

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Causality-Based Parallelization of Input Data Processing

MASTER'S THESIS

Bc. Richard Kakaš

Brno, Spring 2018

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Causality-Based Parallelization of Input Data Processing

MASTER'S THESIS

Bc. Richard Kakaš

Brno, Spring 2018



ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Richard Kakaš
Program:	Informatika
Obor:	Paralelní a distribuované systémy
Garant oboru:	prof. RNDr. Mojmír Křetínský, CSc. (PDS)
Vedoucí práce:	prof. RNDr. Antonín Kučera, Ph.D.
Katedra:	Katedra teorie programování
Název práce:	Automatizovaná paralelizace zpracování vstupních dat na základě kauzality
Název práce anglicky:	Causality-Based Parallelization of Input Data Processing
Zadání:	Cílem práce je navrhnout a vytvořit - formální aparát (idealizovaný programovací jazyk) pro specifikaci systémů, které zpracovávají požadavky na vstupní frontě a během zpracování těchto požadavků dochází k modifikaci interních proměnných systému; - algoritmy pro automatickou paralelizaci těchto systémů, která nemá vliv na jejich funkcionalitu. Součástí práce je funkční implementace včetně několika případových studií. Rovněž je žádoucí porovnat navržené algoritmy pro automatickou paralelizaci z hlediska jejich robustnosti a efektivity.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Richard Kakaš

Advisor: prof. RNDr. Antonín Kučera, Ph.D.

Abstract

There are two ways how to improve the performance of the computer's computation - increasing the clock-speed of the computer's processing unit or attaching more processing units to the computer.

The first approach does not require any changes to the program that carries out the computations. However, there is a limit for the clock-speed which is still possible to achieve due to physical limitations of the materials used in the silicon chips.

The second approach requires changes to the program that make them parallelizable. However, it is more scalable because it does not impose any physical laws. Because of that, it makes sense to study options for automatically parallelize programs execution.

This diploma thesis describes special kind of programs that can be automatically parallelized. Performance results of the implemented tools, i.e., the scheduler and the interpreter, are presented in this diploma thesis.

Keywords

parallelization, parallel, programming language, programming paradigm, message-driven programming, event-driven programming, nodejs, antlr, static code analysis, symbolic execution, compilation, compiler, interpreter

Contents

Introduction	1
1 Motivation	3
1.1 <i>Semantic vs. paradigmatic restrictions in programming languages</i>	3
1.2 <i>Message-driven programming paradigm</i>	4
1.3 <i>Usage of the message-driven programming</i>	5
1.3.1 <i>Server-client applications</i>	5
1.3.2 <i>Graphic user interface applications</i>	6
1.3.3 <i>Node.js callback-based applications</i>	6
2 Parallel scheduler for message-driven system	7
2.1 <i>Formal model</i>	7
2.2 <i>Scheduling algorithm</i>	8
2.2.1 <i>Sequential scheduler</i>	9
2.2.2 <i>Parallel schedulers</i>	9
2.3 <i>Measuring scheduler performance</i>	12
3 Interpreter for the message-driven code	15
3.1 <i>Programming language proposal</i>	15
3.1.1 <i>Main concept and philosophy</i>	15
3.1.2 <i>Syntax and semantics</i>	17
3.1.3 <i>Built-in data types</i>	20
3.1.4 <i>Built-in functions</i>	20
3.2 <i>Static analysis of the program code</i>	23
3.3 <i>Compiling from a high-level OOP programming language</i>	30
4 Implementation	31
4.1 <i>Scheduler</i>	31
4.2 <i>Interpreter</i>	32
5 Results	35
5.1 <i>Performance of the scheduler</i>	35
5.2 <i>Accuracy of the static code analyzer</i>	38
5.3 <i>Performance of a scheduled processing by an interpreted code</i>	43
5.3.1 <i>Server-client application</i>	43

5.3.2 Graphical user interface application	45
Conclusion	49
Bibliography	51
A Running manual	55
B ANTLR lexer definition	57
C ANTLR parser definition	61
D Server-Client application	65
E Graphical user interface application	69

List of Tables

- 3.1 Built-in data types 20
- 3.2 Built-in functions for an equality checking 21
- 3.3 Built-in functions for scalar types comparing 21
- 3.4 Built-in functions for the integer type 21
- 3.5 Built-in functions for integer and floating-point arithmetic 22
- 3.6 Built-in functions for a boolean logic 22
- 3.7 Built-in functions for a string manipulation 23
- 3.8 Built-in utility functions and functions for conversions 23
- 5.1 Comparison table for processing 1000 messages 37
- 5.2 Comparison table for processing 5000 messages 37
- 5.3 Comparison table for processing 10000 messages 38
- 5.4 Server-Client application results table for a 50 seconds simulation 45
- 5.5 Server-Client application results table for a 100 seconds simulation 45
- 5.6 GUI application results table for a 50 seconds simulation 47
- 5.7 GUI application results table for a 100 seconds simulation 47

