is important to explain the basic mnemonics of the CIL instructions. The following list contains the most commonly used instruction codes and their explanations, where the code is basically an abbreviation of the instruction function. Each instruction takes its parameters from the top of the stack puts its results back on the top of the stack.

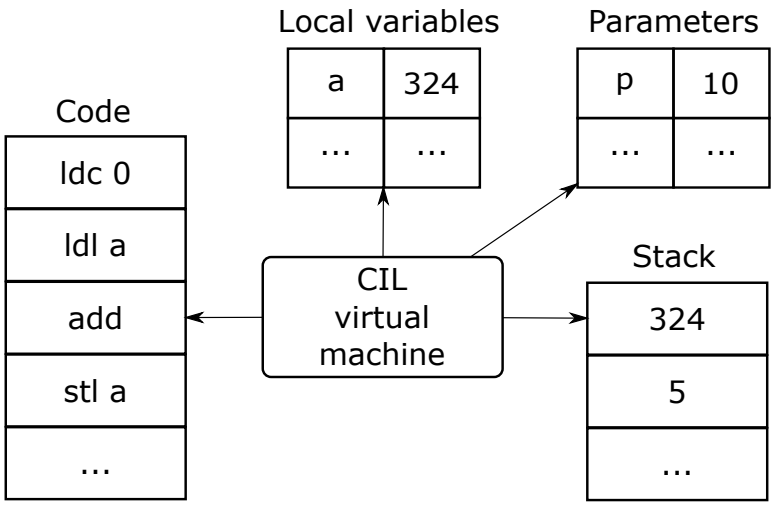

Figure 3.6: CIL virtual machine

- *ldc* – *load constant* written directly in the code
- *ldarg*, *starg* – *load argument*, *store argument*
- *ldloc*, *stloc* – *load local variable*, *store local variable*
- *mul*, *add*, *div*, ... – arithmetical operations (*multiply*, *add*, *divide*, ...)
- *blt*, *bge*, ... – conditional jumps (*brunch less than*, *branch grater than*, ...)
- *ret* – *return* method result

The CIL instruction set contains specialized instruction versions defined by suffixes. For example *ldc.i*4 0 is an instruction that loads zero to the stack as a four-byte integral number. There is also *ldc.i*8 0 and *ldc.r*8 0, for an eight-byte integer and a double precision floating point number. We usually omit these specialized versions, because they specify data types and we store data types separately.

Aside from the data type specialization, there are also short version of instructions that access first few parameters and local variables. These instructions have the suffix identifying the index of the variable or argument, like *stloc*.0 that stores a value into the first local variable and is the short version of the *stloc C* instruction for the code in Listing 3.1. The most commonly used short instructions are *ldloc.N*, *stloc.N*, *ldarg.N*, *starg.N* and *ldc.N*, where N represents the constant value.

Listing 3.1 shows a *minimum* function implemented in C# as a static method and Listing 3.2 shows the same method compiled into CIL. The CIL code contains only the instructions listed above and we added comments that help separate parts of the code to better match it to the original C# code.

```
public static int minimum(int A, int B)
{
  int C;
  if (A < B)
    C = A;
  else
    C = B;
  return C;
}
```

Listing 3.1: Minimum method implemented in C#

3.4 SIMD Instructions

The principles of the vector instructions are important for the algorithm structure and in this section, we revise the instructions and memory model fundamentals with particular emphasis on the aspects important for the studied problem. We will explain vectorization on the Intel / AMD SIMD extensions, mainly SSE.

3.4.1 Vectorization

The *Streaming SIMD Extensions* (SSE) is a vector instruction set extension to the x86 and x64 architecture, designed by Intel and introduced in 1999 as a successor to the older MMX extension. SSE provides new instructions that perform vector calculations on 128 bit vectors, stored in newly added registers. SIMD instructions can increase performance for algorithms that can keep the data in the special registers and perform multiple instructions before moving then back into the memory. The SSE instructions follow the Single Instruction Multiple Data or SIMD model, where a single operation is applied to all elements of a vector at once.

The original SSE was updated to SSE2 and SSE3, with each adding new instructions, and introducing new data types available in the 128 bit vectors. The original SSE supported mostly single precision floating point numbers and the later versions added double precision and integral numbers of differing sizes (8, 16 and 32 bits). The SSE extension was further expanded by the AVX and AVX2 expansions adding 256 bit vectors and new instruction and AVX-512 introducing 512 bit vectors.

The vector instructions have similar requirements as the GPU, which was the original target platform for the blocked Levenshtein distance algorithm. The similarities allow us to adapt the algorithm for our compiler without fundamental changes.

3.4.2 SIMD Instructions Types

SSE and its updates provide vector instructions similar to the arithmetic instructions available in CIL, but instead of a single number the instructions vork with a whole vector of values. The SIMD instructions are divided into multiple categories:

- Arithmetic operations – instructions that perform the vector calculations.

```
.method public hidebysig static int32 minimum (
  int32 A, int32 B) cil managed
{
  .maxstack 2
  .locals init ( [0] int32 C )

  // initialization of C to 0
  IL_0000: ldc.i4.0
  IL_0001: stloc.0
  // if condition (A < B)
  IL_0002: ldarg.0
  IL_0003: ldarg.1
  IL_0004: bge.s IL_000a
  // true branch (C = A;)
  IL_0006: ldarg.0
  IL_0007: stloc.0
  IL_0008: br.s IL_000c
  // false branch (C = B;)
  IL_000a: ldarg.1
  IL_000b: stloc.0
  // return C
  IL_000c: ldloc.0
  IL_000d: ret
} // end of method FlowGraphTest::minimum
```

Listing 3.2: Minimum method defined in Listing 3.1 compiled to CIL

- Logic operations – instructions that perform logical operations on the vector values.
- Memory management – instructions that transfer the data from or to the memory.
- Shuffle operations – instructions that efficiently exchange data in the vector registers.
- Miscellaneous – additional instructions for initialization and configuration of the SIMD unit.

The arithmetic operations are further categorized based on the data type they support. Each instruction interprets the registers as a vector of different values and using incompatible instructions can lead to invalid values.

3.4.3 Memory Organization

The original SSE supported eight new 128 bit registers which was extended to sixteen with the introduction of the 64-bit architecture. The AVX extension added sixteen new 256 bit registers. These registers are dedicated to the vector extensions and they are not available to the CPU for standard instructions.

The vectors must be filled by special SSE or AVX instructions, which either move data from memory or fill the vector with a constant. Moving data to the registers can be costly operation, especially when performed frequently. The applications should reuse the data in registers as much as possible, which means that any compiler producing the SIMD instructions must keep the *live variables* [4] in the registers while they are needed.

The memory transfers to the SSE registers are limited to the data aligned to 16 bytes [2]. The unaligned data can be transfered as well, since SSE2, but the transfer is much slower than for the aligned data. To mitigate this effect, the ParallaX compiler produces code that automatically allocates all the vector data aligned to 16 using a custom memory manager.

3.4.4 SIMD in C++

The SIMD instructions are accessed via intrinsic functions provided by the C++ compiler or its library. The intrinsics are provided as simple functions, where each internally represents a single SSE or AVX instruction, and each call is replaced by the instruction (inlined). The ParallaX compiler produces C++ code containing the intrinsic functions and the C++ compiler produces the final code with the SIMD instructions.

The special registers are represented by special data types, like the $\_\_m128i$ type representing four 32 bit integers. The registers are used by declaring a variable of the data type.

[2] Address is divisible by 16

4. Hybrid Flow Graph

The *Hybrid Flow Graph*, or *HFG* for short, is a compact format designed for representation of procedural code with operational semantics similar to the behavior of streaming systems. All the data types, control flow and data flow are represented by a single graph, unlike the *control-flow graphs* [35] and *data-flow graphs* [36] constructed by the traditional compilers, where each graph represents a single aspect of the processed code. The HFG definition is platform specific; nevertheless, the construction of the graph is a general algorithm that can be adapted for many platforms (intermediate codes).

A *Hybrid flow graph* is an annotated directed graph, where nodes together with their labels represent operations and edges define data transfers. The orientation of the edge indicates the direction of data flow and the edge annotation defines the data type transported along the edge. The complete list of all supported operations and data types is part of the HFG definition.

The HFG defines two sets of operations necessary to represent both the CIL semantics and control flow structure. The first set of operations is borrowed from the underlying environment, the CIL standard [30]. These operations are derived directly from the CIL instructions and we call them the *basic operations*. The second set contains the *special operations* that model the control flow structure, their semantics are defined in detail in Section 4.3.

The data types supported by the HFG edges are taken from CIL standard [30] and we add a single new data type called *Token*, which represents elements without value used to initialize the HFG nodes.

Figure 4.1 shows an example Hybrid flow graph implementing the $factorial$ function defined in Listing 4.1. The graphical notation introduced in this sample is carried over to the other examples. The graph nodes represent:

```
int factorial(int a) {
  int b = 1;
  do {
    b = a * b;
    a = a - 1;
  } while (a > 1);
  return b;
}
```

Listing 4.1: Method containing a single loop

- basic operations – rounded rectangles inscribed with instruction codes
- *start* special operation – squares inscribed with I
- *return* special operation – squares inscribed with R
- *parameter* special operation – squares inscribed with P and parameter name
- *merge* special operations – squares inscribed with M
- *split* special operations – squares inscribed with S
- *loop primer* special operations – squares inscribed with L
- *broadcast* special operations – filled squares without inscription
- *input* special operations – circles inscribed with number

The special operations represent control flow and data distribution, they are defined later, in Section 4.3. The input operations represent specific input for other operations, especially for those with more than one, we omit them for operations with a single input to make the examples more compact. The data types of the edges are defined by their graphics as follows:

- integer – solid lines
- boolean – dashed lines
- Token – dotted lines

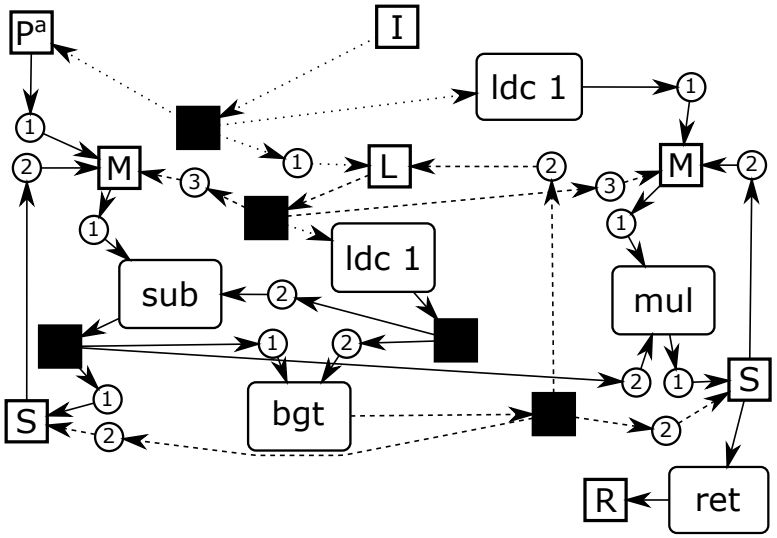

Figure 4.1: The HFG representation of the *factorial* function

4.1 Related Work – Hybrid Flow Graph

We designed the Hybrid Flow Graph (HFG for shot) as an intermediate language capable of representing procedural applications across various streaming environments in a form similar to the structure of their native applications. The HFG is a graph-based declarative language that uses operators defined according to the CIL instructions and it is defined along with a compiler able transform C# code into the HFG. The final structure of the Hybrid flow graph was presented in our work [1], where we defined the graph structure, semantics and construction. The original theoretical basis was formulated in our previous work [5], but the graph was further refined to accommodate all the features necessary for code representation and efficient execution.

The Hybrid Flow Graph described in this work is similar but not identical to other modeling languages, like Petri nets [39] or Kahn process networks [40]. The main difference is that the HFG was designed for automatic generation from the source code, where the other languages are generally used to model the application prior to implementation [41] or to verify a finished system [42].

The Hybrid Flow Graph is not only a compiler data representation, it is a processing model as well, similar to KPN graphs [43]. It can be used as a source code for specialized processing environments, where frameworks for pipeline parallelism are the best target, since these frameworks use similar models for applications [44]. One such a system is the Bobox framework [2], where the flow graph can be used to generate the execution plan similarly to the way Bobox is used to execute SPARQL queries [45].

The Hybrid Flow Graph is closely related to the *program dependence graph* (PDG), which can be also used to represent procedural code [46]. The program dependence graph was designed for traditional procedural languages, like Fortran, where the graph construction relies on the source code statements. Modern languages, like C#, provide far more complex source code syntax and their direct transformation into a graph is impractical, since it means replicating the C# compiler to parse the code. To offset the complexity, we transform the CIL code, which is more simple. The main cost of this approach is increased granularity of HFG compare to the PDG, which we reduce lated using optimizations.

The flow graph has similar traits to frameworks that allow applications to be generated from graphs, like UML diagrams [47] [48], but we concentrate on graph extraction from procedural code, which is more convenient for the design of data processing application.

The flow graph semantics is defined using the graph rewriting system [49], which can be used to design and analyze applications, but it is not convenient for execution. The graph rewriting systems continuously modify the graph structure, which severely limits optimization. There are other frameworks that generate graphs rewriting systems from procedural code like Java [50], but the produced systems are inconvenient for efficient execution and we use them solely to define the graph semantics.

Since the global determinism of our flow graphs is ensured by the generator, the Hybrid Flow Graph has more relaxed local constraints than a KPN, but still more stringent than a Petri net. There are frameworks which are able to construct KPN graphs from simple programs [51], but the flow graph contains

more information about the source code and it is designed for data processing applications.

The Hybrid Flow Graph is closely related to graphs used in compilers, mainly the dependence and control flow graphs [36], where the HFG merges the information from both. The construction of a HFG and its subsequent optimization is related to well-known compiler techniques, like *points-to* analysis [52], *dependence* testing [35] and *control-flow* analysis [53]. In compilers, graphs resulting from these techniques are typically used as additional annotation over intermediate code. On the contrary, our Hybrid Flow Graph contains complete information and, thus, it is an intermediate code representation in itself.

The close relationship to the data-flow (dependence) [36] and control-flow graphs [54] is natural, because together they represent the two most important aspects of the procedural code. However; it is not possible to use either of the graphs alone to fully represent an application, because each lacks some information about the code. The control-flow graph does not define data connections and the data-flow graph is not deterministic. One approach would be to combine both graphs into one, but that generally leads to a graph with an exponential number of nodes relative to the size of the original code.

4.2 Basic Operation Semantics

We use *graph rewriting systems* [48] to define the semantics of operations, which is the approach we used in formal definition of the HFG semantics [6]. We use the graph rewriting systems only as a theoretical tool, because the real implementations work as streaming systems rather than graph rewriting systems.

To properly define the semantics of the Hybrid flow graph we add a third set of operations, the *data operations*. Nodes annotated with these operations represent values processed by the HFG during execution and they do not perform any actual work. We show the data operations as elliptic nodes in the examples.

The data operations are introduced, because the edges represent streams in the streaming systems, but the pure graph representation provides no mechanism to implement this behavior. Instead we chain data as nodes where otherwise would be a stream. These operations are used only to define the semantics and they are not used in the actual implementation, because there, the edges represent streams that store the processed value.

First, we define the semantics of the basic operations based on the CIL instructions. The semantics are defined according to the CIL standard [30], where the consumed and produced values are taken from the input and output streams instead of the common stack or variable memory. It is the responsibility of the compiler to construct the graph so that the edges represent the connections that the stack provides in the original CIL.

Figure 4.2 defines the semantics of a few select CIL instructions, where the elliptic nodes represent the data operations and the $*$ represents an empty token. We do not show the definition of all the instructions, because it is straight forward, the instruction consumes data and produces the results from and to the edges.

The only special case are the branch instructions (like *bgt* in Figure 4.2) which normally move the *program counter*. In our semantics they just produce a boolean

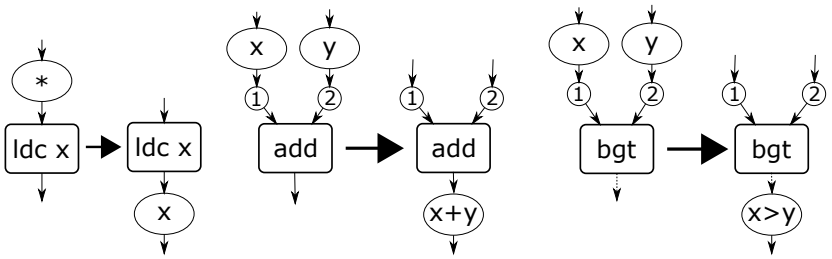

Figure 4.2: Semantics of *load constant*, *add* and *compare (<)*

value that says whether the jump should happen or not and the actual control flow management is handled by the special operations described in the next section.

4.3 Representation of Control Flow

Control flow operations are transformed into *special operations* which interact with the data-flow carried through the *basic operations*. The special control-flow operations are guided by streams of boolean values associated with conditional branches (marked using the dashed edges). The following figures define the semantics and visual notation of all the special operations, where the ellipses represent the data values carried through the edges.

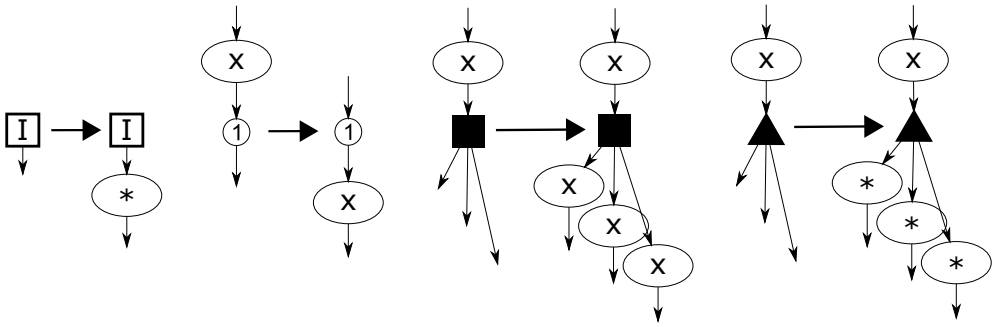

Figure 4.3: The semantics of the *start*, *input*, *broadcast* and *cast* operations

Start

The *start operation* (represented by a square inscribed with I) is used to initialize the graph execution, it primes all the nodes that do not consume any data produced in the graph (like load constant) and it is the only node without input in the HFG. Each graph contains only a single start, which is executed immediately upon start and produces a single *token* that initiates other operations.

Input

The *input operation* (represented bu a circle inscribed with a number) is an empty operation that identifies specific inputs for operations with multiple. The operation passes its input to its output without any change. We omit the input operation for operations with a single input in the examples.

Broadcast

The *broadcast operation* (represented by a black square) has a single input and a variable number of outputs and it creates a copy of each input value for each output. This operation is used whenever an operation must pass its result to multiple consumers and their number defines the number of outputs of the broadcast.

Cast

The *cast operation* (represented by a black triangle) is a special version of the broadcast operation that does not copy the input value, but outputs a single *token* for each consumed value. The cast basically converts a value to a token for all outputs.

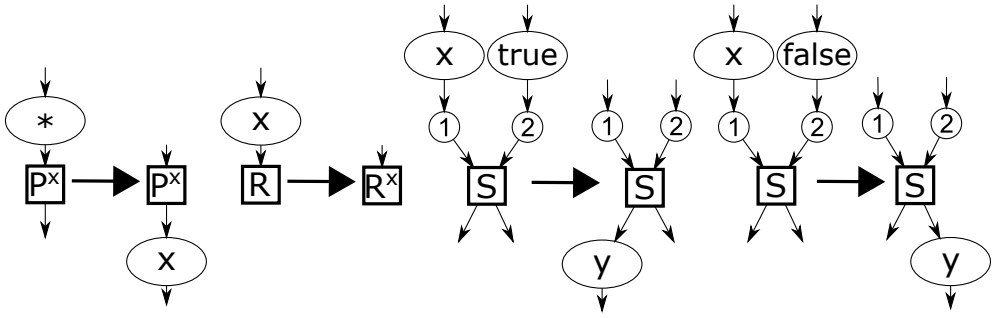

Figure 4.4: The semantics of the *split*, *parameter* and *return* operations

Parameter

The *parameter operation* (represented by a square inscribed with P^x) represents a parameter of the original method and provide the graph with its input value. The parameter has internally stored the input value and it returns it gets input token straight from the start operation. Section 4.4 explains in grater detail the execution semantics of the hybrid flow graph.

Return

The *return operation* (represented by a square inscribed with R^x) is basically the reverse of the parameter operation, it internally stores the input value, which represents the return value of the hybrid flow graph execution. See Section 4.4 for more details.

Split

The *split operation* (represented by a square with S) separates values from a single stream into two independent streams based on its boolean condition input. The operation inspects the condition and passes the input to the appropriate output. The split operation is used with the merge to separate branches and loop iterations (see Section 6.7 for details).

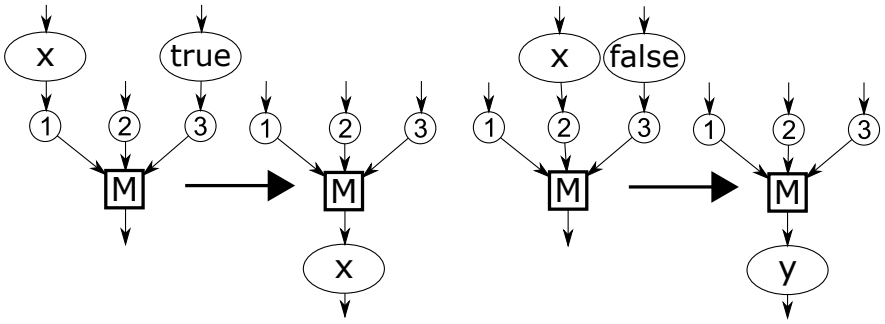

Figure 4.5: The semantics of the *merge* operation

Merge

The *merge operation* (represented by a square with M) combines two input streams into a single output stream according to a boolean condition input stream. The operations first inspects the condition value and then passes the appropriate value from one of the input streams to the output stream, one input contains the values for the *true* condition and the other for $false$.

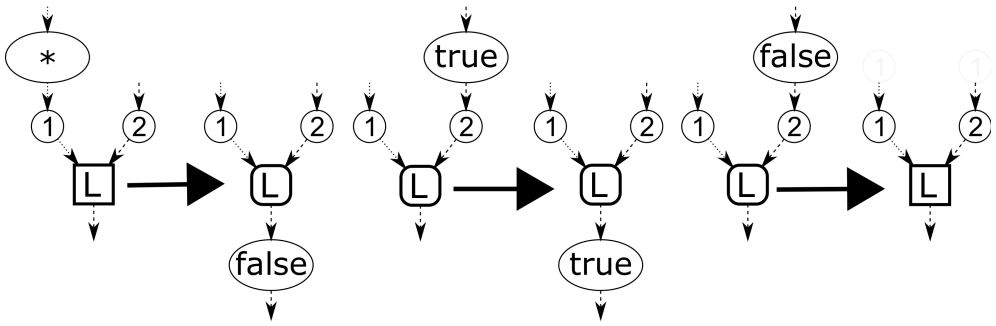

Figure 4.6: The semantics of the *loop primer* operation

Loop Primer

The *loop primer operation* (represented by a square with L) it the only operation with two states. The operations first reads its first input and returns *false*, then it reads its second input while it is *true* and returns *true*. When the second input is *false*, the loop primer returns to the initial state (reads first input). This operation is used with *split* and *merge* to control loop iterations, because the loop body must first consume data produced prior to the loop and than it consumes data produced in the previous iterations.

4.4 Hybrid Flow Graph Execution

The hybrid flow graph is an intermediate language that is not meant to be executed by itself, it is further transformed into a streaming system application, but it is still beneficial to fully define its execution semantics. The hybrid flow graph operational semantics is defined by the graph rewriting system presented

in the previous sections, but an actual execution (even if theoretical) still requires simple preparation.

The HFG is defined and constructed so that there is a single node without input that initialized the entire graph. This means that the parameters must be filled into the graph before it is executed, we call this step the *instantiation* and it requires that all parameters are annotated with their actual values. Next we apply the rules of the graph rewriting system, while there is at least one applicable, we named this phase *evaluation*. And finally we extract the result in an operation we call *reduction*, where we get the value from the return operation. The process is defined in Algorithm 1.

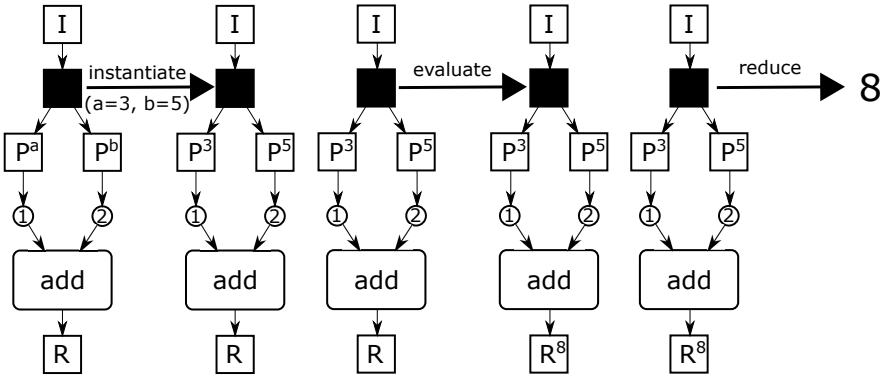

Figure 4.7: Hybrid flow graph execution example

The actual execution can be illustrated on a small example. Figure 4.7 shows a graph that adds the value of two parameters and returns the result. The figure shows all three steps of the HFG execution – instantiation (with parameter values), evaluation and reduction (producing the result).

Algorithm 1 Hybrid flow graph execution

Require: G – hybrid flow graph
GRS – graph rewriting system defining HFG semantics
P – set of all the parameter values
Ensure: X – return value
$G^I := instantiate(G, P)$
$R := r \in GRS : applicable(r, G^I)$
while $R \neq \emptyset$ **do**
$r := first(R)$
$G^I := r(G^I)$
$R := r \in GRS : applicable(r, G^I)$
end while
$X := reduce(G^I)$

4.5 Layered Hybrid Flow Graph

The hybrid flow graph definition is based on the CIL instructions, which means that the graph is generally large and granulated, which can cause inefficiency in

the final streaming application. We designed a transformation called *component extraction* to offset this effect.

The component extraction takes select parts of the graph and combines them into a single operation with equivalent semantics. In practice, this would require adding new operations and their semantics into the hybrid flow graph language definition, which would be impractical.

We define a extended version of HFG to allow the components to be defined directly by their original subgraph. A *layered hybrid flow graph* is a hybrid flow graph which can contain *custom operations* that have semantics defined by another hybrid flow graph.

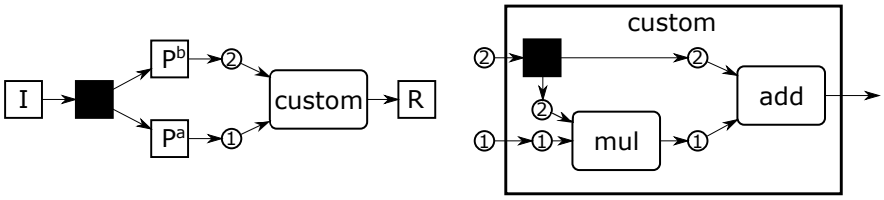

Figure 4.8: Layered hybrid flow graph example

Figure 4.8 shows a small hybrid flow graph with a custom operation defined by another graph that adds and multiplies its two inputs and returns a single value.

5. Compiler for Streaming Environments

The design of applications for streaming environments can be complicated and requires specific knowledge of the environment and sometimes even the underlying hardware. We discussed the specifics in Section 2.2.

The goal of this project is to make the design of applications for streaming environments simpler and more efficient. To accomplish this, we have designed the *ParallaX compiler* that allows developers to design entire applications or their parts in a common procedural language, we selected C# for its wide spread knowledge and well documented intermediate code. Developers simply create an application in C# and our compiler transforms it for a supported streaming environment. We currently support multiple parallel environments, but we mainly focus on a native implementation of the Bobox [2] streaming environment. The other supported environments are specified in Section 8.2.

5.1 Related Work – Intermediate Code and Parallelism

The ParallaX compiler was designed to simplify the development of applications for the Bobox streaming system, its structure is based on our work [1], where we defined the compiler architecture and the compilation process. The theoretical basis of the entire system is based on our previous work [7], where we presented the basic concepts of the Hybrid Flow Graph and its construction.

Compilers of procedural programming languages traditionally use intermediate code separated into two layers – the *control flow* and the *data flow*, the former being a graph of *basic blocks* while the latter assigns either a sequence of instructions or a data-flow graph to each basic block. This textbook approach [57] has become standard whenever the target machine is a CPU which posses direct representation of the control-flow instructions. Data-flow graphs in compilers are usually limited to individual basic blocks – this limitation guarantees acyclic structure of the data-flow graphs (named *dags* therefore) which in turn allows simpler semantics and easier handling.

Attempts to integrate the control-flow and data-flow levels of intermediate code occurred throughout the decades of development of compilers, the most successful being probably the Program Dependence Graph (PDG) [46] used recently in compilers for GPU architectures [58]. Nevertheless, the PDG is only an annotation over the underlying sequential code and modern compiler ecosystems like LLVM still rely on intermediate codes based on instruction sequences.

Different representations are used in hardware synthesis where the result of compilation is not a code but an electronic circuit [59]; however, the source language SystemC, although originated from C, was significantly changed towards the hardware definition domain; consequently, the language as well as the principles of compilation are not easily applicable to general programming.

Synthesis of electronic circuits has been the main motivation for data-flow

representations of procedural code; nevertheless, intermediate codes and conversion algorithms targeted at general computing appeared both in the history [60] and present times [61], sometimes suggesting a new hardware architecture as the target platform [62]. Since conversion of low-level languages like C++ is difficult due to aliasing problems, some conversion algorithms require hints in the form of annotations added to the procedural source code [63, 64].

Although none of the suggested approaches reached wide recognition yet, the current success of high-level data-flow architectures like TensorFlow suggests that data-flow orientation may be a viable alternative to von-Neumann architectures.

In the environment of modern languages like C# or Java, there already were several alternate intermediate codes introduced. Of particular importance to our approach are systems similar to graph-rewriting systems, like the one described for Java [50]. Java was also subject of numerous approaches to vectorization [65, 66] or parallelization [67, 68]. Parallelization often makes use of additional user-input through the use attributes or other ways of annotation [69, 70]. Another technique which shares some similarity with stream-based processing was described in [71].

There are also works focusing on the parallelization of dynamic languages, mainly used for web content production or scripting, like JavaScript, PHP and Python. JavaScript can efficiently take advantage of the parallel capabilities of HTML5 [72], or there is the possibility to employ even more powerful technologies like OpenCL [73]. The parallelization of PHP is one potential goal of the Peachpie compiler [74]. All the previous works focus mostly on loop parallelization or vectorization and cannot be directly adapted for streaming systems, but they show that the dynamic languages can be adapted as an interface for parallel systems.

5.2 Compiler Architecture

We have designed a compiler that takes a C# application compiled to CIL and transforms it to an application for a streaming environment. We call it the *ParallaX compiler*. The compiler mostly follows the standard architecture shown in Figure 5.1, where we omit only the platform specific optimizations as they are performed by the streaming environment and the C++ compiler associated with it.

The architecture in Figure 5.1 is designed for a direct transformation, source code to a machine / binary / byte code, but streaming environments require plan and operations, not just a single code. Therefore; our compiler requires some non-standard parts and a special intermediate language better suited for streaming environments. The structure of our compiler is shown in Figure 5.2. We have designed a special intermediate language with platform independent structure and semantics well suited for streaming environments. We named this language *Hybrid flow graph*, or *HFG* for short, and it allowed us to adapt our compiler for multiple target platforms besides just native Bobox. The language was described in detail in Chapter 4.

Figure 5.2 shows the architecture of the ParallaX compiler with its components grouped into the three main parts – *front-end*, *optimization* and *back-end*. The compilation process is explained in the following chapters, but the basic idea is straight-forward: the CIL instructions are transformed into the graph nodes and

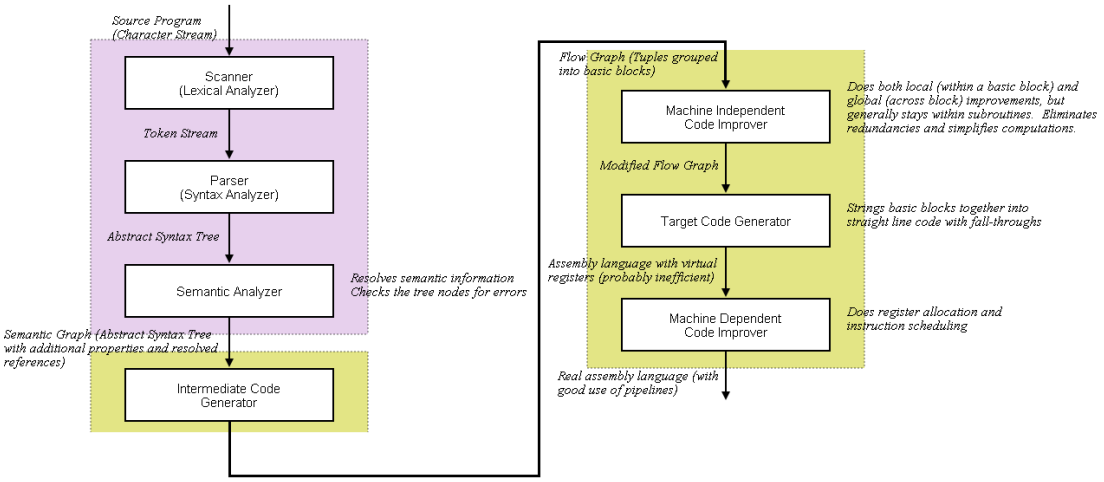

Figure 5.1: Standard compiler architecture overview taken from the book Compilers: principles, techniques, tools [57]

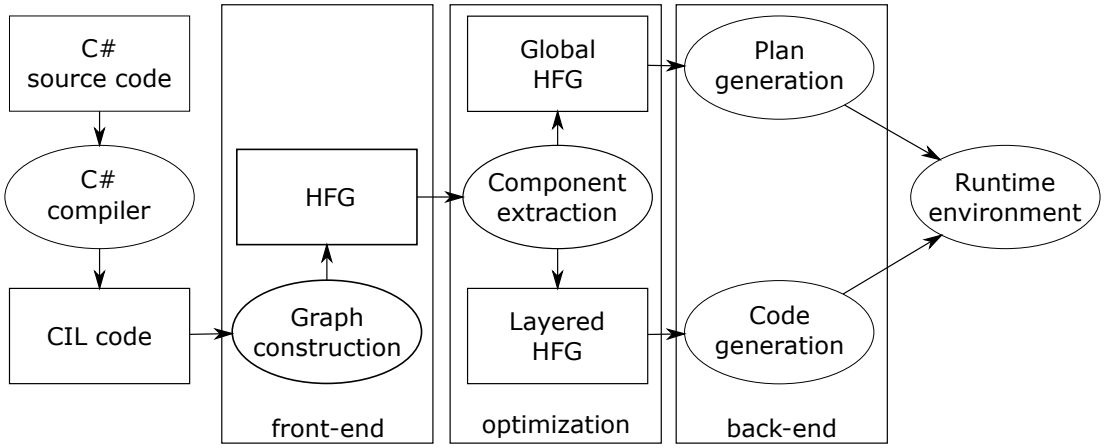

Figure 5.2: Compilation and code generation

connected by edges, representing data transfers, created according to the data and control flow. The instructions are then grouped based on the graph structure to create kernels (basically complex instructions) for the streaming environment. The data transfers between the operations from the execution plan, because they define how the operations are executed and how they interact.

The first part of our compiler is the *front-end*, where the code of a single procedure, usually the *main* method, is transformed into a *hybrid flow graph*. The input of our compiler is a C# code compiled into a CIL code by a C# compiler. Chapter 6 describes all the steps necessary to create a HFG equivalent to the input code.

Next, the graph is transformed into a *layered hybrid flow graph*, defined in Section 4.5. This step is called *optimization*, because it reduce the graph granularity and improves the efficiency of the final application. The process is explained in detail in Chapter 7.

Last part of the compiler is the *back-end*, where we take the layered graph and transform the top layer to the execution plan for the streaming environment and the rest we transform into C++ code implementing the *custom operations*. The

back-end is platform specific, depending on the target environment, we discuss the available versions in Chapter 8. The produced application is then processed by the runtime environment and executed.

5.3 Input Language Restrictions

A compiler must always produce a valid code, which means that it has to be very conservative when it encounters constructs that do not follow a pattern that allows for optimization. This is especially important for a compiler that transforms the code between two very different platforms, like the object-oriented C# and streaming environments.

The structure of the streaming environments is significantly different from the object-oriented environment of the CLR and C# and we impose restrictions on the applications the ParallaX compiler is able to process. Without the restrictions, the produced code would be too inefficient or the application might be impossible to compile. The ParallaX compiler does not currently support the following constructs:

- *objects* – the compiler supports only static methods, which can call other static methods.
- *multi-dimensional arrays* – we support one-dimensional arrays, but it is possible to represent multi-dimensional arrays in one-dimensional arrays.
- *exceptions* – exceptions and guarded blocks are not allowed, errors can be reported by returning invalid values.
- *libraries* – library methods and classes are not supported, because we do not have access to their code (in case of native implementation) and they do not have equivalent in streaming environments. In the future, we will provide special library compatible with the streaming environments.
- *switch* – the compiler supports only if-else, but switch support will be added in the future.
- *break*, *continue* and *goto* – we do not allow these constructs, because they can abruptly interrupt the control flow in a way that is not possible in streaming environments.

This work is a proof-of-concept and the purpose of the ParallaX compiler is not to parallelize arbitrary C# applications, but to provide another interface for the streaming environments to developers without specific domain knowledge of the systems. However; the compiler can be further improved to incorporate some of the restricted concepts.

Switch can be added with slight modifications to the compiler, because it can be represented by a series of if-else branches already supported by the compiler. Custom libraries must be implemented, but there is already a small library integrated with the compiler that can be used as an example.

Break, continue and goto cannot be simple introduced, because the streaming environment lack the means to replicate their behavior. It might be possible to

add break and continue by restructuring the respective loops, but a general goto cannot be added as it can change the control-flow too significantly.

Multi-dimensional arrays can be added as well, but they would introduce complex aliasing and their efficient handling would require advanced optimizations, which are currently not implemented.

Value object types (structs) can be added with proper handling. They do not introduce aliasing, because they do not have reference semantics. Their integration could be achieved in a similar way as single-dimensional arrays, described in Section 6.9.

Objects and exceptions cannot be fully supported, because they do not have any equivalents in the streaming environments. Exceptions can unpredictably change the execution order and modify the application state and there is no way to do that in a streaming system, which is driven by data. Objects along with the garbage collector can be also very unpredictable and they introduce aliasing, which could severely limit available parallelism.

6. Compiler Front-end

Figure 6.1 shows the overview of the HFG construction process. Our input is a CIL assembly compiled from a C# code. The original C# code is compiled using Microsoft *cs* to CIL and packed to an assembly (library or executable) by *MSBuild*. The resulting CIL assembly is analyzed and directly transformed by our compiler to the final Hybrid flow graph.

6.1 Front-end Overview

The compiler transforms a single procedure to a streaming application, which usually means the *main* method of the C# application. Aside from the compiler itself, we provide also a small library of attributes that allow users to specify other procedures for transformation and they can also modify the way it is carried out, used optimizations and generated outputs.

The basic idea of the HFG construction is to transform all the CIL instructions to graph nodes and connect them according to data and control flow. We create a set of *basic nodes*, each representing a CIL instructions. Then we add special nodes based on control flow analysis completing the *HFG*, which represents the entire input code (procedure or application). A complete HFG is later transformed to plan for a streaming environment or another parallel system. Finally, we optimize the complete HFG to better balance overhead and available parallelism.

General parallelization approach is to analyze the source code and locate certain patterns or dependences to find opportunities for parallelism. We use a reverse approach, we transform the application to the HFG directly and then we optimize the resulting graph by merging instruction nodes wherever it is beneficial.

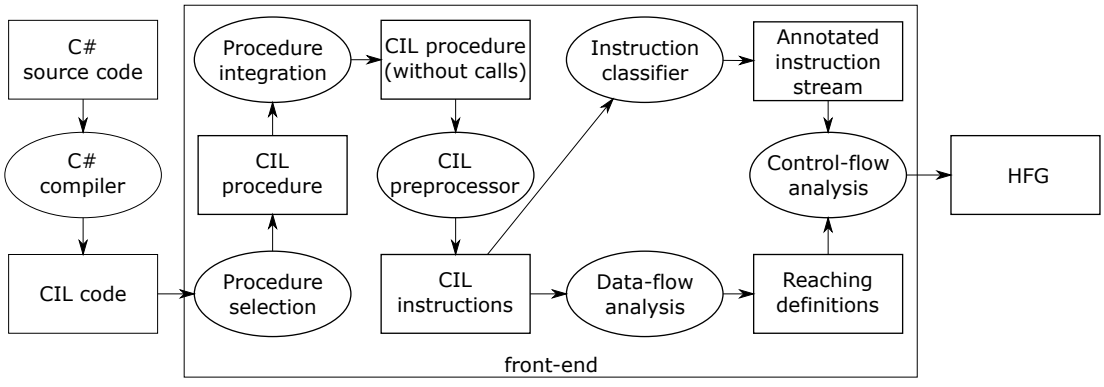

Figure 6.1: Overview of the ParallaX compiler front-end component

Figure 6.1 shows the *front-end* structure with the two main parts being: the data-flow and control-flow analyses. The data flow analysis is similar to the reaching definitions analysis [75], but we calculate it using a graph rewriting system emulating the CIL code behavior. The control flow analysis is based on a context-free grammar and parser generator commonly used in standard compilers.

The compilation starts with three preparation steps that produce a single CIL instruction list that is then transformed into a HFG by the data-flow and control-flow analyses. The preparation steps are: the *procedure selection*, the *procedure integration* and the *CIL preprocessor*. The algorithms are described in Section 6.3, Section 6.4 and Section 6.5 respectively.

We use the *data-flow analysis* to locate all producers and consumers for each instruction, we describe the process in Section 6.6. The data-flow analysis produces a set for each instruction that contains all the instructions that consume a value produced by the instruction, basically a set of all consumer. The output is similar to the result of the *reaching definitions analysis*, which locates all the values able to reach each statement [75].

The *instruction classifier* categorizes the instructions according to their control flow behavior: conditional and fixed jumps, jump targets and standard instructions. The process is described in Section 6.6.5.

The control-flow analysis takes the output of both the data-flow analysis and the instruction classifier and constructs the final hybrid flow graph. The control-flow analysis locates the control flow constructs (loops and branches) and for each construct, the analysis produces its body and connects it to the entire graph using special operations. The non-jump instructions are transformed to *basic operations* and connected to the graph.

CIL code represents control flow as a set of conditional jumps. However, we analyze the CIL code produced by the Microsoft C# compiler, which transforms control flow into a well defined structure of jumps. We reconstruct the control flow based on that structure. This approach does not support general CIL code, but it is sufficient for a CIL generated from a C# source code, which is our target platform.

6.1.1 Transformation Example

We will illustrate the entire transformation process on a single example to provide a more coherent insight into each step. Listing 4.1, from the previous chapter, contains a C# source code of the $factorial$ method and Listing 6.1 contains the CIL code produced by compiling the C# source. Figure 4.1 shows the complete HFG created from the CIL code using the algorithms described in the next sections and throughout this chapter, we will construct the graph step-by-step.

6.2 Related Work – Compiler Front-end

The compiler front-end is responsible for the transformation of a source code into a more manageable intermediate language, which is either simpler, more efficient or similar to the target environment. The textbook solution contains a series of analyses that explore the aspects of the source code and provide information about its structure for the intermediate code construction or further optimizations [57].

The front-end of our compiler follows the standard solution – we construct the Hybrid Flow Graph (intermediate code) from a C# code, compiled into CIL code by a standard C# compiler (source code). To properly construct the HFG,

Listing 6.1: CIL code created from C# code in Listing 4.1

```
.method private static int32 factorial (int32 a)
{
  .maxstack 2
  .locals init ([0] int32 b)

  IL_0000: ldc.i4.1
  IL_0001: stloc.0
  // loop start (head: IL_0002)
    IL_0002: ldarg.0
    IL_0003: ldloc.0
    IL_0004: mul
    IL_0005: stloc.0
    IL_0006: ldarg.0
    IL_0007: ldc.i4.1
    IL_0008: sub
    IL_0009: starg.s a
    IL_000b: ldarg.0
    IL_000c: ldc.i4.1
    IL_000d: bgt.s IL_0002
  // end loop
  IL_000f: ldloc.0
  IL_0010: ret
} // end of method Program::Factorial
```

we require three main analyses: the *data-flow analysis*, the *data type analysis* and the *control-flow analysis*.

Before we apply the aforementioned analyses, we must address the procedure calls present in the source code. The procedure calls limit our ability to safely construct the Hybrid Flow Graph structure and this is especially problematic with procedures where we do not have access to their source code. To eliminate this problem, we limit the use of C# standard library, instead; we provide our own library that contains procedures known to the compiler. We use *procedure integration* (also known as inlining) [76], which reduces the source code into a single procedure without any additional calls [1]. The integration is commonly performed by the CIL runtime to produce more efficient code [77], but we have to perform the integration ourselves since we work with the CIL code directly.

The *data-flow analysis* is responsible for mapping producers and consumers for all potential variables. The analysis creates a set of producers and consumer for each CIL instruction, so we can properly connect the graph nodes based on them. The concept is very similar to the *reaching definitions analysis* [75], where we use a more dynamic solution to gather additional information about the code behavior. We statically emulate the source code to analyze the data-flow, data types of values and most commonly used control-flow paths. The concept is similar to the optimizations based on runtime behavior of the application or a profiler [78].

The compiler accepts CIL code as the input for the front-end, which is a strongly typed language. Therefore; we are not subject to the undecidable data types commonly encountered in dynamic languages, like PHP [79]. The CIL structure still allows for implicit and explicit conversion [31], which can seriously limit the performance of the resulting applications, because it might lead to constant vector conversions. We analyze the data types during the data-flow analysis and assign a strict type constraint to each edge of the resulting HFG [1].

Another common problem of the object-oriented languages is *aliasing*, which is usually addressed by the *points-to analysis* [80]. We do not have to address complex aliasing, because we forbid the use of object types in the source code, with the exception of the arrays. In the Hybrid Flow Graph produced by the front-end, the nodes always exchange complete arrays and modify them locally before passing them on. This inefficient behavior is later addressed by the optimizations, which optimize only arrays that are not aliased in the relevant context, where we use a process similar to the *live variable analysis* [4].

The *control-flow analysis* must determine the order in which the instructions are processed in the source code and we use this information to properly merge the HFG edges, representing data transfers. Our implementation follows the pattern similar to the process of basic blocks extraction [54]. We use a pushdown automaton, commonly used in syntax analysis [57], to analyze the control-flow and we construct the Hybrid Flow Graph in the process (according to the results of the previous analyses).

6.3 Method Selection

The first preparation step must select methods, which will be transformed into streaming applications, and provide their CIL byte code. This step is important,

because the input for the ParallaX compiler is a CIL assembly, but the following steps work with CIL instructions, which must be loaded from the assembly.

We use the *Mono Cecil* library [81] to load the input assembly into an object representation, which allows us to manipulate the code more efficiently. The library is part of the Mono implementation of the CLR standard, but it works with the other implementation as well, including the Microsoft .NET.

Once the code is loaded, we select the methods that will be transformed, each into a separate streaming application. This approach allows users to implement multiple streaming applications in a single assembly and then compile them together. Each application is started by a single method, just as it is in the CIL and most other procedural languages, and the initial method calls the others when they are needed.

The method selection process iterates all the methods of all the types in the assembly and selects those that have the $ParallaX.Interface.Transform$ attribute attached. The attribute is provided with the compiler in tiny library. If there are no methods with the attribute, we simply select the main method and it becomes the only input for the rest of the compilation.

Once all the methods are located, the code of each is loaded and the methods are passed one-by-one to the next compilation step.

6.4 Method Integration

Method integration, also known as *inlining*, takes the method selected in the previous step and eliminates all the calls to other methods, by integrating their code directly into the main method. This way we produce a single sequence of CIL instructions that can be processed by the following steps without the need to incorporate other code.

Another approach would be to analyze the called methods separately and directly create a *layered HFG*, but that would mean that the code structure would influence the granularity of the resulting graph. We might be able to balance the graph structure by moving code between layers (representing procedures) , but that is basically method integration implemented on the intermediate language instead of directly in CIL.

Method calls may also be handled using a call operation embedded in the graph; however, such operation is difficult to implement in streaming systems. Therefore, procedure integration is the favored way to handle function calls in our environment.

The main drawback of the integration is its inability to handle recursive calls, with the notable exception of the tail-recursion [82], which can be replaced by a loop. This is a disadvantage, but the data intensive applications rarely use recursion, because it can severely limit their performance.

Another problem that we face is the fact that we cannot integrate library methods, because they are usually implemented in native libraries and we do not have access to their code. This includes methods for file or graphics management, which cannot be used in the target application anyway, because the streaming environments contain no equivalent to their functionality. For this reason, we do not allow library methods at all. We provide our library that supplies the necessary arithmetic functions and is integrated with the compiler.

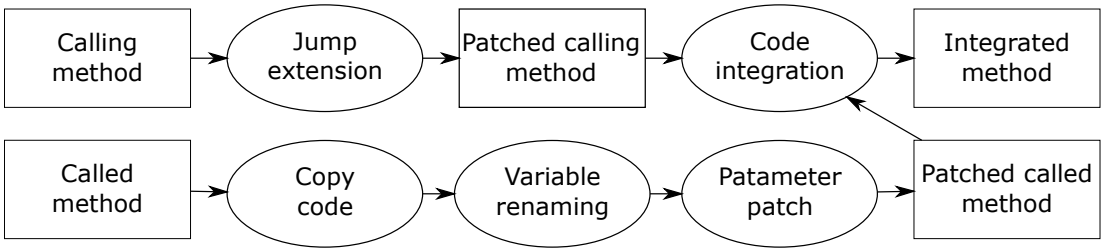

Figure 6.2: Method integration process

We have implemented an explicit method integration algorithm, because we have to eliminate method calls in the analyzed code. The CLR *just-in-time compiler*, or *JIT* for short, integrates small methods, when the code is loaded for execution, which is too late for us. We needed to be able to integrate methods in the CIL so we can produce a code without any calls and transform it to HFG. The following sections describe the process in detail.

6.4.1 Integration overview

The integration process is straightforward but requires many technical details to be addressed for the code to be valid according to the standard [30]. The integration always integrates a single *called method* into a *calling method*. If a method calls multiple methods then they are integrated one after another. Figure 6.2 show the entire integration process for a single method. We use the Cecil library, which allows us to change the code efficiently without complex binary calculations.

The first step is to copy the code of the *called method*, because we do not want to modify the original code, because that would make it invalid. On the other hand, we can modify the *calling method*, because it will be fully valid at the end of the integration. The further steps are described in the following sections in detail.

6.4.2 Variable Renaming

We must rename all the local variables of the called method so that they do not conflict with variables of the calling method. This requires that we also patch the instructions accessing the variables by swapping the original name for the new one. Algorithm 2 shows the renaming, where the $AvailableName$ method generates a new variable name available in the calling method. For example: $VariableN : N \in \mathbb{N} \wedge VariableN \notin Var(calling)$.

6.4.3 Parameter Patch

Parameters of the *called method* must be renamed as well, but they also have to be replaced by local variables and the instructions accessing them changed to reflect this. The process is the same as for the local variables, with the exception, that we also change the instruction operation code. Store argument ($starg$) is changed to store local ($stloc$) instruction and load argument ($ldarg$) is changed to load local ($ldloc$) instruction. The actual implementation is a bit more complicated,

Algorithm 2 Variable renaming

Require: I – instructions of the called method
V – local variables of the called method
C – local variables of the calling method
Ensure: M – patched called method
{Rename variables}
1: $REG := \emptyset$ // associative register mapping renamed and original variables
2: **for all** $v \in V$ **do**
3: $rv := AvailableName(v, C)$
4: $C := C \cup \{rv\}$
5: $REG := REG \cup \{(v, rv)\}$
6: **end for**
{Patch instructions}
7: **for all** $i \in I$ **do**
8: **if** $i.Operand \in V$ **then**
9: $i.Operand := REG[i.Operand]$
10: **end if**
11: **end for**
12: $M := (C, I)$

because it has to properly transform all the specific versions of the instruction to their appropriate counterparts, for example the transformation must preserve the difference between integer and object versions of the instructions or the code would be invalid.

6.4.4 Jump Extension

Before we integrate the code, we have to first ensure that all the jumps in the *calling method* will remain valid. This means that we have to locate all the jumps that traverse the integrated call instructions and extend them by the number of instructions in the *called method*. Algorithm 3 shows the entire process, we assume that the instructions can be compared according to their position in the code. Jump instructions contain target offset (relative distance) as their operand.

6.4.5 Code Integration

The code can be copied to the *calling method*, once both the methods are prepared. The call instruction is replaced by all the previously copied and patched instructions of the *called method*.

6.4.6 Stack Limit

There is one last technical detail that has to be addressed for the code to be valid. The .NET runtime require all methods to indicate the maximal size of the stack they require, see Section 3.3 for details. This information has to be updated for the *calling method*, because the integrated code can change the require size. We calculate the new size by adding the required size of the calling and called methods:

Algorithm 3 Jump extension

Require: I – instructions of the calling method
$C \in I$ – the call instruction
$JUMP$ – list of all jump instruction codes
N – number of instructions of the called method
Ensure: M – patched instructions

```
1: for all i ∈ I do
2:   if i.OpCode ∈ JUMP then
3:     if i < C ∧ i + i.Operand > C then
4:       i.Operand := i.Operand + N
5:     else if i > C ∧ i + i.Operand < C then
6:       i.Operand := i.Operand − N
7:     end if
8:   end if
9: end for
10: M := I
```

$$calling.MaxSize := calling.MaxSize + called.MaxSize \tag{6.1}$$

6.5 Code Preprocessing

Before we transform the CIL code to the hybrid flow graph, we have to prepare it for the compiler. The preprocessor accepts a single sequence of CIL instructions without calls, produced by the *method integration*, and produces another sequence without calls.

The inherent structure of a procedural code has one important feature that has to be addressed, before it can be transformed to a Hybrid flow graph. The most important fact is that procedural application has a memory that is set once but can be read multiple times. This is not possible in streaming environments, because that would mean that such "variable" would have to be connected to all kernels that access its value, which would limit the available parallelism and cause unnecessary synchronization.

There are two situations, where the lack of persistent storage can cause problems. The code in Listing 6.2 shows a situation, where the variable x is repeatedly read in a loop without being assigned there and we have to repeatedly reproduce the value in the resulting graph. The second situation is in Listing 6.3, where the variable y is updated in every other iteration of a loop and we have to preserve the correct value through all the iterations.

Listing 6.2: Value recyclation

```
int x = 2;
int y = 0;
for(int i = 0; i < 5; i++)
{
  y = y + x;
}
```

Listing 6.3: New value discarded

```
int y = 0;
for(int i = 0; i < 5; i++)
{
  if(i % 2 == 0)
  {
    y = y + i;
  }
  //else
  //{
  //  y = y;
  //}
}
int x = y;
```

A variable is considered to be a *live variable* for a place in the source code, if its actual value is read by a later instruction. We locate all the live variables for the beginning of all loops and branches and we insert the statement $x = x$; into respective constructs for every live variable. This means that all live variables are updated in every iteration. We perform a simplified *live variable analysis*, where we locate only variables read before assignment in loops and branches.

This change introduces instructions that do not change the behavior of the application and help us simplify the HFG construction. Any unnecessary kernels that just iterate the same value of a variable are eliminated by optimizations described in detail in Section 7.

Algorithm 4 inserts the statement $x = x$; at the beginning of every loop and branch for every variable live there.

6.6 Data Flow Analysis

The *data-flow analysis* accepts a single sequence of CIL instructions without calls and produces a set of all consumers for each instruction. Each set contains all the instructions that consume values produced by the respective instruction and it is later used to construct the edges of the hybrid flow graph.

We implemented the analysis using a graph rewriting system statically emulating the CIL behavior, because it is more flexible than the static analyses. The *static CIL emulator* is more complex than a general static code analysis, but it inspects only a fixed, limited number of control-flow paths and therefore; is not a true emulator of CIL.

In the process, we use the emulator to gather additional information about the code behavior, like the data types, the stack depth and most used control-flow paths, which we use to design and apply more efficient optimizations. Our *data flow analysis* is basically the reverse of the *reaching definitions analysis* [75], which locates potential producers, where the goal of our analysis is to identify all the consumers for each instruction.

The static CIL emulator accepts the code of a single CIL procedure as its input and assigns unique values to all instructions inputs and outputs, which we than use to match the consumers and producers. It uses CIL memory and

Algorithm 4 CIL code variable access patch

Require: I – sequence of instructions
L – sequence of all loops $\forall l \in L : l \subseteq I$
Ensure: C – patched set of instructions

```
1: C := I
2: for all l ∈ L do
3:    V := ∅
4:    P := ∅
5:    for all i ∈ l do
6:       if i == ldl then
7:          V := V ∪ {i.Operand}
8:       end if
9:       if i == ldarg then
10:         P := P ∪ {i.Operand}
11:      end if
12:   end for
13:   for all v ∈ V do
14:      C = C[0 : l.start]||[ldl v]||[stl v]|C[l.start : C.length]
15:   end for
16:   for all p ∈ P do
17:      C = C[0 : l.start]||[ldarg p]||[starg p]|C[l.start : C.length]
18:   end for
19: end for
```

execution model (variables, stack and program counter) transformed to a graph. This emulator is not a hybrid flow graph, even though it uses graph rewriting to define its semantics. The emulator uses CIL instruction behavior specified in the *ECMA* standard [30] to properly model the CIL behavior.

Instead of emulating the actual calculation, we modify the instruction behavior, the instructions then exchange unique identifiers instead of actual values and we store all the identifiers consumed and produced for each instruction. This way, we assign a unique identifier to each input and output and we can produce the consumer sets that are the final output of the data-flow analysis. We emulate only the minimal number of control-flow paths to limit the complexity of the analysis. The initialization and emulation is explained in the next sections.

Listing 6.4 shows identifiers assigned by our emulator to the CIL instructions of the $factorial$ method (original C# code is in Listing 4.1), where the identifiers are shown as $(inputs) \rightarrow (outputs)$.

We take the identified values and produce the sets of consumers according ot the Algorithm 5, which iterates over all the instructions and assigns each the set of consumers based on the values the instruction produces. The final result of the entire data-flow analysis for the $factorial$ method is in Listing 6.5.

6.6.1 CIL Sequential Graph

A *CIL sequential graph* is an annotated directed graph, with semantics defined by a graph rewriting system. It has a strict control flow that directly copies that of the source code using a program counter and it has nodes representing

```
IL_0000: ldc.i4.1                          ()->(id1)
IL_0001: stloc.0                         (id1)->(id2)
// loop start (head: IL_0002)
IL_0002: ldarg.0                     (p1,id10)->(id3)
IL_0003: ldloc.0                     (id2,id6)->(id4)
IL_0004: mul                         (id3,id4)->(id5)
IL_0005: stloc.0                         (id5)->(id6)
IL_0006: ldarg.0                     (p1,id10)->(id7)
IL_0007: ldc.i4.1                          ()->(id8)
IL_0008: sub                         (id7,id8)->(id9)
IL_0009: starg.s a                       (id9)->(id10)
IL_000b: ldarg.0                       (id10)->(id11)
IL_000c: ldc.i4.1                          ()->(id12)
IL_000d: bgt.s IL_0002           (id11,id12)->()
// end loop
IL_000f: ldloc.0                         (id6)->(id13)
IL_0010: ret                           (id13)->()
```

Listing 6.4: Produced and consumed values

```
IL_0000: ldc.i4.1                  {IL_0001}
IL_0001: stloc.0                   {IL_0003}
// loop start (head: IL_0002)
IL_0002: ldarg.0                   {IL_0004}
IL_0003: ldloc.0                   {IL_0004}
IL_0004: mul                       {IL_0005}
IL_0005: stloc.0                   {IL_0003, IL_000f}
IL_0006: ldarg.0                   {IL_0008}
IL_0007: ldc.i4.1                  {IL_0008}
IL_0008: sub                       {IL_0009}
IL_0009: starg.s a                 {IL_0002, IL_0006, IL_000b}
IL_000b: ldarg.0                   {IL_000d}
IL_000c: ldc.i4.1                  {IL_000d}
IL_000d: bgt.s IL_0002             {}
// end loop
IL_000f: ldloc.0                   {IL_0010}
IL_0010: ret                       {}
```

Listing 6.5: Data-flow analysis assigned consumer sets

Algorithm 5 Consumer set construction

Require: I – sequence of instructions
IN – array containing sets of all the values consumed by each instruction
OUT – array containing sets of all the values produced by each instruction
Ensure: C – array containing sets of consumers for every instruction

```
1: for all i ∈ I do
2:    C[i] := ∅
3:    for all j ∈ I do
4:       if OUT[i] ∩ IN[j] then
5:          C[i] := C[i] ∪ {j}
6:       end if
7:    end for
8: end for
```

all variables, arguments and one for the stack, see Section 3.3 for details of the CIL language. Figure 6.3 shows a sample CIL sequential graph representing the *factorial* method, where the nodes annotated in upper case represent the special operations implementing the execution stack ($STACK$), the variable a (VAR), the argument p (ARG) and the program counter (P). The variables have an initial value according to the CIL standard and the stack has an empty value so we do not have to define separate semantics for an empty stack.

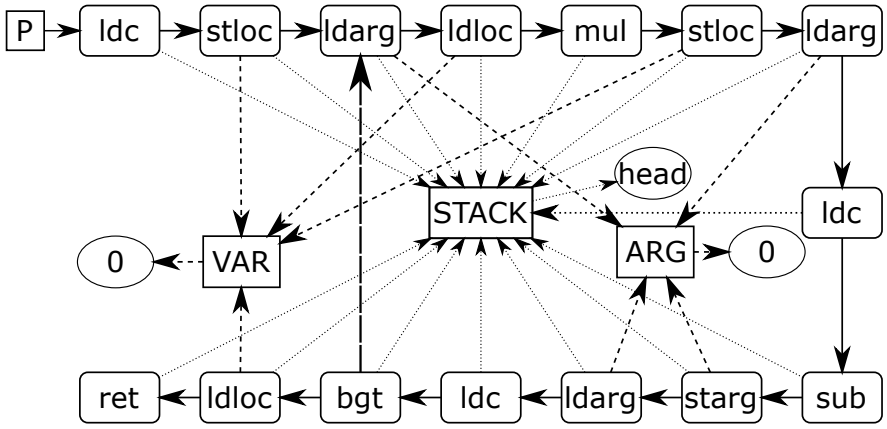

Figure 6.3: CIL sequential graph representing the *factorial* method

The CIL sequential graph is a parallel to the hybrid flow graph, but it is not the same because it has a completely different semantics and its structure is useless for streaming environments as it provides only a very limited parallelism.

The semantics of the CIL sequential graph are defined using a graph rewriting system based on the CIL semantics defined in its standard [30]. Figure 6.4 contains the rules defining the behavior of the basic CIL instructions, we present only the rules necessary for our examples, the rest are defined the same way according to the behavior of the respective instructions. It is important to note that the edges graphical design defines its annotation, which is most important for the conditional jumps, where the thicker dashed edge represents the jump and the regular edge is used when the condition is false.

The semantics directly copy the CIL standard, mainly the stack management.

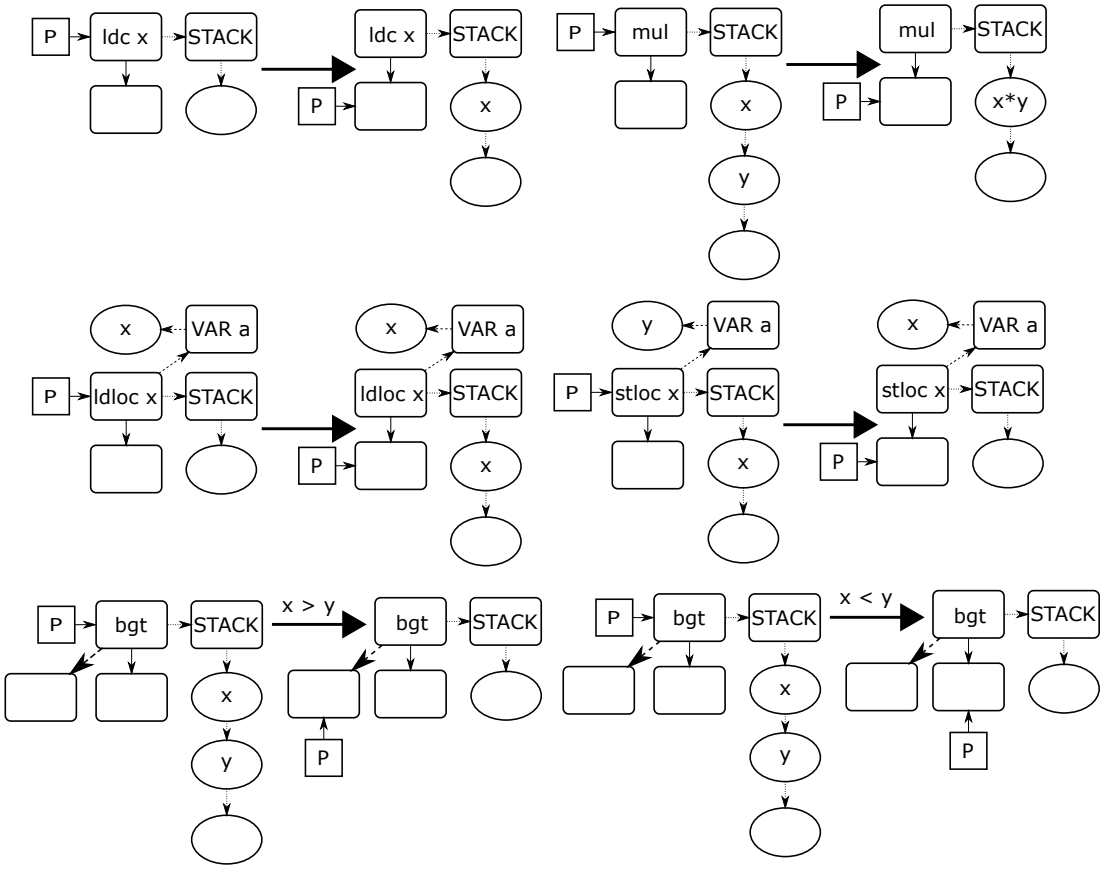

Figure 6.4: Semantics of selected CIL sequential HFG operations

The control-flow is driven by the program counter (node P), which follows the edges between instructions. The data management is handled by the stack and variable operations, where each instruction consumes and produces the proper number of stack levels or changes the variable values, based on its definition. The construction of the CIL sequential graph is described later in Section 6.6.4.

The CIL sequential graph execution follows the same principle as a graph rewriting system, rules are applied as long as there is at least one applicable. The execution copies the work of the CLR virtual machine, the control flow follows the same path, variables and stack contain the same value after every evaluated instruction and there is always at most one applicable rule, because there is only one program counter (P).

6.6.2 Symbolic Semantics of the CIL Sequential Graph

To analyze the data flow, we must introduce a modified version of the CIL sequential graph. The symbolic semantics do not emulate the CIL calculation, instead we collects the consumed values.

We define the *symbolic semantics* of the CIL sequential graph by another graph rewriting system. The modified rules work with instruction identifiers instead of actual values. This way, we can analyze which instructions consume the values produced other instructions. The example symbolic rules are shown in Figure 6.5, we modify other rules the same way.

We assign each instruction an identifier equal to its position in the sequence

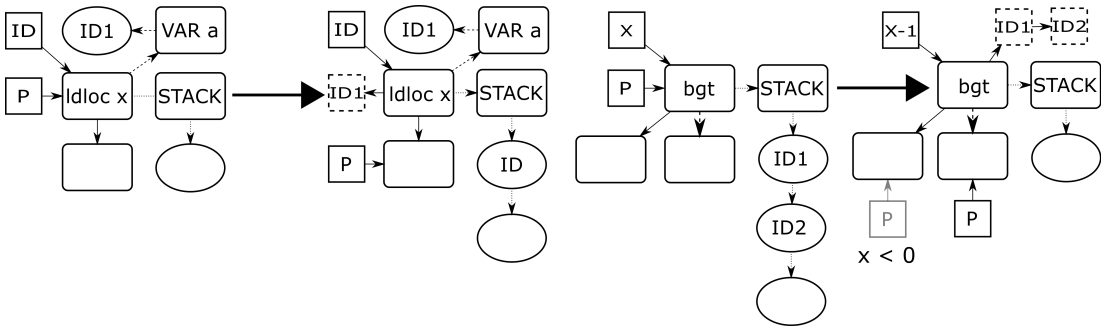

Figure 6.5: Symbolic semantics of selected Sequential HFG operations

($\forall i \in I : id(i) = i$). Last modification is that the rules store identifiers of consumed values, this is indicated by the bold dash-edged nodes in Figure 6.5.

The identifiers of branches decide how many times the branch is evaluated as *true* and how many times it is $false$, we use this to drive the control flow (explained in the next section). Branches do not require an identifier, because hey do not produce any values, they just change the execution order.

6.6.3 Execution of the Symbolic CIL Sequential Graph

We use the symbolic semantics defined in the previous section to analyze the values produced and consumed by instructions, we call this process a *symbolic CIL emulation*. The symbolic semantics work with instruction identifiers instead of actual values and conditional branches behavior is predefined. The result of the symbolic emulation is a list of consumed values we than use to construct sets of consumers for each instruction.

The symbolic emulation requires one extra step before the standard rule application. The conditional branches have to be setup with information about the way they should be executed. This means simply assigning a number to each branch defining how many times it is evaluated as *true* and $false$. To cover all the possible execution paths, we have to use all combinations of branches – each jump must be at least twice *true* and $false$ for every other nested jump, which leads to the maximum of $4^{\#jumps}$ execution paths. This is necessary to model the loops, where the first iteration consumes values produced before the loop and all the following iterations consume the values produced by the loop itself. The number of passes may be high, but the emulator is very efficient, because it uses only the instruction stack behavior, which is a constant number, to model the data-flow. This process is not valid for loops with *goto* and *break*, but we do not allow these constructions in the input code, Section 5.3 contains details.

Algorithm 6 defines the symbolic CIL emulation process. I is a set of arrays that define the setup information for each node (a positive number) – the number of *true* and $false$ evaluations for jumps and identifiers for the other instructions. The *applicable* and *apply* functions check and apply the symbolic rules, they implement the graph rewriting system functionality. The *extract_ids* function obtains the values consumed by the node (dashed boxes in Figure 6.5).

The conditional jumps are evaluated based on their identifier x according to the formula $x \geq 0$, which means that the emulator jumps while the identifier is positive. This way we change the number of loop iterations, because the loop

Algorithm 6 Symbolic emulation

Require: G – CIL sequential graph
R – symbolic rules
I – list of setup information
Ensure: ID – list of collected IDs

1: **for all** $i \in I$ **do**
2: $G_s := G$
3: **for all** $v \in V(G)$ **do**
4: $V(G_s) := V(G_s) \cup \{i[v]\}$
5: $E(G_s) := E(G_s) \cup \{(i[v], v)\}$
6: **end for**
7: **while** $\exists r \in R : applicable(G_s, r)$ **do**
8: $G_s := apply(G_s, r)$
9: **end while**
10: $ID := \emptyset$
11: **for all** $v \in V(G)$ **do**
12: $ID[v] := (v, extract\_ids(v), i[v])$;
13: **end for**
14: **end for**

ends once the jumps identifier is $x < 0$.

Result of the algorithm is an aray of sets of consumed and produced values for each node / instruction in the CIL sequential graph. The array is constructed using the function *extract_ids* that extracts the identifiers of the nodes / instructions. The lists are used to construct the consumer sets that are the output of the entire data-flow analysis.

6.6.4 Construction of the CIL Sequential Graph

The construction of the CIL sequential graph is straightforward and requires only the code sequence and static information about instructions, we do not need any additional analyses. The basic transformation of a CIL code to a CIL sequential graph requires the following four steps:

1. Create nodes based on the instructions.
2. Connect the instruction nodes based on the control-flow sequence.
   - Standard instructions follow one another.
   - Conditional jumps have two outgoing edges, one to the jump target (specified by its operand) and the other leading to the next instruction.
   - Fixed jumps have only one edge going to the jump target.
3. Create special nodes for stack, program counter, variables and arguments.
4. Connect the instruction nodes to the special nodes based on their data-flow behavior.
   - Program counter node is connected to the first instruction in the code.
   - All instructions are connected to the stack.
   - Instructions that access a variable (based on their operand) are connected to its node.

The entire transformation is in Algorithm 7. The instruction set I and variable set M are extracted from the source code and the sets J and F contain instruction codes of all conditional and fixed jumps respectively, as defined in CIL standard [30]. Each instruction contains information about its code and operand accessed using the comma operator. The transformation can be illustrated on the example of the $factorial$ method, Listing 6.4 contains the CIL source code and Figure 6.3 shows the resulting graph.

6.6.5 Instruction Classifier

The *instruction classifier* accepts a single sequence of CIL instructions as its input and it produces a sequence of *classification tokens*. A *classification token* is a tiny structure that contains a *classification* and the respective instruction, if relevant. The classification defines the instruction type and it is a value from the following list:

- $INSTRUCTION$ – *Basic instruction* that does not change control-flow.
- $FIXED$ – Instruction representing a fixed (non-conditional) jump.
- $CONDITIONAL$ – Instruction representing a conditional jump.
- $TARGET$ – Target place of a single jump. The token is not associated with an instruction, because it represents a space between instructions.

We processes the code one instruction at a time and using a table we assign each instruction a classification token. We create a sequence of the *classification tokens* representing the instructions, classified as either the $INSTRUCTION$, $FIXED$ or $CONDITIONAL$. Then we locate all the jumps and for each jump, we add a token with the $TARGET$ classification just before the instruction that

Algorithm 7 Transformation of CIL to Sequential HFG

Require: I – sequence of instructions (source code)
M – set containing all variables and arguments
J – conditional jump instruction codes
F – fixed jump instruction codes
Ensure: G – CIL sequence graph
{Special nodes}
$V := \{P, STACK\} \cup \{V_m : m \in M\}$
{Instruction nodes}
$V := V \cup \{V_i : i \in I\}$
{Control-flow edges}
$E := \{(P, V_{0 \in I})\}$
for all $i \in I$ **do**
  if $i.code \in J$ **then**
    $E := E \cup \{(V_i, V_{i.operand \in I})\}$
    $E := E \cup \{(V_i, V_{i+1})\}$
  else if $i.code \in F$ **then**
    $E := E \cup \{(V_i, V_{i.operand \in I})\}$
  else
    $E := E \cup \{(V_i, V_{i+1})\}$
  end if
end for
{Data-flow edges}
for all $i \in I$ **do**
  if $i.code \in \{stloc, ldloc, starg, ldarg\}$ **then**
    $E := E \cup \{(V_i, V_{i.operand \in M})\}$
  end if
end for
$G := (V, E)$

Listing 6.6: Lexer tokens

```
IL_0000: ldc.i4.1                    (INSTRUCTION, ldc)
IL_0001: stloc.0                     (INSTRUCTION, stloc)
// loop start (head: IL_0002)        (TARGET, null)
IL_0002: ldarg.0                     (INSTRUCTION, ldarg)
IL_0003: ldloc.0                     (INSTRUCTION, ldloc)
IL_0004: mul                         (INSTRUCTION, mul)
IL_0005: stloc.0                     (INSTRUCTION, stloc)
IL_0006: ldarg.0                     (INSTRUCTION, ldarg)
IL_0007: ldc.i4.1                    (INSTRUCTION, ldc)
IL_0008: sub                         (INSTRUCTION, sub)
IL_0009: starg.s a                   (INSTRUCTION, starg)
IL_000b: ldarg.0                     (INSTRUCTION, ldarg)
IL_000c: ldc.i4.1                    (INSTRUCTION, ldc)
IL_000d: bgt.s IL_0002               (CONDITIONAL, bgt)
// end loop
IL_000f: ldloc.0                     (INSTRUCTION, ldloc)
IL_0010: ret                         (FIXED, ret)
```

is the operand of the jump (its target).

Listing 6.6 shows the classification tokens for the *factorial* method defined in Listing 4.1. The token *TARGET* is not assigned to an instruction, it is placed between instructions because it represents the target of a jump.

6.7 Control Flow Analysis

In this section, we will explain in detail the final step of the front-end of the ParallaX compiler – the *control-flow analysis*. We start by analyzing the available control flow constructs and their representation in a hybrid flow graph. Then we explain the algorithms used to transform the procedural code to a hybrid flow graph, based on the control-flow structure.

The *control-flow analysis* consumes the output of both the *instruction classifier* and *data-flow analysis* and produces a hybrid flow graph, the graph is gradually expanded as the code is analyzed. We parse the code, one instruction at a time, and we either add the new instruction directly to the graph (in case of basic instructions), or we create a control flow sub-graph and connect it to the main graph. *Basic instructions* do not influence the control-flow, they are directly transformed into an appropriate node and then connected according to data flow. Control flow constructs are handled in three steps:

1. Recursively create the sub-graph representing the loop or branch body.
2. Add special operations that handle the data flow.
3. Connect the created graph into the final HFG according to the data-flow.

One special construct is the HFG *start* node. It is used to start the entire

graph and it does not have any input from inside of the graph. It sends an input to nodes that do would not have an input otherwise, like a load constant. The starting impulse must be specially handled in control flow, because in branches only some operations must be started (in active branch) and some must be restarted (in loops).

6.7.1 Branch Infrastructure

Each *conditional branch*, also known as *if-else*, is composed of a condition and two branches (represented by sub-graphs), which can be empty. The basic infrastructure used to represent a branch is shown in Figure 6.6 (left), it is the only interface between the branches and the rest of the HFG.

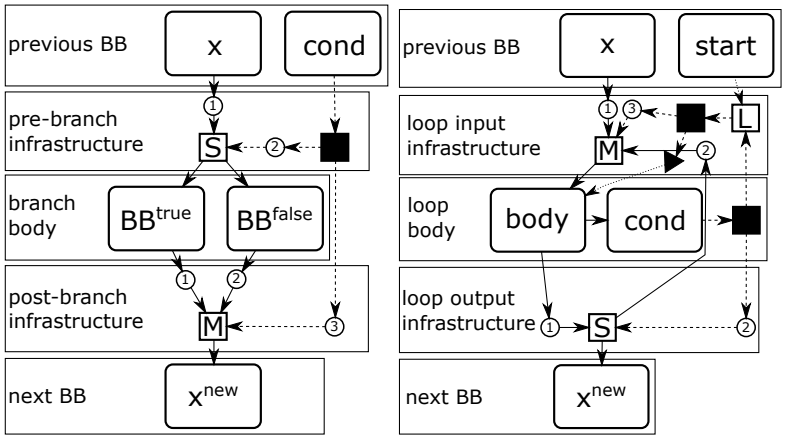

Figure 6.6: Branch (left) and loop (right) infrastructure

Figure 6.6 shows the branch with the surrounding code, where the previous basic block and both branches are already constructed and the next basic block is connected once it is ready. The infrastructure is built around the branches and it serves as their only interface with the rest of the HFG. The figure defines the infrastructure for a single variable. The special nodes (*pre-branch* and *post-branch* sections) must be replicated for each variable used in the branch.

For branches we use combination *split-merge*, where the *split* separates data for branches and *merge* collects them at the end. Split-merge interface is created only for values used in the branch (accessed variables or arguments). The *split* and *merge* nodes are defined in Section 4.3 as part of the Hybrid Flow Graph semantics.

The infrastructure must make sure that the right branch is executed, based on the condition. And it must ensure that only one branch is executed at a time, no nodes from the other can executed, because that would mean that unused values may remain in the graph and make further computations skewed or completely invalid.

The *starter* node is handled like a variable, by a split-merge, and it is added only if a starter impulse is necessary (when a branch contains a node without input, like a load constant).

If one of the branches does not modify a variable, we simply connect the split to the merge, thus the value is transfered unchanged. When a variable is not

used in any of the branches, we skip this entire step since no special operations are necessary.

6.7.2 Loop Infrastructure

A *loop* contains a body, a condition and a backward conditional jump. We consider only *do-while* loops, because the others can be constructed from it using conditional branches (if-else). The infrastructure used to represent a simple loop is in Figure 6.6 (right) and it is the only interface between the branches and the rest of the hybrid flow graph.

For loops, we use combination *merge-split*, where *merge* selects the input and *split* distributes the output of the loop. Split returns the produced values into the loop and outputs the last value when the loop ends. Merge must make sure that the first iteration uses the value produced before the loop and the split value is used for the other iterations. The special nodes (*loop input* and *loop output* sections) must be replicated for each variable used in the loop.

The loop requires a *loop primer* operation that controls the input and output. It is defined in Figure 4.6 and it instructs the input merge to first use the initial input and then use the output of the split while the loop is running.

Starter is necessary in a loop to repeatedly start nodes without other input and we use the output of the loop primer as a starter. The only thing we must do is to convert it from boolean to the starter token via a conversion node (filled triangle node in Figure 6.6).

6.7.3 Control-Flow Analysis Overview

The *control-flow analysis* consumes a single sequence of the *classification tokens*, produced by the instruction classifier, and the *consumer sets*, produced by the data-flow analysis. The output of the control-flow analysis is a hybrid flow graph. We parse the classification token sequence using a pushdown automaton, commonly used by syntax analyzers in parsers, and we use the consumer lists to connect the graph as it is constructed. It is important to note that the CIL structure closely follows the original C# control flow and therefore we can parse it by instructions without loosing information.

The automaton analyzes control-flow constructs in the code, where we recognize: *conditional branches* (if-else), *loops* and *basic instructions* (without control-flow behavior). The automaton creates subgraphs, representing the constructs, and connects them to the final hybrid flow graph, which starts empty at the beginning. The basic instructions are directly converted to nodes and added to the final graph. For the branches and loops, we first construct the graphs representing their bodies, then we add the infrastructure explained in Sections 6.7.1 and 6.7.2 and finally we connect the product to the final hybrid flow graph.

In this section, we continue to use the method $factorial$ defined in Listing 4.1 as an example to explain the functionality of our compiler. When we apply the automaton to the methods classification stream, it first creates nodes representing the basic instruction *ldc*. Next, we encounter a loop and the automaton constructs a HFG representing its body, Figure 6.7 left, and then it adds the loop infrastructure, Figure 6.7 right.

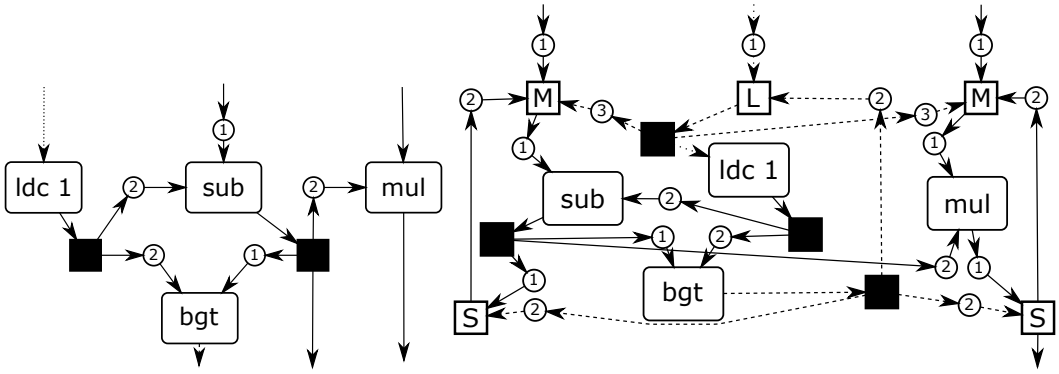

Figure 6.7: *factorial* loop body (left) and infrastructure (right).

Finally, we connect the loop to the *ldc* node and complete the graph by adding the return *ret* node. We eliminate instructions like load local (*ldloc*), because they do not modify data, they are removed by the optimizations presented in the next chapter. The complete graph is in Figure 6.8.

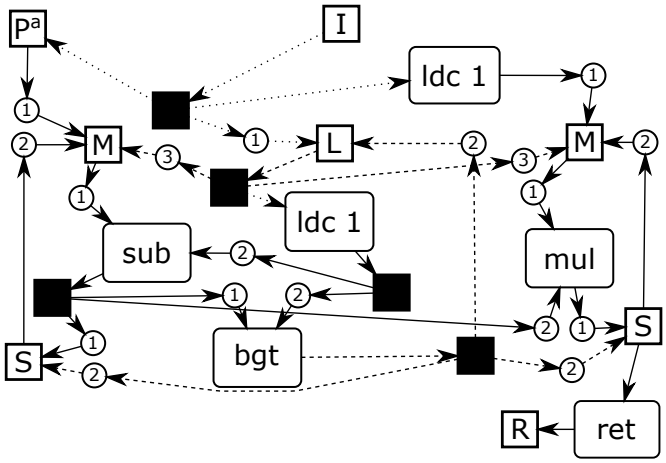

Figure 6.8: The HFG representation of the *factorial* function

6.7.4 Hybrid Flow Graph Construction

To analyze the control flow, we designed a pushdown automaton based on a context-free grammar. The grammar takes the *classifications*, produced by the *instruction classifier*, as *terminals* and assigns semantic values to the *non-terminals* like a purely-synthesized attribute grammar. The automaton is generated from the grammar in Listing 6.7, where the used functions are explained in the following text. The functions are defined using pseudo-code algorithms, which contain the implementation without technical details that would make them too long.

For basic instruction, the *AddBasicNode* function creates a small graph containing a node representing the instruction and its input nodes connected by edges. The function is defined in Algorithm 9. The *block* rule then connects the new graph with the main HFG using the *MergeGraph* function defined in Algorithm 9.

Listing 6.7: Parser grammar

```
start:
        block
        { $$ = CompleteGraph($1); }
;
block:
        block construct
        { $$ = MergeGraph($1, $2); }
        | construct
        { $$ = $1; }
;
construct:
        instruction
        { $$ = $1; }
        | if
        { $$ = $2; }
        | loop
        { $$ = $2; }
;
instruction:
        INSTRUCTION
        { $$ = AddBasicNode($1); }
;
if:
        CONDITIONAL block FIXED TARGET block TARGET
        { $$ = AddBranch($1, $2, $5); }
        | CONDITIONAL block TARGET
        { $$ = AddBranch($1, $2, EmptyGraph); }
;
loop:
        TARGET block CONDITIONAL
        { $$ = AddLoop($4, $2); }
;
```

Algorithm 8 AddBasicNode

Require: REG – associative register mapping instructions to nodes
I – CIL instruction
IN – the number of inputs of I
Ensure: G – graph representing i
REG' – updated register

1: $N := \{V_I\} \cup \{I_n : n \in \mathbb{Z} \wedge n \geq 0 \wedge n < IN\}$
2: $E := \{(I_n, V_I) : I_n \in N\} \cup \{([I], I_n) : I_n \in N\}$
3: $REG[I] := V_I$
4: $G = (N, E)$
5: $REG' = REG$

The graphs are created with input edges that do not have source, but they are annotated with the identifier of the value they represent, $[I]$ in the algorithms. These *empty edges* are connected according to the *consumer sets* when the subgraph is merged to the HFG. We start HFG creation with the first instruction, which has no inputs because the stack is empty, and then we merge others to it. During the construction, we continually update a register mapping the constructed nodes to the original instructions so we can properly connect them according to the data-flow analysis.

Algorithm 9 MergeGraph

Require: REG – associative register mapping instructions to nodes
SET – consumer sets, produced by data-flow analysis
G – previous version of HFG
S – subgraph
Ensure: HFG – new HFG
1: $HFG := (G.N \cup S.N, G.E \cup S.E)$
2: $Empty := ((I, V) \in HFG.E : I \notin HFG.V)$
3: $HFG.E = HFG.E \setminus Empty$
4: **for all** $(I, V) \in Empty$ **do**
5: $\quad HFG.E = HFG.E \cup \{(REG[SET[I]], V)\}$
6: **end for**

The function *AddBranch*, used in the rules for conditional branches (*if-then-else* and *if-then*), creates a subgraph consisting of the conditional jump instruction, and either one or two branches. The jump instruction itself is converted as a basic instruction, because it contains the condition (e.g. *bgt* – branch if greater than). Then we create the split-merge infrastructure, explained in Section 6.7.1, and we connect the branch inputs to the *splits* and the outputs to the *merges*. The resulting graph is then merged to the final HFG using the $MergeGraph$ function.

The $AccessedVariables$ function collects all the variables and arguments accessed by at least one node in a graph. The accessed variables are then used to construct the split-merge infrastructure for the conditional branch, which is handled by the $AddSplitMerge$ function. The merge-split infrastructure replaces the variables in the register (REG) with the appropriate merge or split output.

The rule for *do-while* loops is similar to that for the conditionals, but it uses the loop infrastructure described in Section 6.7.2.

The rule for the complete graph connects the HFG after all instructions and control flow constructs are processed. The function $CompleteGraph$ adds a global starter node (I), parameter nodes (P^x) and a return node (R). We add these nodes here because it is the root non-terminal that has to produce a complete hybrid flow graph.

6.7.5 Broadcast Introduction

One last step is necessary to make the final hybrid flow graph valid, we have to introduce broadcast operations, which ensure that every other operation except *split* has just a single output. The process is defined in Algorithm 12, where a broadcast is introduced for every node with more than one output.

Algorithm 10 AddBranch

Require: REG – associative register mapping instructions to nodes
SET – consumer sets, produced by data-flow analysis
J – conditional jump
$B0$ – false branch
$B1$ – true branch
Ensure: G – new HFG
1: $G := AddBasicNode(REG, J)$
2: $Values := AccessedVariables(REG, G1, G2)$
3: **for all** $val \in Values$ **do**
4: $\quad G := MergeGraph(REG, SET, G, AddSplitMerge(REG, val))$
5: **end for**
6: $G := MergeGraph(REG, SET, G, B0)$
7: $G := MergeGraph(REG, SET, G, B1)$

Algorithm 11 AddLoop

Require: REG – associative register mapping instructions to nodes
SET – consumer sets, produced by data-flow analysis
J – conditional jump
B – loop body
Ensure: G – new HFG
1: $G := AddBasicNode(REG, J)$
2: $Values := AccessedVariables(Body)$
3: **for all** $val \in Values$ **do**
4: $\quad G := MergeGraph(REG, SET, G, AddMergeSplit(REG, val))$
5: **end for**
6: $G := AddLoopPrimer(REG, G)$
7: $G := MergeGraph(REG, SET, G, B)$

Algorithm 12 AddBroadcasts

Require: G – complete hybrid flow graph
Ensure: H – modified HFG
1: $V := V(G)$
2: $E := E(G)$
3: **for all** $v \in V : v.code \neq split \wedge |\{o \in V : \{(v, o) \in E\}\}| > 1$ **do**
4: $\quad OUT := \{o \in V : (v, o) \in E\}$
5: $\quad E := E \setminus \{(v, o) \in E : o \in OUT\}$
6: $\quad V := V \cup \{b : b.code = broadcast\}$
7: $\quad E := E \cup \{(v, b)\} \cup \{(b, o) : o \in OUT\}$
8: **end for**
9: $H := (V, E)$

6.7.6 Source Code and HFG Equivalence

Finally, we present sketch of a proof that the HFG, produced by the presented algorithm, is equivalent to the source code.

To prove the equivalence, we must prove 1) that each instruction is executed in a HFG if and only if it would have been executed in the source code, 2) that each HFG operation produces the same output (including the order of values) as its CIL counterpart and 3) each operation can be executed multiple times without influencing its behavior.

When we prove both, we can conclude that when a HFG is executed, the same instructions are executed, they get the same input and produce the same output as they do in the source code. This means that the outputs are the same and no additional side effects were introduced.

The second requirement is fulfilled by the fact that basic operations are defined exactly according to a CIL instructions. They produce the same output with the same input in the same order - when triggered multiple times with different inputs it produces the same sequence of outputs as the original. The only special case is instructions without input, like *ldc* (load constant). These operations have input in HFG so they can be triggered by control flow and the first requirement makes sure that they are executed exactly as they are in the source code.

The first requirement can be proven via induction over basic blocks. Basic blocks in a HFG are subgraphs containing only basic operations. The blocks are connected by special operations and their interface is a set of live variables.

The operations in a basic block are executed only when they receive input. This means that all the operations are executed once they get all the values in the block interface, because each operations gets an input (either from the interface or from another operation). The starter node is special, we consider it a separate block executed automatically. This behavior is the same as CIL, where all instruction in a block are executed once the block starts.

Next we consider the branch infrastructure (Figure 6.6) and the semantics of the *split* operation. The infrastructure contains split for every variable accessed in any branch and once the these splits get input, they pass the data to just one of the branches. They always select the same branch as they have the same input condition. The selected branch is executed, while the other is not. This behavior is the same as in CIL.

Loop infrastructure (Figure 6.6) is controlled by a set of split operations placed at the end of the loop body. These splits either return the data to the loop or pass it to the next basic block, which controls when the loop ends (block that gets the data is executed). Their behavior is controlled by the loop condition. The loop ends once the condition is *false*, just as in CIL.

The final step is inductive, we have proved that a basic block is executed once it gets inputs and that the control flow infrastructure distributes inputs properly. Now we use induction over the control flow complexity: 1) single basic block executed (via starter) 2) loop and branch distribute inputs and thus start the correct basic block (body/branch or the next one). Thus every basic operation is executed if and only if it is executed in the source code.

6.8 Data types

The Hybrid flow graph is strongly typed to prevent unpredictable or incorrect behavior. The data types are assigned to edges, where each edge has a single type that defines what data can be transfered over it. The information is passed to the streaming environment plan that uses it to properly define the data streams. In HFG, we use the data types based on the types available in C# and we then map the types to the data types of the target platform, C++ types in case of the Bobox system.

We extract the data types during the data flow analysis, see Section 6.6, where we identify both the consumer identifier and the incoming type. The data types of all potential producers for a single instruction must be identical or implicitly convertible (like *int* to *long*) and the compilation fails if the types do not match. It is important to note that all the CIL instructions have a single output, with the exception of the instruction *dup* that duplicates its input and all its outputs have the same type as its input.

The data-flow analysis extracts the output type for each instruction using the following rules that assign data type to each instruction based on its operation code and operand:

- constant – the type is defined by the instruction suffix ($I4$ for $Int32$ etc.)
- conversions – the type is defined by the suffix
- load / store variable – the type is identical to the type of the variable
- load / store argument – the type is identical to the type of the argument
- arithmetical operations – the type is identical to the type of the inputs
- conditional jumps – the output is always *bool*
- fixed jumps – no output, the jumps are never part of the HFG

Data types for edges connecting basic nodes (representing CIL instructions) are all gathered in the data flow analysis. Control flow introduces special nodes that are not part of the original CIL and their data types are dependent on the basic nodes they connect. We solve this situation in the control-flow analysis, where we assign the edges data types once the branch or loop infrastructure is connected to all its inputs and outputs. When the infrastructure is connected, we propagate the data type from inputs, which are typed by the data flow analysis, through the special operations to all the edges.

6.9 Array Support

The compiler supports one dimensional arrays, treating them as any other data type. The compiler supports only one dimensional arrays to limit potential aliasing of the embedded arrays, which is extremely difficult to trace. Most importantly, multi dimensional arrays can be represented in an one dimensional array by appending the embedded arrays one after another. The process is illustrated in Figure 6.9, where the elements can be accessed using the following

formula: $array[y * width + x]$. Thanks to this limitation, the data flow analysis is able to track the arrays through their entire lifetime and eliminate aliasing.

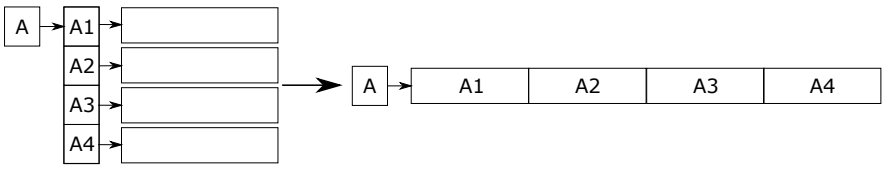

Figure 6.9: Array flattening.

Arrays are transported as other data types, an entire array is a single value in the stream. The elements are extracted by kernels only when needed. This approach is not very efficient, especially when the array is accessed sparsely, but it allows us to support the arrays in any control flow structure including multiple nested loops. Without this approach, it would be impossible to manage loops containing both scalars and arrays, since the array streams would be longer and it would not be possible to match the values and loop iterations. The efficiency is addressed by optimizations that unpack the arrays in situations where it is possible and they restructure the graph around them so that the expansion is managed properly, see Sections 7.

We do not support objects, because they cause similar problems as the multi-dimensional arrays, and they are not necessary to implement the data intensive applications or database procedures. The purpose of the compiler is not to execute any C# code in a streaming environment, but to allow programmers to implement streaming applications using a restricted C#.

6.10 Aliasing

The compiler strongly restricts the input language to limit the effects of aliasing, where a single memory location can be accessed via multiple symbolic names (pointers or references). There are two possible causes in of aliasing in a C# code, multiple references can point to a single object or multiple array indices can point to the same array in a multi-dimensional array. There are also pointers in unsafe C# code, but we do allow the unsafe code at all.

Value types, like numbers, cannot cause aliasing, because they cannot be accessed via references of indices and we do not allow *boxing*[1]. We allow only a single data type with reference semantics – single-dimensional arrays. However; we handle the arrays in a way that they cannot introduce aliasing, because we always transport the entire array as a single value and our data analysis tracks the array from its creation and through all the statements that can access it. The process is explained in Section 6.9.

There is one situation, where aliasing can happen: the parameters of the entire application can be single-dimensional arrays and multiple parameters can contain a reference of the same array. There is no way for our data analysis to decide if the parameters are aliased and we assume that they are never aliased.

[1]Boxing is a process that converts a value type into a reference type by wrapping it into a simple object. The process is used to allow the use of value types in contexts that require reference types, like some containers.

This assumption may cause us problems in general applications, but we focus on the data intensive applications and database queries, which can be analyzed and we can ensure that the parameters are not aliased.

7. Optimization

The previous chapter explained the front-end of the ParallaX compiler that produces a hybrid flow graph from C# code compiler to CIL. The produced HFG can be, thanks to its structure, almost directly executed using a streaming system. However, the HFG produced by the front-end does not offer much kernel parallelism as there is always at most one data element at each edge. In addition, the communication overhead in the streaming system would probably cost more than the sequential execution of the original code, since all instructions are represented by separate nodes. Fortunately, there are several transformations which may improve the level of parallelism available as well as reduce the synchronization overhead.

All the optimizations are a special case of a process we call the *component extractions*. The optimizations are implemented as rules of an *associative graph rewriting system* that modifies a *layered hybrid flow graph*, where the HFG produced by the front-end can be considered a layered HFG with a single layer. The produced layered HFG contains *custom operations* implemented by a subgraph that they replace, which means that they represent more complex nodes without communication overhead (we rely on the C++ compiler to place the variables into registers). For a detailed description of the layered hybrid flow graph, see Section 4.5.

In streaming environments, the custom operations become components (see Figure 5.2) implemented in a single kernel, avoiding the communication overhead. In addition some custom operations can allow other operations to be vectorized, by replacing control-flow constructs, mainly loops.

The optimization process used in the ParallaX compiler generally involves graph transformations, since our intermediate language, the Hybrid Flow Graph, is graph-based [5]. The optimizations we use can be divided in two main groups: general and HFG-specific optimizations [1]. The general optimizations fulfill a similar function as those used in traditional compilers, despite being implemented as a graph rewriting system.

We demonstrate the optimization technique on a *vectoradd* function that adds two vectors and stores the result in third vector. The source code is in Listing 7.1. We transform the source code to a HFG and use the presented optimizations to improve the efficiency of the resulting graph, which is in Figure 7.1. The graph is gigantic and inefficient and we will gradually improve it as we introduce and apply the optimizations.

In unoptimized HFG, any array is treated similarly to a elementary data type, it is passed around the graph, undergoing partial modifications by store instructions which change the array at the referenced positions. Although this behavior might be efficiently mimicked by the implementation if the underlying streaming system allows passing data by reference, this is certainly not as effective as required. Therefore, the optimizations described below try to recognize typical patterns in array handling and transform them into alternative representations which avoid passing the array as a whole. Consequently, the movement of the complete array is used only as a fall-back behavior when these optimizations fail, which is sometimes inevitable given the general undecidability associated with

Listing 7.1: Vector addition

```
int[] vectoradd(int[] a, int[] b, int[] c)
{
  for (int i = 0; i < c.Length; i++)
  {
    c[i] = a[i] + b[i];
  }
  return c;
}
```

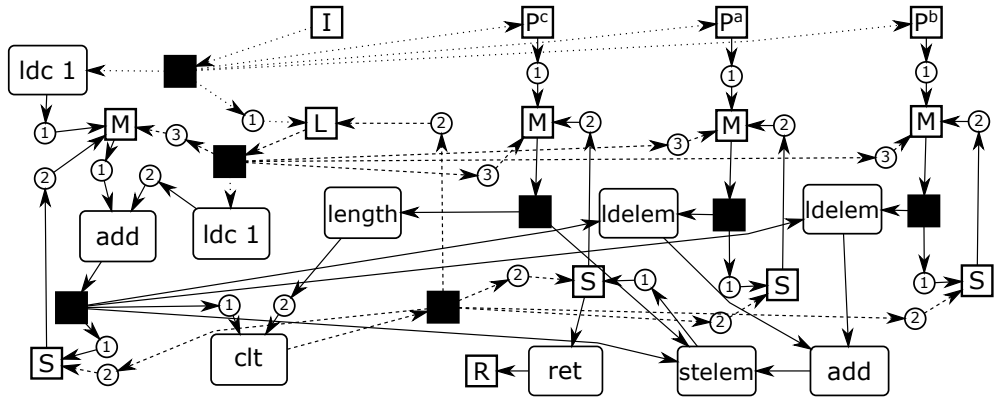

Figure 7.1: Hybrid flow graph representing the *vectoradd* function

Turing-complete programming environments.

The optimizations introduce the split between code and plan generation described in the compiler overview, Chapter 5. The split is defined by the layers of the produced layered HFG, where the top layer represents the execution plan and the layers representing the custom operations are transformed to kernels. Their implementation is defined by the subgraph they replace (the left hand side of the optimization rule, for example Figure 7.3). Both the plan and the separate operations are provided to the streaming environment for execution.

7.1 Related Work – Transformations Improving Parallelism

In the optimization stage, the compiler restructures the intermediate code produced by the front-end to produce a more efficient and compact code, which can be than transformed into an application. The goals of the transformations are similar to some phases known from traditional compilers: the optimizations focus on better memory management and simple code restructuring [57], vectorization [36] or general parallelization [72].

The Hybrid Flow Graph requires custom optimizations that modify its structure, usually reducing granularity of the operations. The basic optimization used in the compiler is the *component extraction*, which performs a single graph transformation, replacing a subgraph by another [1]. The general principle is then used to implement all the other optimizations.

The general optimizations focus mainly on vectorization of the array operations. We locate arrays that are read or modified in a loop following certain structure and we replace the inefficient array management with more efficient custom nodes, able to perform the computations using vector instruction. This process usually involves advanced numerical methods that study the ranges of array indices [36], but we avoid this by restricting the graph structure eligible for optimization. We require that the array is either read or modified (not both) and the index used is always the variable controlling of the loop plus a constant. This constraint does not currently allow for maximal vectorization, but it supports the most common patterns, see Chapter 11 for experimental results. The optimizations will be gradually improved to include more complex code.

Compilers can contain even more advanced optimizations that significantly modify the structure of the intermediate code. These optimizations include transformation, like loop skewing [36], where the loop iterations are restructure to provide more predictable behavior. Our optimizations always restructure the HFG, because they rely on graph rewriting, but the current implementation does not yet support transformation that would completely restructure the code. One potential example is the use of the parallelogram blocks to optimize the matrix-based dynamic algorithms (like Levenshtein distance) [2], which is our main case study. At the moment, we apply this transformation manually (at the input C# code), but it is theoretically possible to do so automatically, which is one area of our future study.

The HFG specific optimizations focus on removing the special operations introduced by control-flow, because they usually prevent efficient vectorization and may limit kernel parallelism. We apply these optimizations after the graph has been vectorized, because we can eliminate the control-flow infrastructure only if the surrounding graph no longer needs the them. The actual process is very similar to the *dead code elimination* commonly used in compilers [57, 36, 84].

7.2 Component Extraction

The *component extraction* is an operation that transforms a layered hybrid flow graph to another layered HFG. The transformation itself is implemented as a rule of an associative graph rewriting system defined over layered hybrid flow graphs. The transformation combines multiple nodes into one complex node, reducing both the kernel parallelism and communication overhead. It can greatly improve throughput of the graph, but it can also limit parallelism, so it has to be used carefully. The concept is shown in Figure 7.2, where three arithmetic operations are replaced by a custom operation with the same behavior.

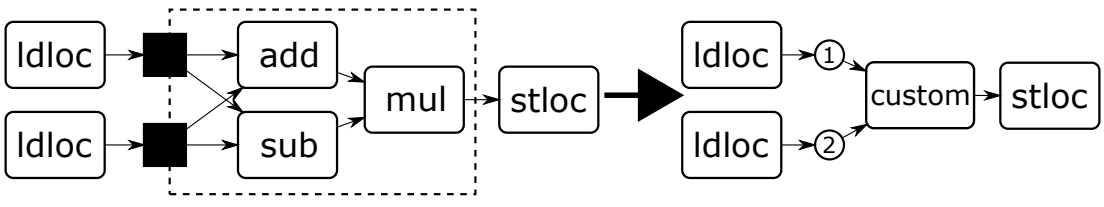

Figure 7.2: Component extraction application

The optimization can be applied to any subgraph, but we limit the application mostly to the bodies of loops which represent the most computation-heavy part of the code. This way, we are able to create one custom node for the loop body and vectorize the entire loop, if its structure supports it. The rule for the example in Figure 7.2 is in Figure 7.3, where the *custom*1 operation is implemented by the second layer, represented by the graph in dashed box titled *custom*1. The custom operation code is represented by the node $1 which represents the subgraph matched by the $[add|mul|sub|div]*$ node in the pattern graph.

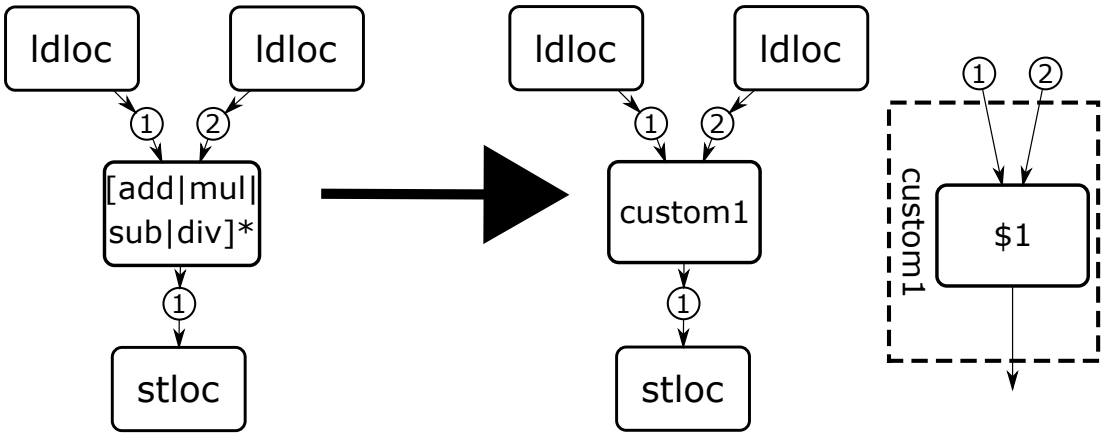

Figure 7.3: Component extraction pattern graph

The theoretical implementation of the custom node produced by this optimization is a custom operation and an embedded subgraph with properly matched inputs and outputs, where the embedded graph defines the semantics of the custom operation it implements. The actual implementation is more efficient and it will be explained in detail in Section 8 that focuses on the implementation of the compiler back-end.

The following optimizations are special cases of the component extraction adapted for common patterns or constructs. The optimization differ slightly from the general version, because they not only introduce a custom component, but they also optimize its implementation (the subgraph). This is possible, because the optimizations match a very specific subgraph with known properties.

7.3 Dead and Empty Nodes Elimination

The component extraction is our main optimization method, but we also introduced two additional optimizations – the *empty node elimination* and the *dead node elimination*. Both these additional optimizations are applied after each application of the component extraction to eliminate unnecessary nodes and make the graph more compact.

The *empty nodes elimination* is an algorithm that removes operations that perform no action and are not needed in the final hybrid flow graph. The algorithm removes all the operations representing the instructions loading and storing local variables and arguments, because they are no longer used to manage the variables, they simply copy their input to output. The optimization is defined by Algorithm 13.

Algorithm 13 Empty node elimination

Require: G – original HFG
Ensure: H – new HFG
1: $V := V(G)$
2: $E := E(G)$
3: **for all** $v \in V : v.code \in \{ldloc, stloc, ldarg, starg\}$ **do**
4: $IN := \{i \in V : (i, v) \in E\}$ // always just one
5: $OUT := \{o \in V : (v, o) \in E\}$ // always just one
6: $E := E \setminus \{(i, v)\ E : i \in IN\}$
7: $E := E \setminus \{(v, o)\ E : o \in OUT\}$
8: $V := V \setminus \{v\}$
9: $E := E \cup \{(i, o) : i \in IN \wedge o \in OUT\}$
10: **end for**
11: $H := (V, E)$

The *dead nodes elimination* recursively removes all the nodes without output, because HFG nodes have no side-effects, nodes without output have no purpose. All the nodes without any output are considered dead and are removed. This transformation is applied recursively. The complete process is defined in Algorithm 14.

Algorithm 14 Dead node elimination

Require: G – original HFG
Ensure: H – new HFG
1: $V := V(G)$
2: $E := E(G)$
3: $change := TRUE$
4: **while** $change$ **do**
5: $dead := \{v \in V : \{(v, o) \in E : o \in V\} = \emptyset\}$
6: $change := dead \neq \emptyset$
7: **for all** $d \in dead$ **do**
8: $E := E \setminus \{(a, d)\ E : a \in V\}$
9: $V := V \setminus \{v\}$
10: **end for**
11: **end while**
12: $H := (V, E)$

7.4 Range Extraction

First optimization we use is called *range extraction*. It replaces the complex structure of the loops *controlling variable* (usually i) with a *range* custom operation. The range operation has three inputs, initial value, limit and step, and it produces a sequence $(x_0, x_1, x_2, ... : x_0 := init \wedge x_{i+1} := x_i + step)$. The range has three outputs, *numerical sequence*, *boolean sequence* and a *final value*. All three sequences are the same length (determined by the loop condition) and the boolean sequence produces flags defining whether the numbers satisfy the comparison. The boolean

sequence is used to drive the remaining loop. The final value is output outside the loop and contains the final value of the control variable.

The pattern required by this optimization is a loop containing *add* node, a merge and a split node (plus some potential broadcasts). The rewriting rule used to implement this optimization is in Figure 7.4.

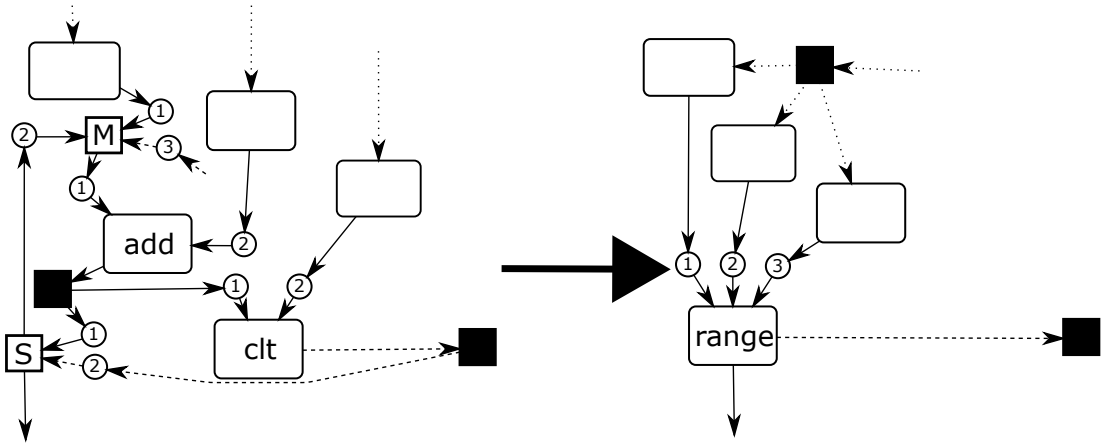

Figure 7.4: Range extraction pattern graph

Figure 7.5 shows the application of the range extraction optimization on the *vectoradd* function, the *add* node and the loop infrastructure are replaced by a *range* operation. All the unnecessary edges are removed.

7.5 Array Extraction

The second optimization we need is called *array extraction*. This optimization eliminates the constant array copying inside loops, removing unnecessary communication overhead. This optimization is based on custom operations *load array* and *store array*. These operations replace a cyclical subgraph that copies an array in every iteration with a single operation that caches the array and provides the values according to the *range* introduced by the previous optimization.

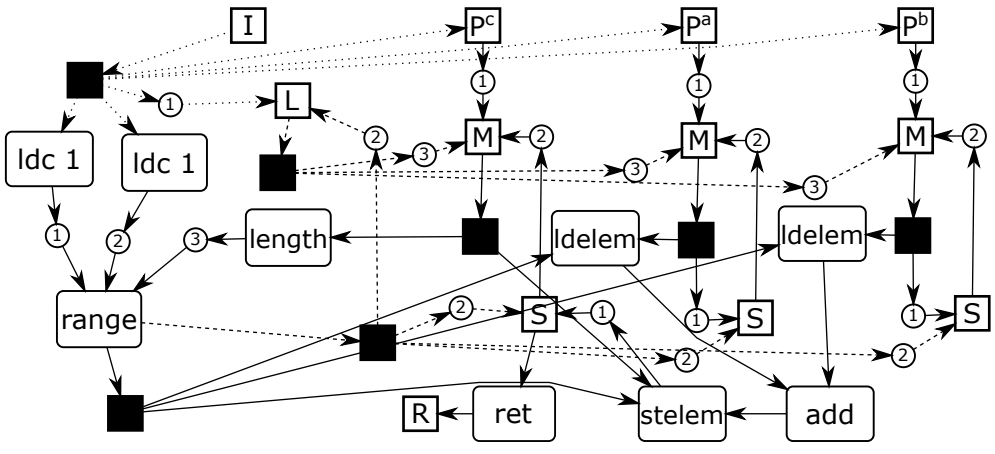

Figure 7.5: Range extraction application

The rules implementing this optimization are in Figures 7.6 and 7.7, where the *load array* and *store array* nodes replace parts of the loop responsible for the constant copying of the array. This optimization is always executed after the *range extraction*, and it ensures that the loop can be vectorized later.

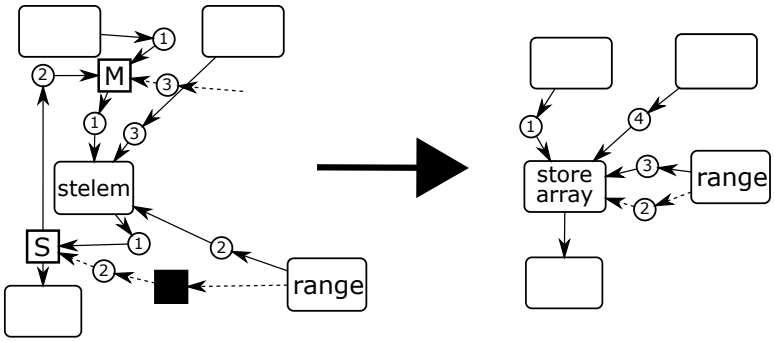

Figure 7.6: Array extraction rule for the *stelem* instruction

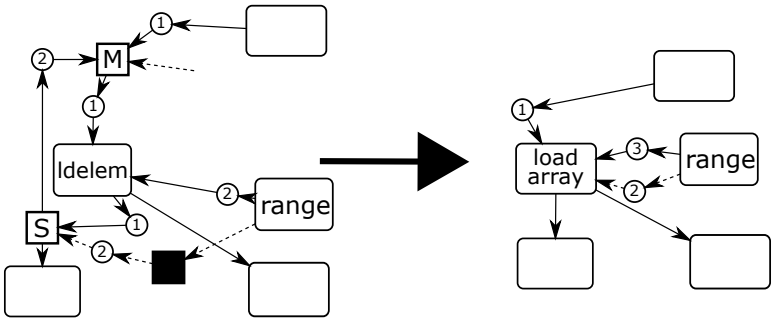

Figure 7.7: Array extraction rule for the *ldelem* instruction

Figure 7.8 shows the application of array extraction on a loop on the *vectoradd* function, we use both the rules repeatedly until we eliminate all the loops iterating the arrays. There are some dead nodes, mainly the loop primer (L), which will be removed in the next step.

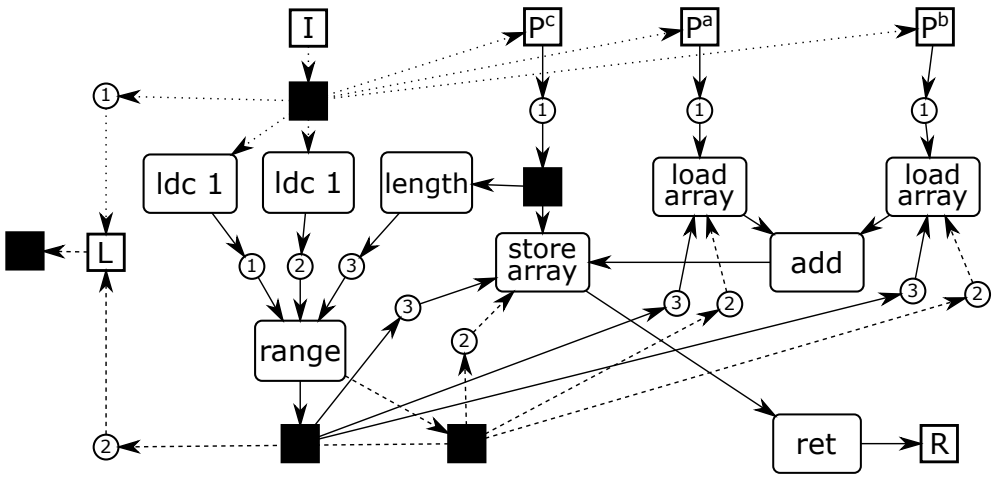

Figure 7.8: Array extraction application

Listing 7.2: Store array node

```
void store_array(stream in1, stream in2, stream in3,
                 stream in4, stream out1)
{
  while(!in1.eof())
  {
    var arr = in1.next();
    while(in2.next())
    {
      arr[in3.next()] = in4.next();
    }
    out1.put(arr);
  }
}
```

Listing 7.3: Load array node

```
void load_array(stream in1, stream in2, stream in3,
                stream out1, stream out2)
{
  while(!in1.eof())
  {
    var arr = in1.next();
    while(in2.next())
    {
      out2.put(arr[in3.next()]);
    }
    out1.put(arr);
  }
}
```

Pseudo-code implementation of the custom operations is in the following listings, *store array* is in Listing 7.2 and the *load array* is in Listing 7.3.

7.6 Token Extraction

Once all the arrays are extracted, we must make sure that the *Tokens* are distributed properly even when the loop is later vectorized. This means that we must replace the loop infrastructure for the *Tokens* with a vectorizable node that properly distributes the tokens according to the *range* node.

The rule implementing this optimization is in Figure 7.9, we just remove the *cast* node connected to the *loop primer* node and add a *token cast* node connected to the *range* node. This way we connect the last of the loops infrastructure to the range node and the loop primer can be safely removed.

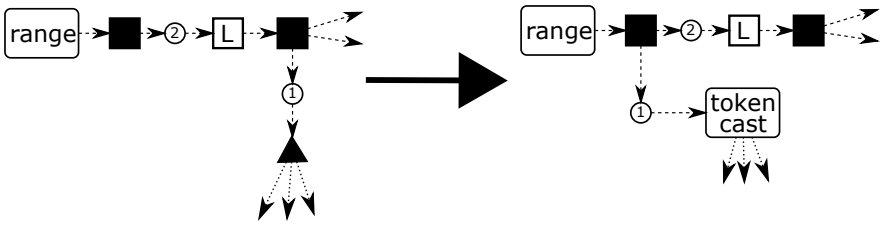

Figure 7.9: Token extraction rule for the *cast* special operation

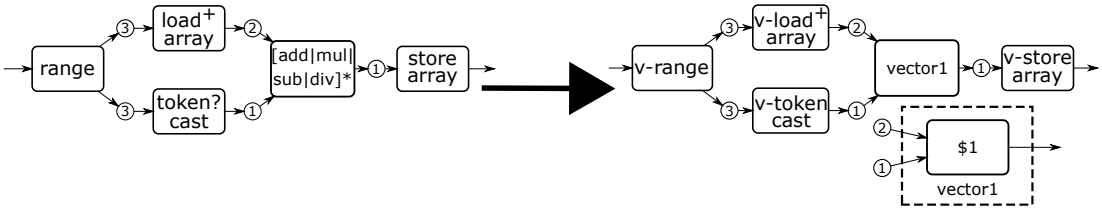

Figure 7.10: Vectorization rule

7.7 Vectorization

Finally, we introduce the *vectorization* to further improve the performance of the graph. Vectorization is almost impossible to use when a graph contains control flow nodes, because consecutive values may take different paths in the graph, which would break the vectors. However, the custom operations introduced by the optimizations are fully predictable and if they replace all the control flow of a loop then we can vectorize that loop.

This optimization takes the body of a loop, which has been completely transformed into custom operations, and transforms it into a single vectorized custom operation, unless it contains another nested loop or branch. The rule implementing the optimization is in Figure 7.10, where the loop body is replace with a new custom operation (*vector*1 in the example), unlike the other optimization, where we use a predefined custom operation designed specifically for the situation.

We illustrate the process on the *vectoradd* function. We take the hybrid flow graph representing the function and apply all the component extraction-based optimizations, the resulting graph is in Figure 7.8. Then we eliminate all the dead nodes and once the loop is completely removed, we can batch the data so that the *range*, *load array* and *store array* operations can process and produce data in vectors containing parts of the arrays not just single elements. We use the vectorization optimization to replace the loop with vectorized versions of the operations, which use vector instructions, like $\_mm\_add\_epi32(v1, v2)$ in SSE. The completely optimized hybrid flow graph is in Figure 7.11.

7.8 Array Extraction and Vectorization Chaining

We can further modify the array extraction and vectorization so they can be chained one after another. First, we introduce the *chained vectorization* optimization, by allowing the output to be a *ldelem* operation in a proper format.

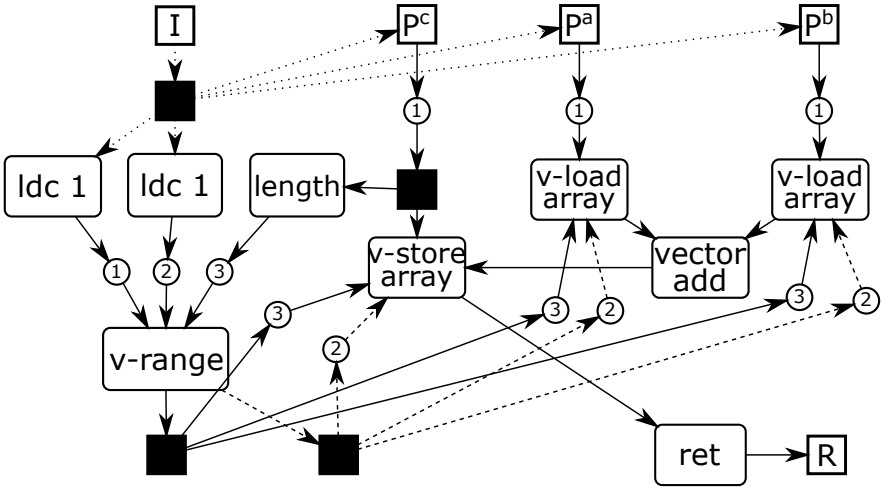

Figure 7.11: Fully optimized and vectorized *vectoradd* function

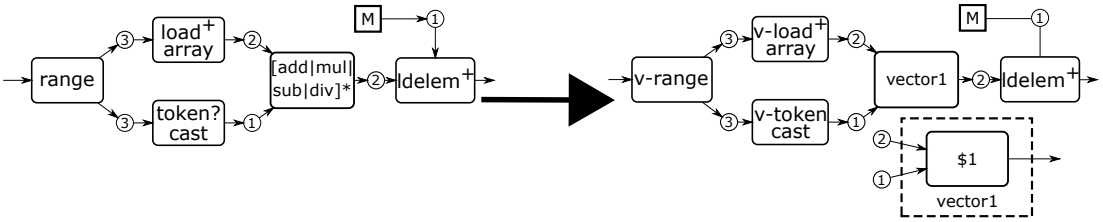

Figure 7.12: Modified vectorization rule

Figure 7.12 shows the modification, where the *ldelem* operation is required to be directly connected to the merge operation to ensure that there is no broadcast. Broadcast would meant that the array is accessed more than once in the loop and cannot be optimized due to potential data dependences.

Next, we introduce the *chained array extraction* optimization, which supports other custom operations as a source of the array index, not just the range operation. The rule is in Figure 7.13.

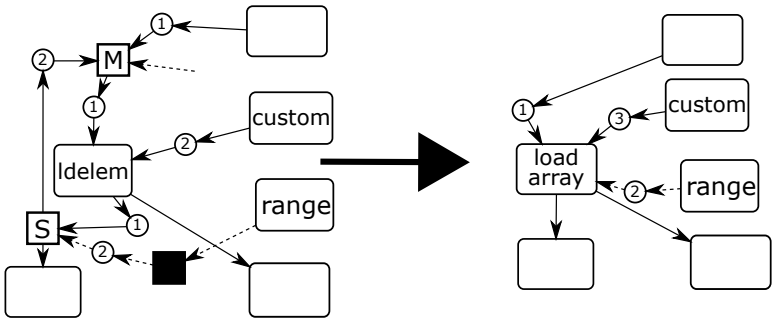

Figure 7.13: Array extraction rule for the *ldelem* instruction

8. Compiler Back-end

The compiler back-end transforms the optimized hybrid flow graph to a platform specific code. It must produce all the components necessary to execute the application in the environment, which usually means a machine code or byte code, but streaming environments are more complex. They require en execution plan and kernel implementations, usually in different language.

We present all the platforms currently supported by our compiler in Section 8.2. The following sections describe the specific back-end implementations for each platform.

8.1 Related Work – Compiler Back-end

The compiler back-end transforms the intermediate code into the code for the target platform. The standard practice is to transform the optimized intermediate code into machine or byte code and apply platform-specific optimizations to produce application best suited for the target environment [57].

The ParallaX compiler back-end transforms an optimized Hybrid Flow Graph into a Bobox application. We cannot implement the back-end of the compiler in the standard, straight-forward way, because the applications for Bobox framework must contain both an execution plan and the code of all used kernels (boxes) [2]. The beck-end is divided in two parts: component transformation and plan construction.

First we transform the subgraph representing the components, created using the *component extraction*, into C++ classes. The native implementation of the Bobox system supports kernels implemented as C++ classes [2] and the managed implementation uses C#. The process is similar to the call graph analysis [85], where the subgraph nodes represent procedures.

The Hybrid Flow Graph structure is very similar to the Bobox execution plan. We transform the top layer of the optimized HFG directly into Bobolang [3]. The process requires only the export of the graph in the correct Bobolang notation.

The produced application must be compiled with the Bobox runtime, which loads both the kernels and the execution plan and executes the application. The Bobox system creates a memory representation of the execution plan, instantiates the used kernels (boxes) and executes the plan. This process is similar to the modern dynamic languages, like C#, where the source code is compiled into an intermediate byte code, which is later executed by a CLR runtime [30].

The back-end traditionally performs platform-specific optimization [57, 36], but we can skip this step, because the optimizations are performed by the C++ compiler (for the boxes) and the Bobox environment (for the execution plan) [2].

8.2 Supported Environments

The compiler currently supports four target platforms. Each platform requires its own back-end implementation that is able to produce all the necessary components for the final application.

The compiler supports the native and managed implementation of the *Bobox streaming system*. Both implementations have similar basics and the main difference is supported language for the kernel implementation. The native implementation requires kernels implemented as C++ classes with the *basic_box* (provided by Bobox) as the base class, while the managed implementation uses C#. Both environments require an execution plan implemented in the *Bobolang* declarative language [3].

The compiler is able to parallelize the C# application directly by embedding the managed Bobox directly into the application assembly. The methods are transformed into plans, executed by the embedded Bobox.

The last currently supported platform are the .NET *asynchronous methods*. The asynchronous methods internally create a graph of parallel tasks executed by the .NET runtime. The HFG structure is similar to that of an application implemented using asynchronous methods, with the nodes representing separate methods and edges representing calls and we use this similarity to transform the hybrid flow graph to the asynchronous methods.

8.3 Bobox Transformation

The *Bobox* system is our main target environment and the back-end for Bobox performs two main operations. We transform the application HFG directly into a Bobox plan and then we add custom operations created by the *component extraction*. Component extraction groups the bodies of optimized loops into single vectorized operations.

All the kernels, based on CIL instructions, are already implemented in a library attached to Bobox. The custom operations must be handled separately, because we must provide Bobox with their implementation. They are transformed into a C++ class and attached to the Bobox system, which is then compiled into the final application. This approach allows us to generate C++ code and the C++ compiler performs all the necessary optimizations.

The transformation is the same for both the native and managed implementation of Bobox, the only difference is the programming language used to implement custom operations, C++ for native and C# for managed. The native implementation is our main platform, because it is more optimized for data intensive applications. The managed implementation is used mainly for testing.

8.4 Transformation Overview

Transforming the HFG into a Bobox plan is a straight-forward process and if there are no custom operations then the plan is the only thing necessary to complete the application, because the standard operations are already implemented in a library attached to Bobox.

The Bobox execution plan must be implemented in the Bobolang declarative language, which is described in detail in Section 2.2. The plan is divided into two sections, kernel and stream declaration. Kernels are declared along with their type, unique name, parameters, inputs and outputs (including their unique

identifiers). Streams connects declared kernels including the identification of a specific input or output, if the kernels have more than one.

It is important to note that we include in the HFG *input nodes*, which always represent the inputs of other nodes, perform no action and contain only identifier and data type. The inputs are added for all other nodes and every edges has either input as a source or a sink. The input nodes are used in the transformation fo Bobolang to match the stream inputs and outputs and they are removed in the process.

8.5 Execution Plan Construction

Algorithm 15 Hybrid flow graph transformation to Bobolang plan

Require: G – source HFG
Ensure: B – Bobolang plan

1: $B := "operatormain() -> (){"$
2: **for all** $n \in N(G)$ **do**
3: $IN := \{i \in N(G) : (i, n) \in E(G)\}$
4: $INPUTS := ""$
5: **for all** $i \in IN$ **do**
6: $INPUTS := INPUTS + "(\{i.TYPE\})[i\_\{i.ID\}]"$
7: **end for**
8: $OUT := \{o \in N(G) : (n, o) \in E(G)\}$
9: $OUTPUTS := ""$
10: **for all** $o \in OUT$ **do**
11: $OUTPUTS := OUTPUTS + "(\{o.TYPE\})[o\_\{o.ID\}]"$
12: **end for**
13: $B := B + "\{n.ANNOTATION\}\{INPUTS\} \rightarrow \{OUTPUTS\}\ \{n.ID\};"$
14: **end for**
15: **for all** $e \in \{ed \in E(G) : e.SINK.ANNOTATION = Input\}$ **do**
16: $ID := e.SINK.ID$
17: $B := B + "\{e.SOURCE\}[o\_\{ID\}] \rightarrow \{e.SINK\}[i\_\{ID\}]"$
18: **end for**
19: $B := B + "}"$

Algorithm 15 shows the entire transformation process using the C# string notation, where the $\{X\}$ part is replaced in the final string by the value of X. The algorithm first transforms nodes to Bobolang kernels, where we omitted the kernel parameters, since they are used only for the constants to indicate their values. Next, we transform the edges that connect a node through an input to another node, we use the input to match the stream ends.

The transformation is best demonstrated on an example, we take a small HFG and transform it to Bobolang plan. We transform the right graph in Figure 8.1 and the produced Bobolang plan is in Listing 8.1. In the graph, we omit input nodes for nodes with only one input and we perform the *component extraction* optimization described in Section 7.2, which replaces the arithmetical nodes with a custom operation.

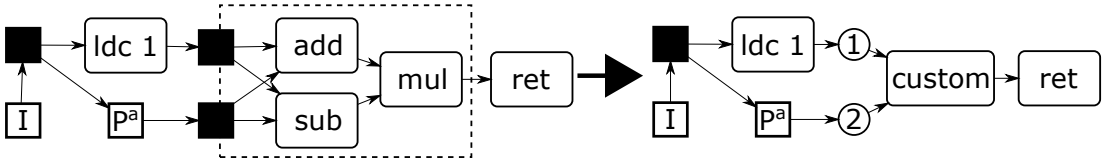

Figure 8.1: Optimized vector addition HFG

```
operator main()->() {
Start()->(token)[o_6] start_0;
Ldc(token)[i_7]->(int)[o_9] ldc_1(1);
Ldarg(token)[i_8]->(int)[o_10] ldarg_2("a");
Ret(int)[i_11]->() ret_3;
broadcast(token)[i_6]->(token)[o_7],(token)[o_8] broad_4;
CustomBody(int)[i_9](int)[i_10]->(int)[o_11] custom_5;

start_0[o_6] -> [i_6]broad_4;
broad_4[o_7] -> [i_7]ldc_1;
broad_4[o_8] -> [i_8]ldarg_2;
ldc_1[o_9] -> [i_9]custom_5;
ldarg_2[o_10] -> [i_10]custom_5;
custom_5[o_11] -> [i_11]ret_3;
}
```

Listing 8.1: Bobolang plan produced from the HFG in Figure 8.1

All the nodes in the Bobox plan are standard instruction kernels provided by our library or broadcasts built in Bobox, the only exception is the *CustomBody* node produced by the component extraction. This kernel has to be crated by the compiler back-end and provided along with the plan, which is explained in the next section.

8.6 Component Extraction

Component extraction is an optimization defined in Section 7.2, which reduces communication overhead of the final application by grouping multiple HFG nodes into a single custom node.

A graph without control flow can be directly transformed into a single custom operation with semantics defined by a C++ function. The transformation is performed in two steps. First, we sort the HFG nodes topologically and then we transform each node into a method call, one after another. The topological ordering makes sure that all the methods are called in correct order.

Figure 8.2 shows an example of the transformation, where the graph is topologically ordered top-to-bottom and left-to-right. The produced code calls only tiny methods representing CIL instructions, which are attached as a library and inlined by the C++ compiler.

Algorithm 16 transforms a HFG without control flow nodes to a C++ function. The algorithm constructs the code directly from nodes, where each node is represented by a call to a tiny function. We use C# string notation to build the

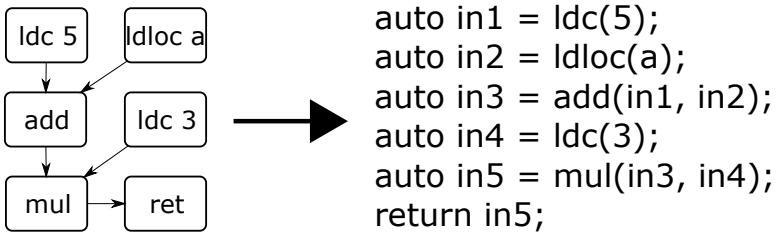

Figure 8.2: Transformation of a graph without control flow

Algorithm 16 Component transformation algorithm

Require: G – source HFG
Ensure: C – C++ code (sequence of statements)

1: $C := ""$
2: $NODES := topology\_sort(G)$
3: $REG := \emptyset$
4: $X := 0$
5: **for all** $node \in NODES$ **do**
6: $\quad REG[X] := node$
7: $\quad IN := \{"in" + REG[n] : n \in NODES \wedge (n, node) \in E(G)\}$
8: $\quad S := "auto\ in\{X\} = \{annotation(node)\}(\{join(',', IN)\})"$
9: $\quad C := C + S$
10: $\quad X := X + 1$
11: **end for**

statements, where the $\{X\}$ is replaced by the value of X.

In the Bobox environment, this requires that we create a new kernel class and compile it into the final application. We use Algorithm 16 to produce the code for the custom box. We can do this, because we perform the optimization only on the bodies of optimized loops that do not contain any control flow nodes. We produce code that uses only the basic instructions implemented as tiny functions immediately inlined by the compiler, which does not limit performance.

Once we have the source code for the kernel body, we create the class representing the kernel. This class is always very similar, the only difference is the *Body* method and the input / output streams. We transform the HFG to the source code of the *Body* method and add the rest of the class implementation. We omit the actual algorithm as its only job is to generate the input and output stream names and copy the body code.

Listing 8.2 shows the implementation of the custom kernel used in Figure 8.1. The *Body* method implements the component showed in the left graph in Figure 8.1, which is then extracted and replaced by the custom node, shown in the right graph.

```
class merge_box : public basic_box {
public:
  typedef generic_model<merge_box> model;

  BOBOX_BOX_INPUTS_LIST(in1, 0, in2, 1);
  BOBOX_BOX_OUTPUTS_LIST(out, 0);

  merge_box(const box_parameters_pack &box_params)
    : basic_box(box_params)
  {}

  virtual void sync_body() override
  {
    input_stream<int> in1(this, inputs::in1());
    input_stream<int> in2(this, inputs::in2());
    output_stream<int> out(this, outputs::out());

    while (!in1.eof() && !in2.eof()) {
      int l = in1.current()->get<0>();
      int r = in2.current()->get<0>();
      out.next()->get<0>() = Body(in1, in2);
    }
  }

  int Body(int in1, int in2)
  {
      auto i1 = add(in1, in2);
      auto i2 = sub(in1, in2);
      auto i3 = mul(i1, i2);
      return i3;
  }
};
```

Listing 8.2: Implementeation of the custom kernel in Figure 8.1

9. Additional Environments

The ParallaX compiler contains multiple back-end implementations that provide the support for other platforms, besides the managed and native implementation of the Bobox streaming environment. The compiler also supports the .NET asynchronous methods and a direct integration of the managed Bobox system. In this chapter, we will explain the basics of the other platforms and we will describe the implementation of the respective back-ends.

9.1 .NET Asynchronous Method

The *asynchronous methods* are parallel methods managed by the .NET framework. The methods are implemented and called as synchronous methods, but instead of immediately calculating their result, they return a *task* object that contains the information about the ongoing calculation. The framework introduces the *await* operator that pauses the caller, until the given task is completed and then it returns the result.

The asynchronous methods are designed to mainly improve responsiveness and throughput of applications that require time consuming operations, like network communication. They are not well suited to parallelization of calculation heavy or data intensive applications. As such, we use this platform mainly for testing and evaluation.

9.1.1 Transformation Overview

The back-end for this platform transforms a HFG into a C# code composed of asynchronous methods. The transformation process must handle three cases – a subgraph without control flow, a branch (split-merge) infrastructure and a loop (merge-split-primer) infrastructure. First, we explain the entire algorithm and then we introduce the three separate steps.

All the CIL instructions are implemented as asynchronous methods in a library attached to the produced code, so they can be used directly.

Algorithm 17 transforms an input HFG info a C# code containing a list of methods representing the control-flow constructs and a single *main* method, which is an entry point to the produced code. The algorithm uses procedures that transform loops, branches and graphs without control-flow, which are defined in the following sections along with the pattern graphs $LOOP$ and $BRANCH$. Besides the transformation procedures, we us additional helper procedures described in the following list:

- *copy* – creates a deep copy of a graph
- *match* – matches an associative graph rewriting rule (see Section 3.1.1)
- *replace* – replaces a subgraph by a single node with the given annotation

Algorithm 17 HFG transformation to asynchronous methods

Require: G – source HFG
$LOOP$ – loop pattern graph
$BRANCH$ – branch pattern graph
Ensure: C – C# code (set of methods)

```
1: C := ∅
2: H := copy(G)
3: X := 0
4: M := ∅
5: while Split ∈ N(H) do
6:   if match(LOOP, H, M) then
7:     C := C ∪ {LoopToMethod(G, M, "loop" + X)}
8:     H := replace(M, "loop" + X)
9:   else if match(BRANCH, H, M) then
10:    C := C ∪ {BranchToMethod(G, M, "branch" + X)}
11:    H := replace(M, "branch" + X)
12:  end if
13:  X := X + 1
14: end while
15: C := C ∪ {ToMethod(H, "main")}
```

9.1.2 Graph Without Control-Flow

A hybrid flow graph does not contain any control-flow constructs, when it contains no *split* or *merge* nodes. Without control-flow a HFG can be directly transformed into a C# method the same way we transform the subgraphs after the *component extraction* in the Bobox back-end.

We use the procedure $ToMethod$ to transform a HFG without control-flow to a single C# asynchronous method. The method is implemented using a modified version of Algorithm 16 defined in the previous chapter (Section 8.6). A slight modification is necessary for the algorithm to produce C# code instead of C++, we replace the variables automatic type *auto* by the C# equivalent *var*.

9.1.3 Branch Transformation

The branch pattern graph shown in Figure 9.1 matches the infrastructure including the conditional jump and the bodies of both branches, where we do not use the traditional notation for *split* and *merge* so we can add the plus sign meaning repetition in the associative graph rewriting systems. The bodies are HFG subgraphs, without any control flow nodes that can be transformed by the $ToMethod$ procedure. Iterative application of the algorithm ensures, that nested branches and loops are replaced from the inside of the nest out, starting with constructs containing no embedded control flow.

Algorithm 18 transforms the subgraph matched by the pattern graph in Figure 9.1 to a single asynchronous method, which is then added to the final code. The subgraph is then replaced by the method name.

Figure 9.2 shows the transformation of a simple branch subgraph matched by the patter graph. The transformation produces a single asynchronous method

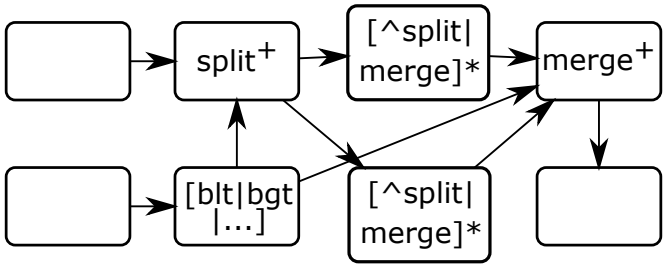

Figure 9.1: Branch pattern graph

Algorithm 18 BranchToMethod algorithm

Require: G – original HFG
M – matched HFG subgraph
N – method name
Ensure: C – C# code (method)

1: $IN := \{"p\{i.ID\}" : i \in N(M) \wedge \exists(n,i) \in E(G) \rightarrow n \notin N(M)\}$
2: $OUT := \{"p\{o.ID\}" : o \in N(M) \wedge \exists(o,n) \in E(G) \rightarrow n \notin N(M)\}$
3: $C := "async\ void\ \{N\}(\{join(',', IN)\}, out\ \{join(',out', OUT)\})\{"$
4: $C := C + "if(\{ToMethod(M.JUMP)\})\{"$
5: $C := C + ToMethod(M.BODY\_TRUE)$
6: $C := C + "\}else\{"$
7: $C := C + ToMethod(M.BODY\_FALSE)$
8: $C := C + "\}\}"$

that is called by the other methods produced from the HFG.

9.1.4 Loop Transformation

The loop transformation is similar to the branch, but the final method is recursive to accommodate the loop behavior. A loop can be transformed to either a C# loop or recursion, where the recursion allows the separate iterations to run in parallel, if they are independent, taking an advantage of the asynchronous behavior.

Pattern graph in Figure 9.3 matches an array infrastructure including its conditional jump and body. The loop body does not contain any other control flow constructs.

Algorithm 19 transform the matched loop subgraph to a recursive asyn-

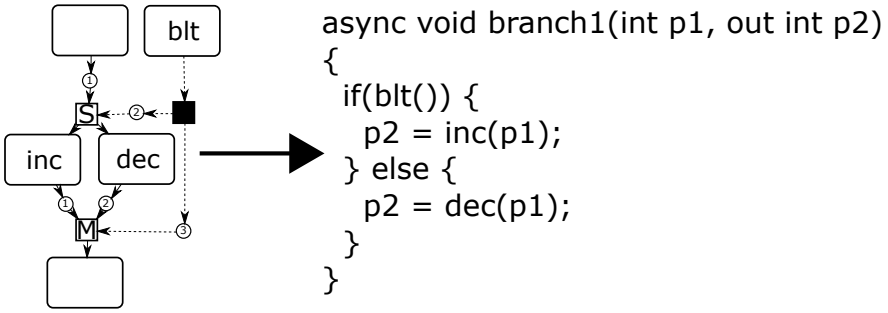

Figure 9.2: Branch transformation

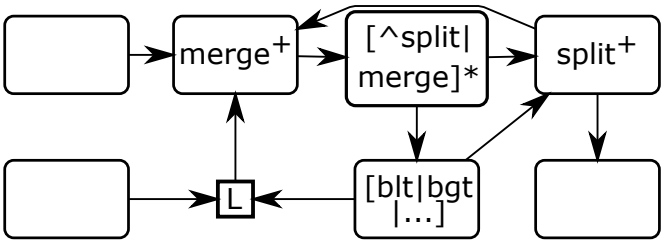

Figure 9.3:

Algorithm 19 LoopToMethod algorithm

Require: G – original HFG
M – matched HFG subgraph
N – method name
Ensure: C – C# code (method)

1: $IN := \{"p\{i.ID\}" : i \in N(M) \wedge \exists(n,i) \in E(G) \rightarrow n \notin N(M)\}$
2: $OUT := \{"p\{o.ID\}" : o \in N(M) \wedge \exists(o,n) \in E(G) \rightarrow n \notin N(M)\}$
3: $C := "async\ void\ \{N\}(\{join(',', IN)\}, out\ \{join(', out', OUT)\})\{"$
4: $C := C + ToMethod(M.BODY)$
5: $C := C + "if(\{ToMethod(M.JUMP)\})\{"$
6: $C := C + "\{N\}(\{join(',', IN)\}, out\ \{join(', out', OUT)\});"$
7: $C := C + "\}else\{"$
8: **for all** $I \in IN \wedge O \in OUT$ **do**
9: $\quad C := C + "\{I\} = \{O\};"$
10: **end for**
11: $C := C + "\}\}"$

chronous method. The algorithm is a bit simplified in regarding the parameters passing, where the inputs and outputs are matched, because every merge-split pair represents a single variable. The process is illustrated in Figure 9.4, where a simple loop is transformed according to the algorithm.

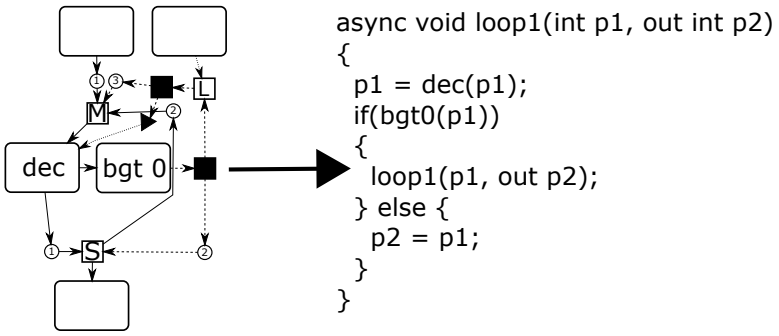

Figure 9.4: Loop transformation

9.2 Managed Bobox Integration

The last currently supported target platform is an integrated managed Bobox. In this setup, the compiler integrates the managed Bobox directly into the optimized

application, which is transformed into HFG and then to Bobox plan for the Bobox to execute. The process overview is in Figure 9.5. This platform was implemented as the original proof of concept and it is not optimized for efficiency.

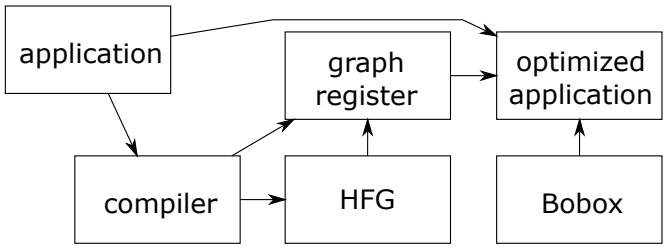

Figure 9.5: Application parallelization overview

The compiler transforms the application methods to HFGs and replaces their original code with a simple command that executes the produced plan by Bobox. The compiler also adds a *graph register*, which efficiently stores all the HFGs transformed to Bobox plans. Bobox is attached as a dynamic library to the application so it can be used to execute the plans.

The compiler creates an application that produces the same outputs as the original, but its methods are implemented in HFGs transformed in Bobox plans.

9.2.1 Graph Execution

Bobox computation is defined by an *execution plan* that is very similar to a hybrid flow graph, it is also an *annotated directed graph*. We have designed an adapter that transforms a HFG to a Bobox plan which is then executed by Bobox. The adapter is part of the *graph register* that stores all the HFGs and executes them in Bobox.

A HFG is basically identical to a Bobox execution plan, but it requires specific kernels to be executed, because Bobox must have access to the operations assigned to each node. We must provide a set of kernels for all CIL instructions and all control flow nodes. The kernels are already part of the Bobox library attached to the optimized application.

Once all the kernels are available, we simply convert each HFG to a plan by transforming each node to an appropriate kernel and connecting the kernels according to the edges.

The source HFG is transformed to a Bobox plan and stored in a graph register (see Section 9.2.2). This way we are able to execute the HFG efficiently and repeatedly, which is necessary for methods called in a loop or similar.

9.2.2 Graph Integration

We use the compiler front-end presented in Section 6 to transform the methods to HFGs. The Produced graphs are stored in the *graph register*, which is used by the rest of the application to execute them via Bobox. The resulting structure of the application is in Figure 9.6, in an UML-like notation.

The resulting parallel application contains some of its original code, parallelized methods, a graph register and a Bobox library. Which methods are parallelized is driven by the configuration described in Section 12.1. The register

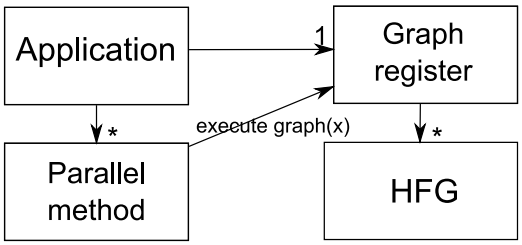

Figure 9.6: Structure of the parallelized application

contains a list of graphs, one for each parallelized method, and an instance of the Bobox environment. The compiler provides configuration tools that allow users to specify which methods are parallelized and it automatically integrates methods before parallelization.

The register is a singleton injected into the application by the compiler. Its code is in Listing 9.1. The graphs are stored in the register in the variable *graphRegister*, where they are inserted in the *Initialize* method. The graphs are initially stored as strings and in the *Initialize* method, they are transformed into a binary representation, optimized for efficient execution. The register initialization is performed only once when the application starts.

Listing 9.1: Graph register

```
public static class GraphRegister
{
  static Bobox.Scheduler scheduler;
  static List<Tuple<Box[], ModelInstance>> graphRegister;
  static bool initialized = false;
  static sbyte methodCounter;

  private static Scheduler Scheduler
  {
    get
    {
    if (scheduler == null)
    scheduler = new Bobox.SerialScheduler(1);
    return scheduler;
    }
  }

  private static void Initialize()
  {
  initialized = true;
  Logger.debug_out = false;
  graphRegister = new List<Tuple<Box[], ModelInstance>>();

  FlowGraph graph = FlowGraph.DeserializeFromString("");
  ModelInstance instance = new ModelInstance();
  Box[] _interface = flow_graph.BuildPlan(graph,instance);
  graphRegister.Add(_interface, instance);
```

```
  }

  public static void ExecuteGraph(int id)
  {
  if (!initialized)
    Initialize();
  Scheduler.Go(graphRegister[id].Item2);
  }

  public static Bobox.Box[] GetInputOutput(int id)
  {
  if (!initialized)
    Initialize();
  return graphRegister[id].Item1;
  }
}
```

The Bobox environment is also managed by the graph register. The register uses the C# implementation of Bobox. The environment is represented by the variable *scheduler* that contains the scheduler that manages the Bobox *execution plans*. The graph register provides the *ExecuteGraph* method which submits selected graph to the scheduler that executes it.

The parallelized methods are replaced by a short code that uses the register to execute the HFG constructed from their original code. The replacement code of the method is in Listing 9.2.

Listing 9.2: Method replacement

```
{
  int id = 0;
  Box[] param = register.GraphRegister.GetInputOutput(id);
  (param[2] as param<Int32>).Value = P;
  (param[1] as ldc_i4_repeat).Repeat = 1;
  register.GraphRegister.ExecuteGraph(id);
  return (param[0] as ret<Int32>).AcceptedValues.First();
}
```

The replaced method must do only three things - provide parameters for the graph (based on its actual parameters), execute the graph and obtain the returned value. In the code in Listing 9.2, the first four lines prepare parameters, calling the *ExecuteGraph* executes the graph using the parallel environment and the last line returns the value produced by the graph.

10. Case Study: Matrix-based Dynamic Programming

In the previous chapters, we presented the ParallaX compiler for streaming environments that transforms a C# application to an application for the Bobox streaming system. In this chapter, we will focus on the basic problem of *edit distance*, the *Levenshtein distance*, that will serve as a case study for the compiler. The Levenshtein distance calculates the number of changes necessary to make two strings equal, thus calculating their relative similarity.

The structure of the algorithm calculation, shown in Figure 10.1, severely limits any available parallelism. To improve this situation, the author of this work, as a member of a wider research team, has developed a special optimization that restructures the calculation to blocks that offer parallelism, suitable for vector instructions.

The author of this work designed the *blocked Levenshtein distance algorithm* along with two colleagues and the algorithm was initially published with GPU-specific optimizations [8] and then in a wider study that added Intel Xeon Phi and CPU optimization [2]. In the team, the author was responsible for the algorithm optimization for the GPUs. In the collaborative work, we described optimizations greatly improving the application of SIMD instructions and the main principles are described in the following sections.

The Levenshtein distance is just one problem of the matrix-based dynamic programming class. A similar problem that uses dynamic programming is *dynamic time warping* (DTW) defined by Müller [87]. Since the performance issue is also quite important for DTW applications, Sart et al. discussed parallelization techniques for GPUs and FPGAs [88]. They focused mainly on a specific version of DTW algorithm, which reduces the dependencies of the dynamic programming matrix, thus allows more efficient parallelization. Another example of a problem suitable for dynamic programming is the Smith-Waterman algorithm [89], which is used for protein sequences alignment. One of the first attempts to parallelize this algorithm on GPUs was made by Liu et al. [90].

We evaluate the ParallaX compiler correctness and efficiency on the blocked version of Levenshtein distance and compare the results to its serial C# implementation. The compiler must correctly transform the application to a streaming processing application and then add the necessary kernel and vector parallelism to improve its performance.

The rest of this chapter describes the blocked algorithm and its parallelization by our compiler. In Section 10.2 we explored other approaches to optimize the Levenshtein distance. Section 10.1 explains the distance algorithm and its properties. In Section 10.3 we present the details of the blocked version of Levenshtein distance. In Section 10.4 we adapt the blocked algorithm for streaming environments and in the next chapter, we present thw experiments with the applications produced by our compiler.

10.1 Problem Details

In the original research, we have selected the Wagner–Fischer dynamic programming algorithm [91] for the Levenshtein distance problem as a representative for our implementation since its computational simplicity emphasizes the communication and synchronization overhead associated with parallel computations. In this work, we use the blocked Levenshtein distance to evaluate the ability of the ParallaX compiler to exploit parallelism in a complex algorithm that provides opportunities for vectorization.

In the matrix-based dynamic programming algorithms, the values $v_{i,j}$ of the matrix are calculated recursively based on algorithm-specific function $f(i,j)$. The Functions $f(i,j)$ employed in these calculations are often very simple. In the case of the Wagner-Â–Fischer dynamic programming algorithm [91] for the Levenshtein distance, the function involves only comparison, incrementation, and minimum:

$$f(i,j) = \min(v_{i-1,j} + 1, v_{i-1,j-1} + 1 - \delta_{u[i]}^{v[j]}, v_{i,j-1} + 1),$$

where the Kronecker δ compares the i-th and j-th positions in the input strings u and v respectively.

The dependencies between individual invocations of the formula f significantly limit the parallelism available in the problem. For a $M \times N$ matrix (i.e., inputs of size M and N respectively), at most $min(M,N)$ elements may be computed in parallel using the diagonal approach illustrated in Figure 10.1. Therefore, when the computation of f does not take orders of magnitude more time than the data exchange, a pure diagonal approach cannot be effectively employed on current CPU and GPU architectures, because it would require global synchronization after every diagonal is processed.

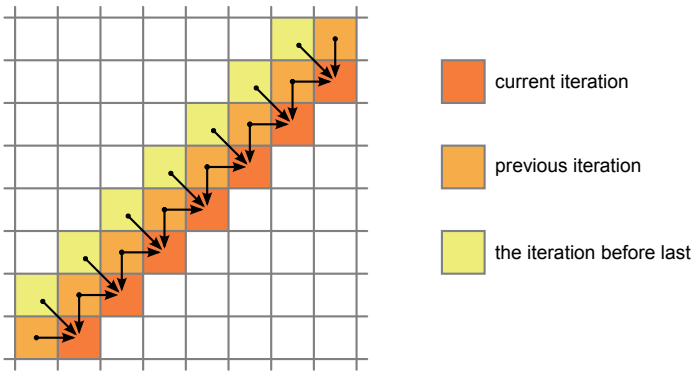

Figure 10.1: Dependencies in the matrix and the diagonals

On the other hand, the diagonal approach gives us basis for processing subsections of the distance matrix using SIMD instructions (in case of CPU and Intel Xeon Phi) or threads running in lockstep (in case of GPU). Although possible, the diagonal approach still produces insufficient parallelism and unnecessary memory transfers, because the diagonals must be constantly rotated through the SIMD registers. In Section 10.3 we present a special transformation that divides the algorithms matrix into block, which can be processed directly by SMID instructions without unnecessary memory operations.

10.2 Related Work – Matrix-based Dynamic Programming Parallelization

The algorithm and optimizations presented in this chapter are based on our previous work focusing on the parallelization of the matrix-based dynamic algorithms on GPUs [8] and out next research that further extended the optimizations to encompass Intel Xeon Phi [2]. Both our previous works focused on the vectorization capabilities of the respective accelerators. In this work, we transfer the optimizations to the environment of streaming systems using the ParallaX compiler described in the previous chapters.

Most parallel algorithms are based on an observation of Delgado et al. [92], who studied the data dependencies in the dynamic programming matrix. Two possible ways of processing the matrix were defined in their work – *uni-directional* and *bi-directional* filling. The original idea allows limited concurrent processing, but it needs to be modified for massively parallel environment.

One of the first papers that covers the whole issue of the parallelization of Levenshtein distance on GPUs was presented by Tomiyama et al. [93]. Their approach divides the dynamic programming matrix into parallelogram blocks. Independent parallelograms are computed by separate CUDA thread blocks while each block is computed in a highly cooperative manner. The main focus of the work addressed the problem of appropriate block size and its automatical selection. On the other hand, their experiments are currently out of date, since they were performed on a GPU with compute capability 1.3 only.

Manavski et al. [94] reimplemented the Smith-Waterman algorithm using CUDA technology. Slightly different solution was presented by Ligowski et al. [95]. Their work focused on searching in the entire database of proteins. Khajeh-Saeed et al. [96] utilized the computational power of multiple GPUs to solve this problem. Perhaps the most recent version was presented again by Liu et al. [97] and it combines observations from the previous work.

10.3 Levenshtein Distance Blocked Algorithm

As illustrated by Figure 10.2, the dependencies between elementary calculations in the two-dimensional matrix allow parallel computation for all elementary tasks on any diagonal line. An *elementary task* is consists of evaluation of the function $f(i, j)$ producing the value for a single element of the algorithm matrix. Thus, the computation may be done by a single sequential sweep through all the diagonals in the matrix whilst each diagonal is computed in parallel. Unfortunately, such simple approach to parallelization suffers from two deficiencies: First, the size of the diagonal (n) varies throughout the matrix which requires frequent addition and removal of computing units (threads) during the sweep. Second, regardless of the exact assignment of the physical threads to the elementary tasks, each thread processing a task must interchange information with at least one other thread when the computation advances to a subsequent diagonal. The computational cost of the elementary tasks is assumed to be small, thus the thread communication cost plays an important role as well.

When more than one instance of the problem is computed simultaneously, it is

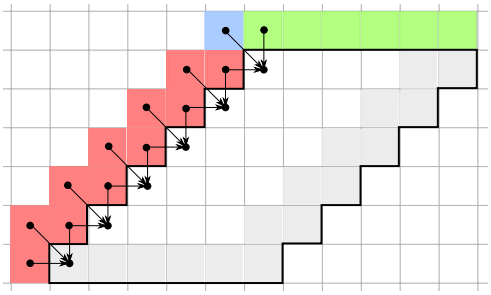

Figure 10.2: A single block with emphasized input, output, and data dependencies

also possible to compute the instances in parallel while each instance is evaluated sequentially (e.g., row-wise). However, this approach suffers from the inability to utilize the SIMD while efficiently using the caches.

Thus, although the multi-instance problem seems to be embarrassingly parallel, non-trivial algorithms must be investigated to overcome the memory and SIMD instruction constraints.

10.3.1 Parallelogram Blocks

Due to the nature of data-dependencies in the computation matrix, SIMD instructions work best when assigned to parallelogram-like portions of the matrix, as depicted in Figure 10.2. The picture also shows the input values for such a block, divided into *left buffer* (red), *upper-left buffer* (blue), and *upper buffer* (green). The arrows show the dependencies for the first diagonal in the block which will be computed in parallel. The output of the parallelogram block is shown in light gray, the white fields inside the parallelogram correspond to temporary values computed and later discarded during the block evaluation.

Each block is processed in three steps: load input data from the memory to SIMD registers, compute the values, and write the results back to the memory for the subsequent blocks. A block of height H and width W, with H equal to the SIMD register size (4 for SSE and 8 for AVX), is processed by SIMD instructions in W iterations. Between subsequent iterations, data is shuffled in the registers.

If H is bigger than the SIMD register size then the data must be processed in multiple SIMD streams, which must exchange data between registers.

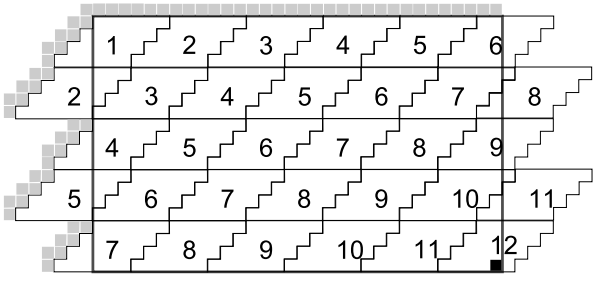

Figure 10.3: Distance matrix divided into parallelogram blocks

10.3.2 Blocked Algorithm

The elements of the distance matrix are grouped into parallelogram blocks as shown in Figure 10.3, where the numbers denote *coarse diagonals*. Blocks that are not completely contained in the distance matrix can be processed the same way as the others, which allows us to avoid checking any conditions during the block calculation, which comes at the cost of calculating some values that are not used. Gray area represents the input values for the left and upper blocks and the black square is the final result (computed edit distance). The blocks communicate through memory buffers, allocated in advance and aligned to 16 bits. The division of the matrix into the blocks follows a similar pattern as in the work of Tomiyama [93].

Similarly to the fine diagonals in Figure 10.2, the coarse diagonals allow parallel processing of blocks. Blocks retain the same scheme of dependencies as single elements, thus the blocks in a coarse diagonal can be processed in parallel, because they depend only on blocks from two previous coarse diagonals. The block numbers depicted in Figure 10.3 indicate the order in which they are computed.

Each coarse diagonal of blocks is computed by a group of identical kernels, where each corresponds to a single parallelogram block. Selecting appropriate block size (i.e., its width and height) is an important factor that affects the efficiency of the algorithm.

10.4 Matrix-based Dynamic Programming in Streaming Environments

In the previous section, we presented the blocked implementation of the Levenshtein distance algorithm originally designed for CPUs. In this section, we examine the way the ParallaX compiler transforms the C# implementation of the algorithm into a streaming application. The blocked algorithm can be adapted for other problems that share similar dependences and data structure.

However; we can adapt the optimizations, presented in Chapter 7, to vectorize the blocked implementation. This allows programmers to optimize their algorithm once it has been adapted to the blocked design.

The compiler is able to vectorize the code using the Intel CPU streaming extensions, SSE and AVX, but it can be later extended to other hardware, like GPUs, once the Bobox system supports it.

10.4.1 Blocked Algorithm Implementation in C#

We start with the blocked version of the Levenshtein distance algorithm implemented in C# and we will deconstruct it to explain the necessary optimizations. This is necessary, because the blocked implementation is complex and optimize is all at once not be practical.

Listing 10.1: Blocked Levenshtein edit distance algorithm

```
int BlockedLevenshteinDistance(int[] str1, int[] str2,
    int width, int height)
```

```
{
  #region Initialization

  var strWidth = str1.Length;
  var strHeight = str2.Length;
  var blockHeight = strHeight / height;
  var offsetWidth = blockHeight * (height - 1);
  var blockWidth = (strWidth + offsetWidth) / width + 1;
  var maxWidth = width * blockWidth;
  var appendixWidth = maxWidth - strWidth;

  int[][] prelast = new int[blockHeight][height + 1];
  int[][] last = new int[blockHeight][height + 1];
  int[][] current = new int[blockHeight][height + 1];
  int[][] horizontal = new int[blockWidth][width + 1];
  int[][] newhorizontal = new int[blockWidth][width + 1];

  Set(prelast, MaxValue);
  Set(last, MaxValue);
  last[0][1] = 1;
  Set(newhorizontal, MaxValue);

  for (int i = 0; i < blockWidth; i++)
  {
    for (int j = 0; j <= width; j++)
    {
      horizontal[i][j] = i * width + j;
    }
  }

  #endregion // Initialization

  for (int j = 0; j < blockHeight; j++)
  {
    for (int i = 0; i < blockWidth; i++)
    {
      var offset1 = offsetWidth + i*width - j*(height-1);
      var offset2 = j * height;
      var newHor = SolveBlock(width, height,
        ref prelast[j], ref last[j], ref current[j],
        horizontal[i], newhorizontal[i],
        str1, str2, offset1, offset2);

      res[0] = (i>0)? horizontal[i - 1][width]: MaxValue;
      var t = horizontal[i];
      horizontal[i] = res;
      newhorizontal[i] = t;
    }
```

```
  }

  var pos = offsetWidth + strWidth;
  return horizontal[pos / width][pos % width];
}
```

The blocked version of the Levenshtein distance algorithm is implemented in Listing 10.1. The implementation uses two additional functions, the function *Set* initializes all elements of an array to a specified value and the function *SolveBlock* calculates the results for a single parallelogram block. The *SolveBlock* function is implemented in Listing 10.2

Listing 10.2: Solve single parallelogram block

```
int[] SolveBlock(int width, int height,
  ref int[] prelast, ref int[] last, ref int[] current,
  int[] horizontal, int[] newHorizontal, int[] str1,
  int[] str2, int offset1, int offset2)
{
  prelast[0] = horizontal[0];
  int x = 1;
  do
  {
    int nX = x - 1;
    last[0] = horizontal[x];
    int y = 1;
    do
    {
      int nY = y - 1;
      int s1 = offset1 + nX - nY;
      int s2 = offset2 + nY;

      int char1 = str1[s1];
      int char2 = str2[s2];
      int left = last[nY];
      int upper = last[y];
      int upperleft = prelast[nY];

      // ParallaXMath provides functions similar to
      // C# Math, designed for the ParallaX compiler
      int dist1 = ParallaXMath.Min(left, upper) + 1;
      int dist2 = upperleft +
        ParallaXMath.NotEquals(char1, char2);
      current[y] = ParallaXMath.Min(dist1, dist2);
      y++;
    }
    while (y <= height);

    newHorizontal[x] = current[height];
```

```
    var temp = prelast;
    prelast = last;
    last = current;
    current = temp;
    x++;
  }
  while (x <= width);
  return newHorizontal;
}
```

10.4.2 Blocked Algorithm in ParallaX Compiler

The Levenshtein distance calculates the edit distance of two strings, by comparing every combination of characters in both. In this section, we call $strWidth$ the length of the first string and $strHeight$ the length of the second one. This basically means that the algorithm fills a matrix with $strWidth * strHeight$ elements and returns the one in the lower right corner.

The blocked algorithm is a nest of four loops, where the outer two loops iterate over bocks and the inner two loops solve a single block. The vectorization is centered in the innermost loop, which does most of the actual computations and is executed the most times. The body of the innermost loop is executed for every element of the virtual matrix, which means that it is executed $strWidth * strHeight$ times.

The outer loops offer no significant opportunity for vectorization (they just swap arrays) and they are executed significantly less times. The *SolveBlock* function is called approximately $(strHeight/height) * (strWidth/width)$, which is far less that the inner loop body, for reasonably big blocks.

Inner Loop Vectorization

Listing 10.3: Inner loop code

```
do
{
  // index computation
  int nY = y - 1;
  int s1 = offset1 + nX - nY;
  int s2 = offset2 + nY;
  // array loads
  int char1 = str1[s1];
  int char2 = str2[s2];
  int left = last[nY];
  int upper = last[y];
  int upperleft = prelast[nY];
  // distance computation
  int dist1 = ParallaXMath.Min(left, upper) + 1;
  int dist2 = upperleft +
```

```
    ParallaXMath.NotEquals(char1, char2);
  // array store
  current[y] = ParallaXMath.Min(dist1, dist2);
  y++;
}
while (y <= height);
```

We focus the optimization effort on the innermost loop, its code is in Listing 10.3 with its parts grouped and commented according to their function. The separation is important, because it will help us to vectorize the code.

First, it is important to note the algorithm uses the ParallaX library distributed along with the compiler, which provides basic mathematical and utility functions. Here, we use the functions Min, calculating minimum of two numbers, and $NotEquals$, which compares two numbers and converts the result to an integer (similar to $(int)(a! = b)$ in C++). The library functions are known to the compiler and they are directly transformed to a node or a subgraph (depending on the function).

Next, we can apply the component extraction optimizations presented in Chapter 7. We can use the *range extraction* without any modification, because the inner loop satisfies the pattern graph of the optimization.

The loop body can be divided into four separate parts, introduced by comments in the Listing 10.3. Schema of the loop structure is in Figure 10.4.

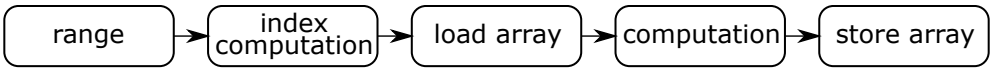

Figure 10.4: Blocked algorithm inner loop structure

The original implementation of the *array extraction* and *vectorization* optimizations rely on a loop, where we first load data from arrays, perform computation and then store the results, as shown in Figure 10.5. The inner loop structure is a bit more complicated, because it does not simply use the indices produced by the *range* operation, instead it calculates them from the range outputs.

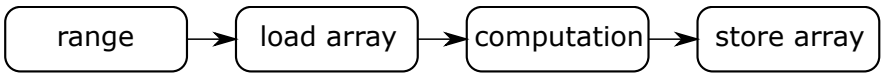

Figure 10.5: Loop structure expected by the basic optimizations

To optimize the inner loop, we use the *chained vectorization* and *chained array extraction* presented in Section 7.8 at the end of Chapter 7.

Once we apply the chained rules, we can use the original versions to vectorize the remaining parts of the inner loop. We use the standard *array extraction* to vectorize the *array store* part and we use the standard *vectorization* (see Section 7.7) to transform the *distance computation* part.

The vectorized inner loop is in Figure 10.6, including the subgraph representing both the custom operations *vect*1 and *vect*2. The vectorized graph contains incoming and outgoing edges that will be connected to the loop infrastructure of the outer loops, which will be the focus of the next section.

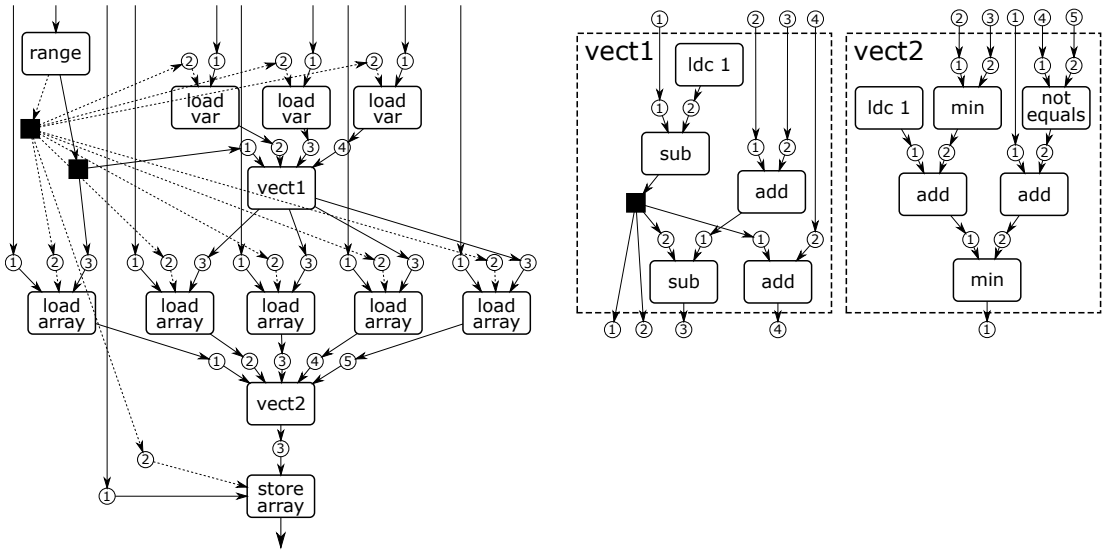

Figure 10.6: Fully vectorized inner loop of the blocked version of the Levenshtein distance algorithm

The HFG contains operation *load var*, which is an accumulator providing the repeatedly read variables $offset1$, $offset2$ and nX. The custom operation $vect1$ implements the index computation, which provides the indices used to access the arrays. The $vect2$ operation represents the actual distance computation, where we use the built-in functions *min* and *notEquals* to efficiently calculate the distance.

Outer Loop Vectorization

The outer loops of the entire algorithm contain almost no arithmetic operations and thus a very limited options for vectorization. We will inspect the loops one after another and focus on the optimizations we can apply there.

The outer loop of the *SolveBlock* function calculates the value of the nX variable and swaps the arrays used in the inner loop. This loop supports range extraction and we can extract the variable calculation as a tiny custom operation, but the rest of the loop must be preserved for the arrays to be swapped properly.

The inner loop of the blocked algorithm calculates the value of the two offsets and swaps the arrays used by the *SolveBlock*. This loop supports the range extraction and the offset calculation can be extracted as a custom operation, but the rest of the loop infrastructure (merge-split) must remain to properly swap the arrays.

The outermost loop contains no additional code and supports only the range extraction.

The vectorization of the outer loops is limited, but they are not the most important, because it is the inner loop that does most of the work. The inner loop is completely vectorized including all the array accesses and computations. In the next chapter, we will present experiments performed with the compiled blocked algorithm.

11. Experiments

We described the ParallaX compiler in the previous chapters and in this chapter, we will present the experiments we used to validate the correctness of the produced code and performance of the final applications. The experiments are divided in two groups: the *correctness experiments* validate the code produced by the ParallaX compiler against the original code and the *performance experiments* compare the performance of the produced applications with the C# original.

The actual implementation of the ParallaX compiler is a prototype that does not support the full range of optimizations necessary to efficiently parallelize all target applications. The goal of our work is to provide a proof of concept – a working compiler, with a framework for optimizations, able to produce a correct code. Because of this limitation, we test the performance of the produced code on a limited spectrum of applications, where the presented optimizations are sufficient to produce efficient applications.

11.1 Correctness Experiments

Before, we look at the performance, we must first verify that the applications produced by the ParallaX compiler are equivalent to the original C# code, they are based on. This is necessary to evaluate, because a compiler must primarily produce a correct code. We test the Bobox applications produced by the ParallaX complier. We use the managed implementation of Bobox, because that enables us to evaluate both the original and the product in the same environment, the Microsoft .NET.

In the case of the Bobox applications produced by the ParallaX compiler, a correct code is an application representing a C# method that can be executed in any implementation of Bobox and produces the same results as the original code. The results of an application are only its return values, there are no side effects, because we integrate all the called methods and we do not support libraries besides the one specially provided with our compiler.

We verify the correctness of the compiler on a huge set of generated methods and on well known algorithms including the factorial, greatest common divisor and Levenshtein distance. The process is same for all the experiments:

1. Input – the CIL code of the tested method
2. Compile the CIL code to a Bobox application using the ParallaX compiler
3. Execute the tested method using *.NET reflection*
4. Execute the Bobox application using the managed implementation of the Bobox system
5. Compare the returned values for both environments
   - Values equal – report success
   - Values differ – report failure

Listing 11.1: Generated method interface

```
public static int {0}(int p, int m)
{{
  int a = 0;
  {1}
  return a + m;
}}
```

The original method is executed using the *.NET reflection*, defined in the CLR standard [31], which allows extensive work with the data types and code structure during runtime. The classes implementing reflection allow the programmers to modify the application or execute dynamically loaded code and we use it to execute the original version of the tested method.

The Bobox application produced by the ParallaX compiler is executed in the managed (C#) implementation of Bobox to simplify the testing process. This way we can use solely the managed .NET code to verify the correctness of the produced code.

We test the compiler mostly on a set of methods, which we compose of the code fragments presented on the following listings. The code fragments are written in the C# string format notation, where the text N is replaced by the parameter N. For example, the call $Format(\text{"}\{0\} - is - \{1\}\text{"}, 0, \text{"}number\text{"})$ would result in the string $\text{"}0 - is - number\text{"}$.

The method interface is in Listing 11.1, it has two parameters, a single local variable and returns an integral number. We generate the methods by combining the other fragments nesting them in one another (including repetitions) until the depth three. Algorithm 20 generates all the methods, based on a set of all binary trees of depth three, and Algorithm 21 implements the $TreeToCode$ function, which generates the methods code based on a binary tree. The function $Format$ perform the replacement of the N parts with the parameters and the function $RandomName$ generates a random string that is valid as a C# method name.

Algorithm 20 GenerateMethods

Require: $TREES$ – set of all binary trees containing values (0-4)
$FRAGMENTS$ – sequence of all code fragments
$INTERFACE$ – method interface
Ensure: $METHODS$ – generated methods
1: $METHODS := \emptyset$
2: **for all** $t \in TREES$ **do**
3: $METHODS := METHODS \cup \{Format(INTERFACE, RandomName(), TreeToCode(t, 0, FRAGMENTS))\}$
4: **end for**

The following fragments are combined by the Algorithm 21 to the method code. We use the fragments presented in the following listing – a fragment for an *arithmetic operation*, *conditional branch*, *loop* and a *sequence*. Plus we add an

Algorithm 21 TreeToCode

Require: $TREE$ – binary tree containing values (0-4), represented in an array
N – index in the tree, initially 0
$FRAGMENTS$ – sequence of all code fragments
Ensure: $CODE$ – generated code
1: **if** $N < TREE.Length$ **then**
2: $CODE := Format(FRAGMENTS[N],$
$TreeToCode(TREE, 2 * N + 1, FRAGMENTS),$
$TreeToCode(TREE, 2 * N + 2, FRAGMENTS))$
3: **else**
4: $CODE :=$ ""
5: **end if**

Listing 11.2: Arithmetic operation

```
a *= p;
```

empty string representing empty code, which is used to terminate the replacement at the appropriate depth (three).

Using the presented algorithm and code fragments, we are able to generate a huge number of methods with all possible nesting of control flow constructs up to the depth of three. We generate nested loops, branches and any combination. The decremental behavior in loops is necessary to prevent endless iteration and assure that the code always ends and produces a valid result.

The correctness test is implemented in the ParallaX project automatic MS Unit test named – $CILParserMultitest$, distributed along with the project. The description of the project structure and instructions involving the automatic tests are in the chapter *Attachments: Digital Content*.

11.2 Levenshtein Distance Experiments

In this section, we evaluate the performance of the application produced by our compiler. We take the blocked implementation of the Wagner–Fischer dynamic programming algorithm [91], as explained in Section 10.3. We parallelize the blocked algorithm using the compiler and compare the performance of the resulting application with the original C# implementation, executed in .NET 4.6 with all optimizations. The blocked version of the algorithm enables the compiler to

Listing 11.3: Conditional branch

```
if(a % 2 == 0)
{{
  {0}
}}
```

Listing 11.4: Loop

```
do {{
  {0}
  a += p--;
}}
while (p > 0);
```

Listing 11.5: Construct sequence

```
{0}
{1}
```

optimize the block code and employ vectorization as described in Section 10.4.2.

The compiler produces an application for the native implementation of Bobox. We compare two versions, one is accelerated using the older SSE streaming extension and the other uses the AVX extension. We measure both versions to provide better insight into the vectorization impact on the overall performance of the application.

We measure the performance of both the versions on pairs of strings of the same size varying between 8 and 128 MB. We also measure the performance for different block sizes, because the block size influences the ratio between the kernel parallelism and vectorization. Bigger blocks provide more opportunities for vectorization, because the blocks inner loop is strongly optimized, where smaller blocks leave more room for parallel computation of separate blocks. We use blocks with height equal to the vector size (4 for SSE and 8 for AVX) and with is always specified in the experiment.

We measure the performance on two separate systems. First we a server running Windows 10 equipped with the Intel Xeon E5-4620v4 processor, with 10 physical cores and 20 virtual threads, and 16 GB of DDR4 random access memory. The second platform is a workstation running Windows 10 with Intel i7-6500U CPU, with 2 physical cores and virtual threads, and 8 GB of RAM. We use the high precision Performance counter, available in Windows, which is capable to measure the time in nanoseconds.

We measure only the actual computation run time, all data is loaded prior to the application start. In every experiment, we measure the application 10 times for both the original and the compiled version and we print the results in a single box plot to compare the performance including any special cases.

The experiment results are displayed as the time consumed to complete an elementary task, for Levenshtein distance, this means the computation of the value of a single element of the matrix. The results are acquired by dividing the total wall time by the size of the matrix. The results are summed in two-quartile boxplots with whiskers signifying the lowest value within $1.5 * IQR$ (interquartile range) of the lower quartile, and the highest value within $1.5 * IQR$ of the upper quartile. The plots vertical axes are in logarithmic scale.

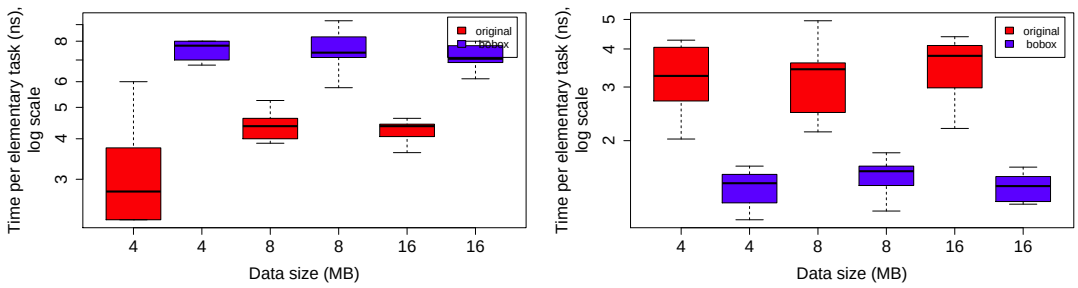

Figure 11.1: Levenshtein distance calculated with block size 16, workstation(left) server(right) , Bobox application is accelerated with SSE

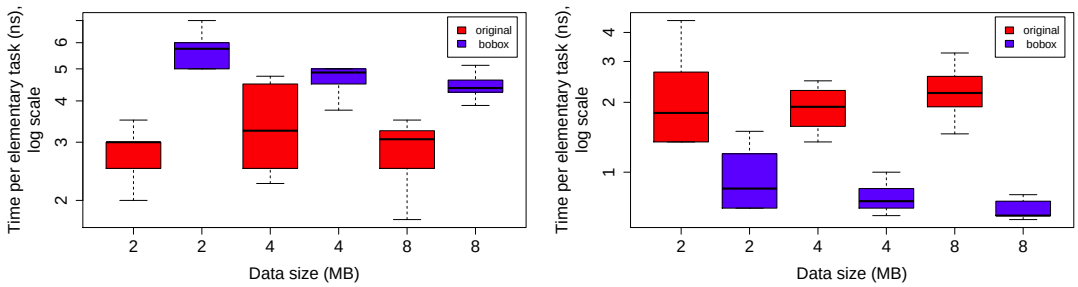

Figure 11.2: Levenshtein distance calculated with block size 128, workstation(left) server(right) , Bobox application is accelerated with SSE

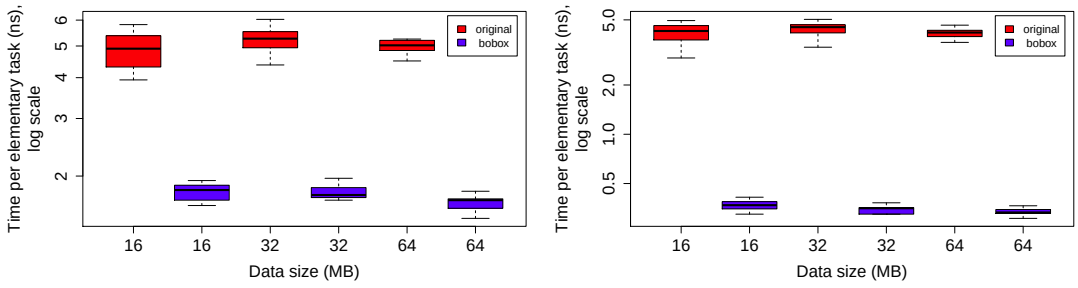

Figure 11.3: Levenshtein distance calculated with block size 1024, workstation(left) server(right) , Bobox application is accelerated with SSE

Figures 11.1, 11.2, 11.3, 11.4 contain the results for the Bobox application accelerated with SSE, where we measure the performance for the block sizes 16, 128, 1024 and 32768. The block size defines the width of the parallelogram blocks explained in the previous chapter, the block height is always equal to the vector size (4 for SSE and 8 for AVX).

Figures 11.5, 11.6, 11.7, 11.8 contain the results for the Bobox application accelerated with AVX. The results are better than the SSE acceleration, because AVX supports bigger vectors and provides better performance, especially for bigger blocks.

The compiler is able to apply all the optimization described in Section 7 and

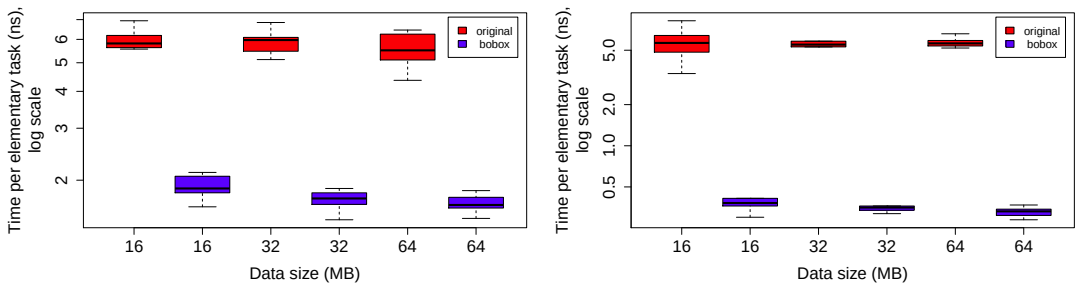

Figure 11.4: Levenshtein distance calculated with block size 32768, workstation(left) server(right) , Bobox application is accelerated with SSE

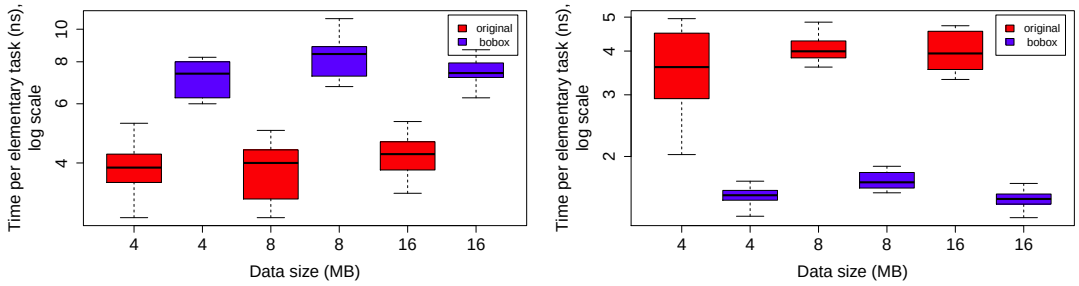

Figure 11.5: Levenshtein distance calculated with block size 16, workstation(left) server(right) , Bobox application is accelerated with AVX

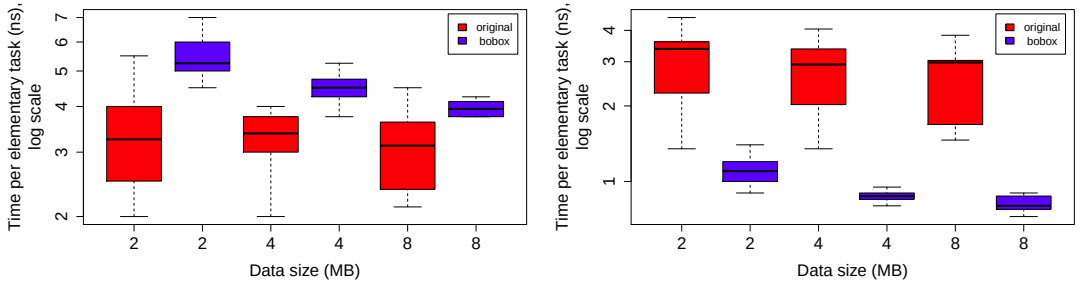

Figure 11.6: Levenshtein distance calculated with block size 128, workstation(left) server(right) , Bobox application is accelerated with AVX

the special optimization added in Section 10.4. The produced application was able to employ both vectorization and kernel parallelism thanks to the optimizations which significantly reduced the HFG complexity and granularity.

The performance is better for bigger blocks, which provide more opportunities for vectorization and reduced loop overhead. The Bobox application outperforms the original implementation in all experiments, with the exception of the smallest block size on the workstation, where the small number of cores and block size limit parallelization.

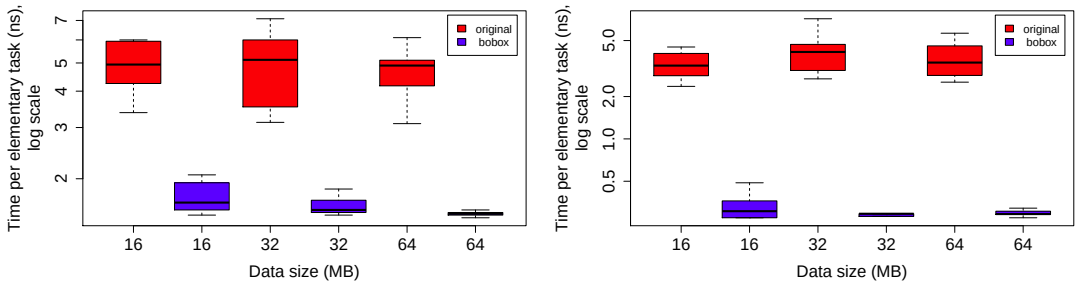

Figure 11.7: Levenshtein distance calculated with block size 1024, workstation(left) server(right) , Bobox application is accelerated with AVX

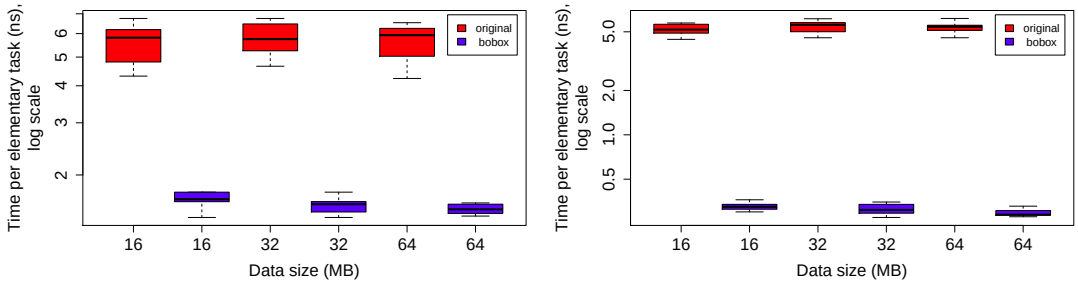

Figure 11.8: Levenshtein distance calculated with block size 32768, workstation(left) server(right) , Bobox application is accelerated with AVX

11.3 Streaming Experiments

In this section, we evaluate the ParallaX performance for a series of one dimensional algorithms. We present a convolution from the DSP category, with data dependences similar to the Levenshtein distance, but in one dimension. Next, we present an algorithm similar to the S-boxes used in the Serpent cipher, with multiple inputs and outputs. We conclude the experiments with a series of very simple algorithms, that apply a single function to a stream of value, to test the ParallaX compiler on very simple applications.

The experimental environment is the same as in case of the Levenshtein distance. We measure the performance on two separate systems. First, we use a server, running Windows 10, equipped with the Intel Xeon E5-4620v4 processor, with 10 physical cores and 20 virtual threads, and 16 GB of DDR4 random access memory. The second platform is a workstation running Windows 10 with Intel i7-6500U CPU, with 2 physical cores and virtual threads, and 8 GB of RAM. We use the high precision Performance counter, available in Windows, which is capable to measure the time in nanoseconds.

We measure only the actual computation run time, all data is loaded prior to the application start. In every experiment, we measure the application 10 times for both the original and the compiled version and we print the results in a single box plot to compare the performance including any special cases.

The experiment results are displayed as the time consumed to complete an elementary task, for Levenshtein distance, this means the computation of the

value of a single element of the matrix. The results are acquired by dividing the total wall time by the size of the matrix. The results are summed in two-quartile boxplots with whiskers signifying the lowest value within $1.5 * IQR$ (interquartile range) of the lower quartile, and the highest value within $1.5 * IQR$ of the upper quartile. The plots vertical axes are in logarithmic scale.

11.3.1 Convolution Experiment

The first experiment is a simple convolution algorithm, which represents simpler algorithms from the *digital signal processing* category. The algorithm constructs a new signal by combining the neighboring values of the input signal, using the function:

$$A[I] = 2 * B[I] - 0.5 * B[I-1] + 0.5 * B[I-2]$$

The function contains data dependences between the accesses to the array A, but the dependences are of the type read-read, which do no prevent parallelization. The application code is in Listing 11.6.

Listing 11.6: Convolution algorithm code

```
int[] Convolution(int[] a, int[] b)
{
  int i = 2;
  int len = a.Length;
  do
  {
    var prelast = b[i - 2];
    var last = b[i - 1];
    var current = b[i];
    a[i] = 2 * current - 0.5 * last + 0.5 * prelast;
    i++;
  } while (i < len);
  return a;
}
```

The experiment results shown in Figures 11.9 and 11.10. The ParallaX compiler successfully transformed, optimized and vectorized the code and the resulting application outperforms the original C# implementation on both platforms.

11.3.2 Cryptography Experiment

The second experiment is based on the linear transformation used in the Serpent Cipher to mix the outputs of the S-Boxes, the boxes combine the data with parts of the key and the transformation produces the final cypher [98]. The algorithm has four inputs and outputs and combines the values according to the following equations, where $\oplus$ denotes xor, <<< denotes left rotation and << denotes left shift:

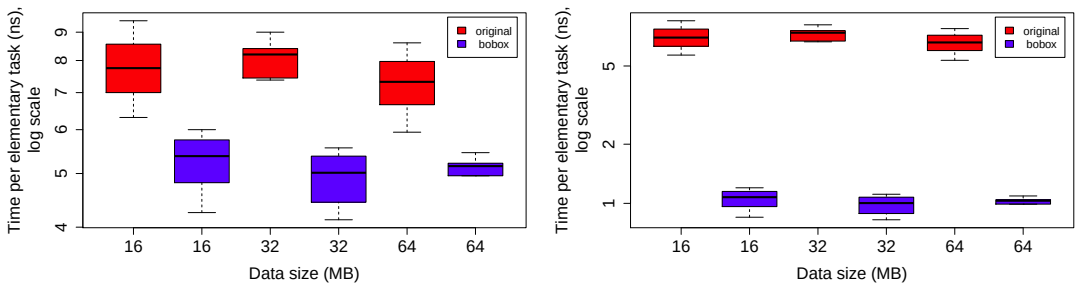

Figure 11.9: Convolution algorithm performance, workstation(left) server(right) , Bobox application is accelerated with SSE

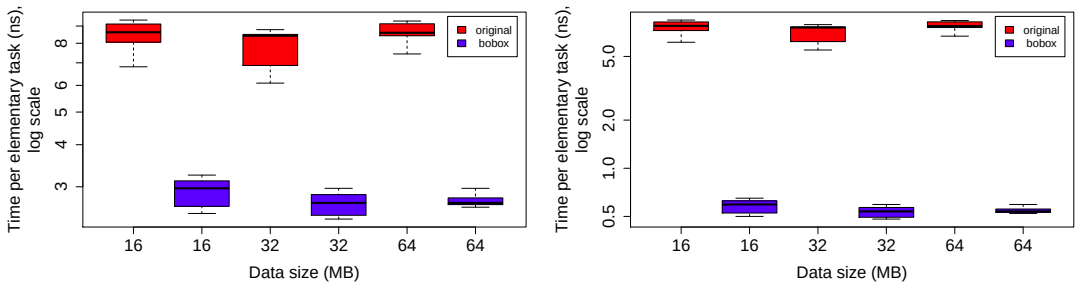

Figure 11.10: Convolution algorithm performance, workstation(left) server(right) , Bobox application is accelerated with AVX

$$
\begin{aligned}
b_0 &= b_0 <<< 13;\\
b_2 &= b_2 <<< 3;\\
b_1 &= b_1 \oplus b_0 \oplus b_2;\\
b_3 &= b_3 \oplus b_2 \oplus (b_0 << 3);\\
b_1 &= b_1 <<< 1;\\
b_3 &= b_3 <<< 7;\\
b_0 &= b_0 \oplus b_1 \oplus b_3;\\
b_2 &= b_2 \oplus b_3 \oplus (b_1 << 7);\\
b_0 &= b_0 <<< 5;\\
b_2 &= b_2 <<< 22;
\end{aligned}
$$

The C# implementation of the algorithms is in Listing 11.7. The original cipher was designed for parallelization and the ParallaX compiler is able to fully exploit the available parallelism. The experiment results are summed in Figures 11.11 and 11.12, the algorithm is more complex than the convolution experiment, because it contains more arithmetic operations that take more time to evaluate.

Listing 11.7: Convolution algorithm code

```
void Cipher(int[] a0, int[] a1, int[] a2, int[] a3,
  int[] b0, int[] b1, int[] b2, int[] b3)
{
  int i = 0;
  int len = a0.Length;
  do
  {
    var b_0 = b0[i];
    var b_1 = b1[i];
    var b_2 = b2[i];
    var b_3 = b3[i];
    b_0 = (b_0 << 13) | (b_0 >> -13);
    b_2 = (b_2 << 3) | (b_2 >> -3);
    b_1 = b_1 ^ b_0 ^ b_2;
    b_3 = b_3 ^ b_2 ^ (b_0 << 3);
    b_1 = (b_1 << 1) | (b_1 >> -1);
    b_3 = (b_3 << 7) | (b_3 >> -7);
    b_0 = b_0 ^ b_1 ^ b_3;
    b_2 = b_2 ^ b_3 ^ (b_1 << 7);
    b_0 = (b_0 << 5) | (b_0 >> -5);
    b_2 = (b_2 << 22) | (b_2 >> -22);
    a0[i] = b_0;
    a1[i] = b_1;
    a2[i] = b_2;
    a3[i] = b_3;
    i++;
  } while (i < len);
}
```

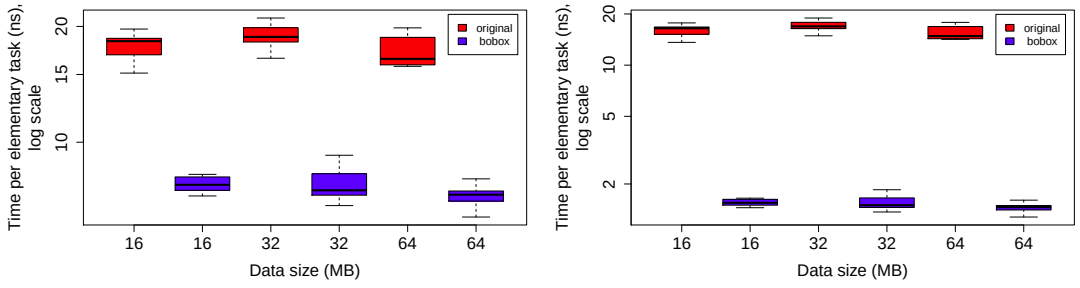

Figure 11.11: Serpent linear transformation algorithm performance, workstation(left) server(right) , Bobox application is accelerated with SSE

11.3.3 Baseline Experiments

To evaluate the performance of the applications produced by the ParallaX compiler for extremely simple applications, we constructed a series of filter-like applications with different structure to examine the application performance in different

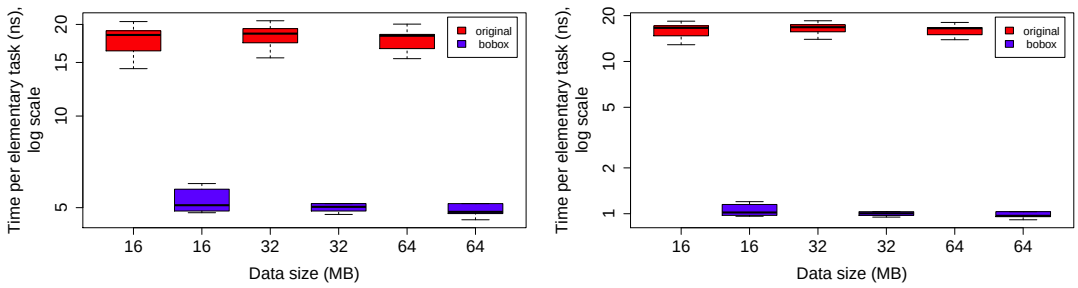

Figure 11.12: Serpent linear transformation algorithm performance, workstation(left) server(right) , Bobox application is accelerated with AVX

situations. We use the function *Filter* defined in Listing 11.8 to process the values and we use them in varying control flow structures to test the performance. The functions are purely artificial, designed to provide the required computational complexity and neither compiler uses the common sub-expressions to optimize the application (we verified it by studying both the HFG and the CIL produced by C# compiler).

Listing 11.8: Filter code

```
int Filter(int[] b, int i)
{
  int x = b[i] + i
    * i * i * i * i * i * i
    * i * i * i * i * i * i
    * i * i * i * i * i * i
    * i * i * i * i * i * i;
  return i * x;
}
```

11.3.4 Single Filter

The first application applies the *Filter* function to a series of values in a single loop, Listing 11.9 contains the application code. The performance of the application is shown in Figure 11.13.

Listing 11.9: Complex filter code

```
int[] SingleFilter(int[] a, int[] b)
{
  int i = 0;
  do
  {
    a[i] = Filter(b, i);
    i++;
  } while (i < a.Length);
```

```
  return a;
}
```

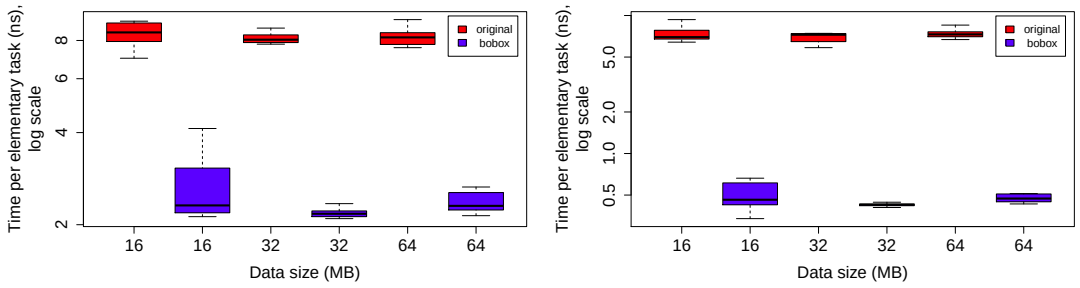

Figure 11.13: Performance of the *SingleFilter* application, workstation(left) server(right) , Bobox application is accelerated with AVX

11.3.5 Serial Filter

The second application applies two filters to a series of data in two loops, one after another. Listing 11.10 contains the application code, where the *Filter* function is either the *SimpleFilter* or *ComplexFilter*. The performance of the application is shown in Figure 11.14.

Listing 11.10: Complex filter code

```
int[] SerialFilter(int[] a, int[] b, int[] c)
{
  // filter 1
  int i = 0;
  do
  {
    a[i] = Filter(b, i);
    i++;
  } while (i < a.Length);

  // filter 2
  i = 0;
  do
  {
    c[i] = Filter(a, i);
    i++;
  } while (i < c.Length);

  return c;
}
```

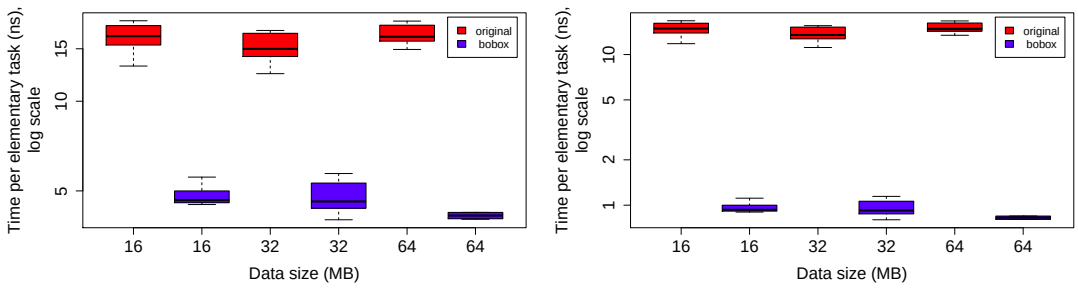

Figure 11.14: Performance of the *SerialFilter* application, workstation(left) server(right) , Bobox application is accelerated with AVX

11.3.6 Multiple Filter

The last application applies two filters in parallel and then it merges the result in three loops. Listing 11.11 contains the application code, where the *Filter* function is either the *SimpleFilter* or *ComplexFilter*. The performance of the application is shown in Figure 11.15.

Listing 11.11: Multi filter code

```
int[] MultiFilter(int[] a, int[] b, int[] c, int[] d)
{
  // filter 1
  int i = 0;
  do
  {
    b[i] = Filter(a, i);
    i++;
  } while (i < a.Length);

  // filter 2
  int i = 0;
  do
  {
    d[i] = Filter(c, i);
    i++;
  } while (i < c.Length);

  // merge
  int i = 0;
  do
  {
    a[i] = b[i] + d[i];
    i++;
  } while (i < a.Length);
  return a;
}
```

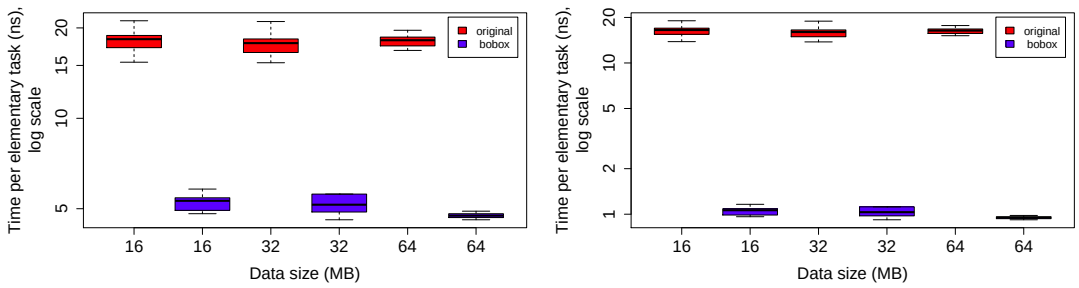

Figure 11.15: Performance of the *MultiFilter* application, workstation(left) server(right) , Bobox application is accelerated with AVX

11.3.7 Experiment Conclusions

The results show that the currently produced Bobox plans are efficient enough that the final Bobox applications outperform the original C# implementation. The results are better for applications that contain more arithmetic operations than control flow, especially for bigger data sets. This means that the more calculations outweigh control flow, the better is the Bobox doing in comparison to the original application.

The results are best for the more complex applications accelerated with the AVX extension. The produced Bobox applications achieve speedup roughly equal to the number of cores of the CPU. The achieved performance is not ideal, but the results suggest that the approach is viable. Additional optimizations can further improve the performance of the final applications.

12. Conclusions

We introduce the Hybrid Flow Graph as an alternative intermediate representation for procedural code. The main advantage of the HFG is its conceptual equivalence to streaming systems which, consequently, allows the use of streaming systems as run-time environments for such programs. HFG can be used across multiple systems, because it does not contain any domain-specific concepts.

We successfully implemented a prototype compiler that allows programmers to design simple streaming applications in a restricted version of the C# language and transform the produced code into HFG which is then executed in the Bobox streaming system.

The compiler supports code written in a restricted version of the C# language, where we limit the use of concepts that significantly modify control flow, like exception management. We experimentally verified the correctness of the produced applications on a huge collection of generated methods, which we transformed, executed and compared the results with the original.

Although the HFG allows direct execution by a streaming system, it requires advanced optimizations to produce efficient applications. As our measurements presented in Section 11 show, these steps reduce the synchronization overhead so significantly that the performance bonuses, associated with vectorized execution available in streaming systems, become clearly visible. The produced applications do not use the system resources to the maximal extent, but additional optimizations should further improve the performance.

In our experiment setup, we were able to produce plans for the Bobox streaming environment that outperform the original C# code. These result suggest that transforming a well-known procedural language to streaming plans is a viable alternative to dedicated non-procedural streaming languages. This approach allows the use of streaming systems by programmers who are not experts in C++ and/or parallel programming.

The steps used to optimize the code used for our experiments (Section 11) are only a part of the effort required to encompass the variability of programming patterns which must be supported by any parallelizing or vectorizing technique. Thus, our current and future work will focus on further optimizing steps – several traditional compiler techniques such as loop skewing may be transferred to the environment of HFG; however, there are also optimization techniques native to the streaming environments that we would like to introduce to our system.

The compiler is currently able to handle all aliases in the code, because we restrict reference data types in the input code, with the exception of arrays. Arrays can cause aliasing between subscripts, but we handle the arrays as compact objects, which prevents aliasing at the expense of efficiency. This behavior is changed by optimizations that break arrays into streams and access the elements one by one, but the optimizations are designed to be applicable only when there is no risk of aliasing.

12.1 Future Work

In our future work, we will focus on widening the scope of applications the ParallaX compiler is able to transform and optimize efficiently. We will remove some of the restrictions currently imposed on the supported source code and we will further improve the code generation process to allow the compiler to produce building blocks of the Bobox applications to encompass the use cases presented in the motivation.

We currently use method integration (inlining) to process method calls, but it is also possible to use the layered HFG to perform inter-procedural analysis. We can introduce more layers to represent the called methods and their components, thus creating basically a graph of Hybrid Flow Graphs. This way, the compiler will be able to handle a wider scope of applications, including recursion.

The structure of streaming applications does not allow for a full object model supported by C#. We will add at least *structs*, because they do not have reference semantics and thus do not introduce complex aliasing. A full object support is not necessary, because the compiler was not designed to optimize general C# applications, but as a tool to implement algorithmic parts of bigger streaming applications, such as database-like queries. These applications require at least arrays, but the introduction of structures could simplify their design.

We can treat the structures as arrays and always transfer and access the entire structure at once, or we might be able to break the structures into separate variables and handle them more efficiently. The second approach means that we treat $a.x$, $a.y$, $b.x$ and $b.y$ as separate variables for $Vector2\ a, b$. This approach is more efficient, but it requires more complex analysis to prevent aliasing, especially when the method integration introduces multiple structure variables.

Our current experiments focus mainly on data intensive applications as our target use cases, which are mostly streaming processing friendly. Other applications are also supported by the compiler, but their performance is inferior, because they would require other special optimizations which are not currently implemented. In the future we will focus on additional optimizations necessary to produce efficient code for a wider selection of application.

In this work, we present a compiler that allows programmers to design streaming applications with the C# programming language. Our work was motivated by the intention to design algorithmic parts of bigger streaming applications in a general programming language and integrate them directly into the streaming application. To achieve this goal, the compiler must be able to produce parts of streaming applications and integrate them with existing plans. This process is not yet implemented, but it requires only a minor modification of the inputs and outputs, which must be properly connected to the wider plan and we will implement the changes in our future work.

Appendix A: Compiler Infrastructure

The ParallaX compiler for streaming environments presented in this work is a complex piece of software, which requires configuration to provide access to all its features. This sections provides information about the configuration, general work-flow and examples of an application structure that takes advantage of the compiler features.

The compiler internally uses a newly developed intermediate language called *Hybrid flow graph* or *HFG* to represent the parallelized code. Each processed application is transformed to HFG, the resulting code can be used to parallelize applications or it can be exported directly and executed by external parallel environment, like *Bobox*. This allows us to process an application, produce a set of HFG algorithms that we can combine directly in a streaming environment. The compiler can be used as a cross-compiler for parallel environments that require input in a form of computation plan represented by a directed graph, like *Bobox*.

12.2 Compiler Work Flow

The compiler is presented in detail in Chapters 5, 6, 7 and 8. It provides a unified interface, based on C# code, and it offers multiple target platforms, focusing mostly on the Bobox system. The work-flow is strongly influenced by the target platforms, because they introduce additional steps and requirements.

The main target platform is the native implementation of the Bobox streaming environment. The overview of its work flow is in Figure 12.1, where the compiler produces execution plan and kernels for Bobox and the user must integrate the components with the environment. In the case of a managed Bobox, this can be done dynamically by attaching Bobox as a library, but the native (C++) implementation must be compiled along with the custom kernels to achieve optimal performance. This allows the C++ compiler to optimize the kernels along with the rest of the code.

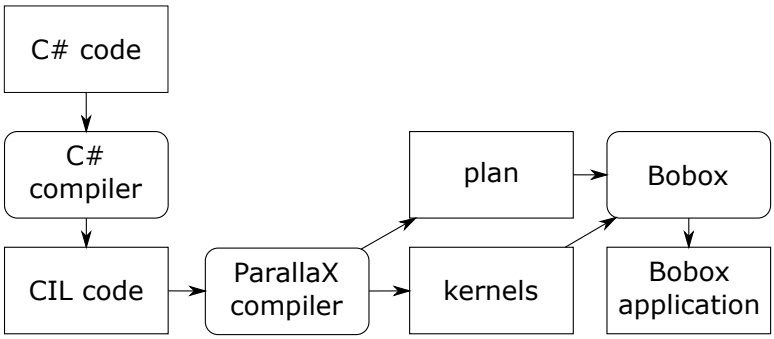

Figure 12.1: ParallaX compiler architecture

Our compiler also provides back-end for direct parallelization of C# applications, either by integrating the managed implementation of Bobox or by transforming the application code into asynchronous methods. The work-flow for both

platforms is very similar and requires much less additional work on the side of the programmer. The compiler simply produces a new .NET application that with different code but the same behavior. This work-flow is summed in Figure 12.2, where the Bobox runtime and execution plans are necessary only for the Bobox integration method.

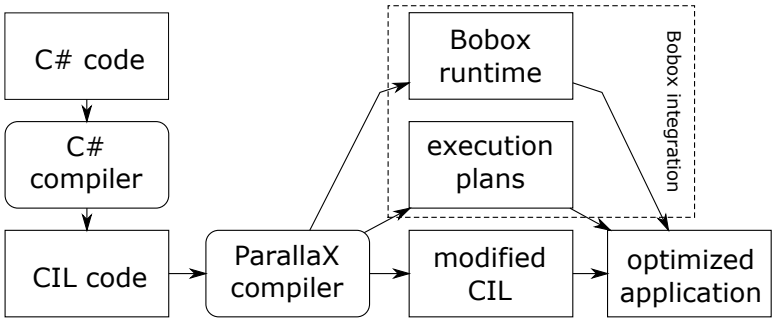

Figure 12.2: Compiler work flow overview

Both the different work-flows require that the input application is implemented in C#, compiled into CIL and then fed to our compiler. This extra step can be automatized via scripts or pre-build code.

12.3 Application structure

The compiler provides the user with a complete control over the application processing and parallelization, because parallelism is beneficial only for certain algorithms. The optimization process is controlled by two factors - the application structure and the compiler configuration. The configuration is explained in next section.

An application, that is to be parallelized, can be annotated by special attributes provided as part of a tiny library distributed with the compiler. These attributes do not modify the application behavior in any way, they are just used by the compiler to locate parts of code that should be parallelized or transformed to plan and kernels.

The code in Listing 12.1 shows a simple application that adds numbers in a loop. The *Add* method has an attribute called $Transform$ that tells the compiler what is to be done with the function. The attribute has a parameter that specifies the operation performed by the compiler. Each operation is identified by a string and it is applied to all methods or classes that have the Transform attribute with that exact string as parameter.

The actual operations are defined the compiler configuration described in next section. The configuration defines the operation identifiers which are then compared to the parameters of the $Transform$ attributes used in the application.

The *Add* method is very simple and it would have been integrated by the .NET JIT compiler, but by transforming it to a HFG, we can make it run in parallel for each iteration of the loop. This does not improves performance as the method is too simple, but it serves as an example.

Listing 12.1: Annotated application

```
class Program
{
  [Transform("CreateBobolang")]
  static int Add(int A, int B)
  {
        return A + B;
  }

  static void Main(string[] args)
  {
        for(int i = 0; i < 1000; i++)
        {
          int res += Add(i, 100);
    }
    Console.WriteLine(res);
  }
}
```

12.4 Compiler Configuration

The compiler work is controlled by a XML configuration file. The configuration contains a list of transformation files, where each transformation file represents a set of transformations applied to a single assembly and its result is exported as a new assembly. The transformation files are applied in series, which allows for recursive optimizations.

A transformation file contains a list of profiles, where each profile represents a single transformation applied to all methods with the attribute $Transform$, where the profiles name matches the attributes parameter $profile\_name$. Figure 12.3 contains a schematic view on the application configuration in an UML-like notation.

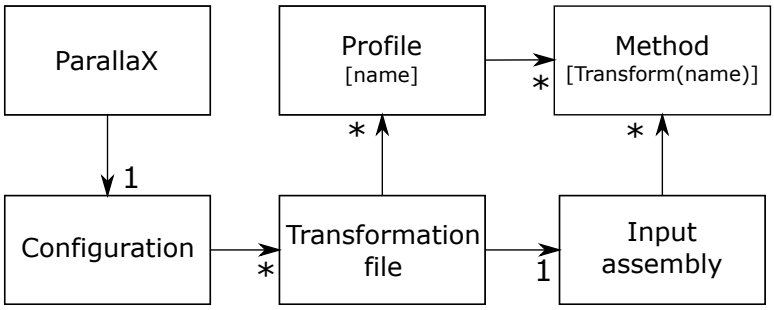

Figure 12.3: Configuration file structure

12.4.1 Configuration File

The configuration also contains parameter *OptimizationLevelLogging* that defines how much information is logged during the optimization. Available values

Listing 12.2: Compiler configuration file

```
<?xml version="1.0"?>
<GlobalConfiguration>
  <TransfileNames>
    <string>transfile.xml</string>
  </TransfileNames>
  <OptimizeationLoggingLevel>None
  </OptimizeationLoggingLevel>
</GlobalConfiguration>
```

are *None*, *Info* and *Debug*, where None prints no logs (only fatal errors) and Debug outputs everything. Listing 12.2 shows a sample configuration file.

12.4.2 Transformation File

Transformation files have two parameters $TargetAssembly$ and $OutputAssembly$, which identify input and output assemblies. The transformation files can be chained to incrementally modify an assembly, because every transformation file produces a new version that can be loaded by the following transformation file.

All transformation files and profiles are processed in the order they are defined. The profiles always apply only to the methods with the attribute $Transform$ with the profiles name as parameter. Each transformation file loads a single assembly, processes all the profiles and produces a new assembly.

Listing 12.3 shows a simple transformation file, which contains a profile named $CreateBobolang$. The C# source code shown in Listing 12.1 would be affected by this configuration, because the *Add* method has an attribute $Transform$ with a parameter "$CreateBobolang$". This particular transformation file contains a single profile that represents the $CreateBobolang$ optimization, which produces a Bobox execution plan from every method with the proper $Transform$ attribute.

Listing 12.3: Compiler transformation file

```
<?xml version="1.0"?>
<Transfile>
  <TargetAssembly>Example.exe<TargetAssembly>
  <OutputAssembly>Example_out.exe<OutputAssembly>
  <Profiles>
      <Profile>
        <Name>CreateBobolang</Name>
        <ProfileOptimization>
          CreateBobolang
        </ProfileOptimization>
        <OptimizationParameters>
          <string>bobolang1.txt</string>
        </OptimizationParameters>
      </Profile>
  </Profiles>
</Transfile>
```

Appendix A: Compiler Library

This chapter is focuses on the library distributed along with the ParallaX compiler. The library, called $ParallaXMath$, is part of the ParallaX interface that provides the attributes used to control the compiler process, see Appendix A for details.

The library is designed as an analogy of the $Math$ library distributed as part of the .NET framework, we include most of the same functions, like minimum (min) or absolute value (abs). The library provides fully functioning procedures that can be tested with the C# source code.

The library functions are recognized by the ParallaX compiler, which does not integrate them. Instead, the compiler transforms each call to one of the functions directly into an optimized custom node that implements its functionality. This gives programmers the basic arithmetic functionality and also allows them to produce more efficient code, because the compiler can use specifically optimized nodes, instead of using general optimizations.

The library currently provides only a limited selection of functions, because it is a prototype designed to support the tests and experiments presented in this work. The final release will contain more functions and algorithms to provide a good foundation for the applications designed for the compiler. The library currently provides the following functions:

- Min – minimum of two numbers.
- Max – maximum of two numbers.
- Abs – absolute value of a number.
- $Sign$ – sign of a signed integral number.
- $Equals$ – compares two values for equality and converts the result to integer.
- $NotEquals$ – negation of $Equals$.

Bibliography

[1] Michal Brabec and David Bednárek. Programming parallel pipelines using non-parallel C# code. *CEUR Workshop Proceedings*, 1003:82–87, 2013.

[2] Zbyněk Falt, Martin Kruliš, David Bednárek, Jakub Yaghob, and Filip Zavoral. Locality aware task scheduling in parallel data stream processing. In David Camacho, Lars Braubach, Salvatore Venticinque, and Costin Badica, editors, *Intelligent Distributed Computing VIII*, volume 570 of *Studies in Computational Intelligence*, pages 331–342. Springer International Publishing, 2015.

[3] Zbyněk Falt, David Bednárek, Martin Kruliš, Jakub Yaghob, and Filip Zavoral. Bobolang: A language for parallel streaming applications. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 311–314. ACM, 2014.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[5] Frédéric Boussinot and Robert De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[6] Xin Li and Reinhard Von Hanxleden. Multithreaded reactive programming – the Kiel Esterel processor. *Computers, IEEE Transactions on*, 61(3):337–349, 2012.

[7] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[8] A. Hagiescu, W. F. Wong, D. F. Bacon, and R. Rabbah. A computing origami: Folding streams in FPGAs. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 282–287, July 2009.

[9] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[10] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[11] Dhruba Borthakur, Jonathan Gray, Kannan Sarma, Joydeep Sen Spiegelberg, Nicolas Muthukkaruppan, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes

realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[12] Hakan Kocakulak and Tugba Taskaya Temizel. A hadoop solution for ballistic image analysis and recognition. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 836–842. IEEE, 2011.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[14] Milind Bhandarkar. Mapreduce programming with apache hadoop. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–1. IEEE, 2010.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[17] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[18] Zbyněk Falt, David Bednárek, Martin Kruliš, Jakub Yaghob, and Filip Zavoral. Bobolang: A language for parallel streaming applications. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 311–314, New York, NY, USA, 2014. ACM.

[19] Martin Kruliš, David Bednárek, Zbyněk Falt, Jakub Yaghob, and Filip Zavoral. Towards semi-automated parallelization of data stream processing. In *Intelligent Distributed Computing IX*, pages 235–245. Springer, 2016.

[20] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-Jörg Kreowski. *Handbook of graph grammars and computing by graph transformation*, volume 3. world Scientific, 1999.

[21] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[22] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180. IEEE, 1973.

[23] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, 1993.

[24] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation: part ii: single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars*, pages 247–312, 1997.

[25] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. 1976.

[26] Stephen A Edwards. Kahn process networks. *Languages for Digital Embedded Systems*, pages 189–195, 2000.

[27] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: the compaan/laura approach. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10340. IEEE Computer Society, 2004.

[28] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[29] Željko Vrba, Pål Halvorsen, Carsten Griwodz, and Paul Beskow. Kahn process networks are a flexible alternative to mapreduce. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 154–162. IEEE, 2009.

[30] European Computer Manufacturers Association et al. *Standard ECMA-334: C# Language Specification*. ECMA, 2006.

[31] ECMA ECMA. 335: Common language infrastructure (cli). *ECMA, Geneva (CH),*, 2005.

[32] David S Platt. *Introducing Microsoft. Net*. Microsoft press, 2002.

[33] Miguel de Icaza, P MOLARO, R PRATAP, D PORTER, et al. The mono project. *Available from www. mono-project. com*, 2004.

[34] Michal Brabec, David Bednárek, and Petr Malỳ. Transformation of pipeline stage algorithms to event-driven code. In *ITAT*, pages 13–20, 2014.

[35] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann Publishers, 1997.

[36] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann San Francisco, 2002.

[37] Michal Brabec and David Bednárek. Transforming procedural code for streaming environments. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 167–175. IEEE, 2017.

[38] Michal Brabec and David Bednárek. Procedural code representation in a flow graph. In *DATESO*, pages 89–100, 2015.

[39] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[40] KAHN Gilles. The semantics of a simple language for parallel programming. In *Information Processing: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[41] Mark B Josephs. Models for data-flow sequential processes. In *Communicating Sequential Processes. The First 25 Years*, pages 85–97. Springer, 2005.

[42] Joaquin Ezpeleta, Jose Manuel Colom, and Javier Martinez. A Petri net based deadlock prevention policy for flexible manufacturing systems. *Robotics and Automation, IEEE Transactions on*, 11(2):173–184, 1995.

[43] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *Programming languages and systems*, pages 319–334. Springer, 2003.

[44] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 281–290. IEEE, 2009.

[45] Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, and Filip Zavoral. Parallel SPARQL query processing using Bobox. *International Journal On Advances in Intelligent Systems*, 5(3 and 4):302–314, 2012.

[46] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[47] Leif Geiger and Albert Zündorf. Graph based debugging with fujaba. *Electr. Notes Theor. Comput. Sci.*, 72(2):112, 2002.

[48] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2007.

[49] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Graph transformation systems. *Fundamentals of Algebraic Graph Transformation*, pages 37–71, 2006.

[50] Andrea Corradini, Fernando Luís Dotti, Luciana Foss, and Leila Ribeiro. Translating java code to graph transformation systems. In *Graph Transformations*, pages 383–398. Springer, 2004.

[51] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 9–14. ACM, 2007.

[52] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[53] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[54] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[55] Michal Brabec, David Bednárek, and Petr Malý. Transformation of pipeline stage algorithms to event-driven code. In Vera Kurkova, Lukás Bajer, and Vojtech Svátek, editors, *Proceedings of the 14th Conference on Information Technologies - Applications and Theory, Jasna, Slovakia, 2014.*, volume 1214 of *CEUR Workshop Proceedings*, pages 13–20. CEUR-WS.org, 2014.

[56] Michal Brabec and David Bednárek. Programming parallel pipelines using non-parallel c# code. In *ITAT*, pages 82–87, 2013.

[57] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[58] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[59] S. A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design Test of Computers*, 23(5):375–386, May 2006.

[60] S. J. Allan and A. E. Oldehoeft. A flow analysis procedure for the translation of high-level languages to a data flow language. *IEEE Transactions on Computers*, C-29(9):826–831, Sept 1980.

[61] P. A. Arras, D. Fuin, E. Jeannot, and S. Thibault. DKPN: A composite dataflow/Kahn process networks execution model. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 27–34, Feb 2016.

[62] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. Hybrid dataflow/von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1489–1509, June 2014.

[63] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. Data stream processing via code annotations. *The Journal of Supercomputing*, pages 1–15, 2016.

[64] M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. Introducing parallelism by using repara c++11 attributes. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 354–358, Feb 2016.

[65] Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. Vectorization for Java. In *Network and Parallel Computing*, pages 3–17. Springer, 2010.

[66] Curt Albert, Alastair Murray, and Binoy Ravindran. Applying source level auto-vectorization to Aparapi Java. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 122–132. ACM, 2014.

[67] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Parallel execution of Java loops on graphics processing units. *Science of Computer Programming*, 78(5):458–480, 2013.

[68] Rafael Duarte, Alexandre Mota, and Augusto Sampaio. Introducing concurrency in sequential Java via laws. *Information Processing Letters*, 111(3):129–134, 2011.

[69] Walter Cazzola and Edoardo Vacchi. @ java: Bringing a richer annotation model to Java. *Computer Languages, Systems & Structures*, 40(1):2–18, 2014.

[70] Cristian Dittamo, Antonio Cisternino, and Marco Danelutto. Parallelization of C# programs through annotations. In *Computational Science–ICCS 2007*, pages 585–592. Springer, 2007.

[71] Soumya S Chatterjee and R Gururaj. Lazy-parallel function calls for automatic parallelization. In *Computational Intelligence and Information Technology*, pages 811–816. Springer, 2011.

[72] Yeoul Na, Seon Wook Kim, and Youngsun Han. Javascript parallelizing compiler for exploiting parallelism from data-parallel html5 applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):64, 2016.

[73] Martin Kruliš. Is there a free lunch for image feature extraction in web applications. In *International Conference on Similarity Search and Applications*, pages 323–331. Springer, 2015.

[74] Jakub Misek, Benjamin Fistein, and Filip Zavoral. Inferring common language infrastructure metadata for an ambiguous dynamic language type. In *Open Systems (ICOS), 2016 IEEE Conference on*, pages 111–116. IEEE, 2016.

[75] Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Variable-precision reaching definitions analysis. *Journal of Software Maintenance: Research and Practice*, 11(2):117–142, 1999.

[76] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 32(3):137–142, 1989.

[77] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, pages 209–265. Springer, 2002.

[78] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[79] Jakub Misek and Filip Zavoral. Control flow ambiguous-type inter-procedural semantic analysis for dynamic language compilation. *Procedia Computer Science*, 109:955–962, 2017.

[80] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. *ACM SIGPLAN Notices*, 38(5):103–114, 2003.

[81] Jb Evain. Mono cecil. *a vailable at: http://www. mono-p roject. com/Cecil, accessed May*, 2007.

[82] Henry G Baker. Garbage collection, tail recursion and first-class continuations in stack-oriented languages, December 31 1996. US Patent 5,590,332.

[83] David Bednárek, Michal Brabec, and Martin Kruliš. Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64:175–193, 2017.

[84] Hongwei Xi. Dead code elimination through dependent types. In *PADL*, volume 99, pages 228–242. Springer, 1999.

[85] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[86] Martin Kruliš, David Bednárek, and Michal Brabec. Improving parallel processing of matrix-based similarity measures on modern gpus. In *International Conference on Similarity Search and Applications*, pages 283–294. Springer, 2015.

[87] Meinard Müller. Dynamic time warping. *Information retrieval for music and motion*, pages 69–84, 2007.

[88] Doruk Sart, Abdullah Mueen, Walid Najjar, Eamonn Keogh, and Vit Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 1001–1006. IEEE, 2010.

[89] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[90] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. Gpu accelerated smith-waterman. In *Computational Science–ICCS 2006*, pages 188–195. Springer, 2006.

[91] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[92] Guillermo Delgado and C Aporntewan. Data dependency reduction in dynamic programming matrix. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 234–236. IEEE, 2011.

[93] Ayumu Tomiyama and Reiji Suda. Automatic parameter optimization for edit distance algorithm on gpu. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 420–434. Springer, 2013.

[94] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[95] Lukasz Ligowski and Witold Rudnicki. An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[96] Ali Khajeh-Saeed, Stephen Poole, and J Blair Perot. Acceleration of the smith–waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229(11):4247–4258, 2010.

[97] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):117, 2013.

[98] Ross Anderson1 Eli Biham2 Lars Knudsen. Serpent: A proposal for the advanced encryption standard. In *First Advanced Encryption Standard (AES) Conference, Ventura, CA*, 1998.

Attachments: Digital Content

The optical disk distributed with this work contains the following items:

- This thesis in a digital form (*thesis.pdf* in the root directory).
- LaTeX source codes of this thesis including all the figures and graphs (the *thesis* directory).
- Source code of the ParallaX compiler including all the tests and experiments (the *parallax* directory).
- Source code of the native Bobox framework is included in the ParallaX compiler so we can directly integrate the experiments with it and execute them (the *parallax/NativeBobox/bobox* directory).
- The results of the experiments presented in this work are in the *results* directory. The data is in the SCV format which can be opened in Excel.
- Detailed instructions for experimental environment setup (*readme.txt*).

The digital content can also be downloaded from:

`https://data.ksi.ms.mff.cuni.cz/svn/brabec_parallax/ThesisData.zip`

The attached software contains the source code of the experiments and it is possible to build and execute them using the included batch scripts. Visual Studio with support of both the C# and C++ is necessary to build the experiments[1]. The compiler was developed and tested with Visual Studio 2015. The following guide documents the steps necessary to execute the experiments:

1. Open the solution *parallax.sln* in Visual Studio.
2. Build the solution ($Build \rightarrow Build\ Solution$).
3. Open windows console
4. Make the *parallax/Experiments* directory the working directory.
5. Build the experiment applications using either the *build.bat* or *complete_build.bat*.
   - Use the *complete_build.bat* if the MSBuild.exe is installed in the directory *%ProgramFiles(x86)%/MSBuild/14.0/Bin/MSBuild.exe*
   - Use the *build.bat* if the MSBuild.exe is located in another directory, execute: *build.bat "path_to_MSBuild.exe"*
6. Execute the experiments by running the *measure.bat*.
   - The results are in the *parallax/Experiments/results* directory.
   - The plots are in the *parallax/Experiments/plots* directory.

The unit tests are part of the *parallax* solution (*parallax.sln*). The tests must be executed in the Visual Studio using the command $Tests \rightarrow Run \rightarrow All\ Tests$ in menu.

[1] The Community edition can be downloaded from Microsoft – https://www.visualstudio.com/vs/community/. See *readme.txt* for details.

Index

Bibliography of the Author

[1] Michal Brabec and David Bednárek. Transforming procedural code for streaming environments. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 167–175. IEEE, 2017.

[2] David Bednárek, Michal Brabec, and Martin Kruliš. Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64:175–193, 2017.

[3] Michal Brabec and David Bednárek. Programming parallel pipelines using non-parallel C# code. *CEUR Workshop Proceedings*, 1003:82–87, 2013.

[4] Michal Brabec, David Bednárek, and Petr Malỳ. Transformation of pipeline stage algorithms to event-driven code. In *ITAT*, pages 13–20, 2014.

[5] Michal Brabec and David Bednárek. Procedural code representation in a flow graph. In *DATESO*, pages 89–100, 2015.

[6] Michal Brabec, David Bednárek, and Petr Malý. Transformation of pipeline stage algorithms to event-driven code. In Vera Kurkova, Lukás Bajer, and Vojtech Svátek, editors, *Proceedings of the 14th Conference on Information Technologies - Applications and Theory, Jasna, Slovakia, 2014.*, volume 1214 of *CEUR Workshop Proceedings*, pages 13–20. CEUR-WS.org, 2014.

[7] Michal Brabec and David Bednárek. Programming parallel pipelines using non-parallel c# code. In *ITAT*, pages 82–87, 2013.

[8] Martin Kruliš, David Bednárek, and Michal Brabec. Improving parallel processing of matrix-based similarity measures on modern gpus. In *International Conference on Similarity Search and Applications*, pages 283–294. Springer, 2015.