List of Figures

- 1.1 Schema of the message-driven program 4
- 1.2 Schema of the parallel message-driven program 5
- 2.1 Graph of the $P_1(x)$ function for the $|Vars| = 20$ 13
- 3.1 Syntactic support for the object type 16
- 3.2 Schema of the function declaration 17
- 3.3 Examples of the constant literals 17
- 3.4 Examples of the field names 18
- 3.5 Schema of the constant assignment statement 18
- 3.6 Schema of the copy assignment statement 18
- 3.7 Schema of the function call assignment statement 19
- 3.8 Schema of the condition statement 19
- 3.9 Schema of the return statement 20
- 5.1 Example of the code using message-dependent conditions 39
- 5.2 Boolean expressions generated for the message-dependent conditions 39
- 5.3 Example of the code using side-effect free, recursive call dependent conditions 40
- 5.4 Boolean expressions generated for the side-effect free, recursive call dependent conditions 40
- 5.5 Example of the code using global variable dependent conditions 41
- 5.6 Boolean expressions generated for the global variable dependent conditions 41
- 5.7 Example of the code using recursive side-effect function dependent conditions 42
- 5.8 Boolean expressions generated for the recursive side-effect function dependent conditions 42

Introduction

Historically, efforts for improving computer processing speed were focused on increasing computer processors' working frequencies. This is the easiest and the most straightforward way of increasing performance of computation - doing more work sequentially during the single time unit.

However, this method is reaching its physical limits, which is somewhere around 6 GHz in normal environmental conditions. For this reason, more than one processor has to be involved in the computation process to further improve its performance. To make such parallel computation possible computer program has to be specially designed - by a programmer or by an intelligent compiler, which creates the final executable program from the program description in some programming language.

The process of designing parallel program is challenging and error-prone for the programmer. Unfortunately, it is also generally impossible for a compiler to parallelize each program. However, it is possible to design compiler which can help to parallelize program written in a programming language with some specific syntactic and semantic restrictions.

This diploma thesis contains proposition of the programming paradigm and the programming language which allows automatic parallelization of the programs written according to the paradigm in the proposed language.

In the *Motivation chapter*, an auto-parallelizable programming paradigm is described, and usefulness of studying such paradigm is discussed.

The *Scheduler chapter* formally explains scheduler algorithm which is required for automatic parallelization. Methods for measuring scheduler's performance are also proposed.

In the *Interpreter chapter*, a custom low-level programming language is introduced alongside with the interpreter. This interpreter performs extensive static analysis before executing the code. Thanks to these analyses using of scheduling algorithms are possible.

The *Implementation chapter* contains a description of implementations which are part of the diploma thesis. Namely, it is the imple-

mentation of the scheduling algorithms and tools to measure their performance. Then it is the implementation of the interpreter.

In the *Results chapter*, the performance of the scheduling algorithms is discussed. In addition to that, the behavior of the whole interpreter is shown on several real-world applications.

1 Motivation

In this chapter overall aim of the diploma thesis is described. A unique programming paradigm is introduced together with the list of its possible and already existing applications [1].

1.1 Semantic vs. paradigmatic restrictions in programming languages

During last decades many programming languages emerged. Every such language has a syntax which restricts possible statements expressible in the particular programming language. Each of these statements has semantics which describes its real effect, e.g., changing variable's value and printing something to the screen [2].

Syntax and corresponding semantics are restricting programs that are expressible in the particular programming language. For example, in most functional programming languages there is no *syntactically* correct statement which has *semantics* of assigning a value to the global variable. Therefore, it is **semantically** restricted that side-effects are not allowed in the program written in such functional language. On the other hand, if side-effects free restriction is desired in an imperative programming language, which has syntax and semantics for global variable assign, functional programming paradigm has to be correctly used by the programmer. This restriction is **paradigmatic** and is very error-prone because programmer's mistakes can lead to violating it.

A good thing about **paradigmatic restrictions** is that it is possible to use multiple different programming paradigms in one general purpose language. However, because *restrictions* are not semantically required, a compiler or an interpreter often cannot perform optimizations that might be easily possible within **semantically restricted** programming language.

This diploma thesis introduces programming paradigm and programming language which **semantically** restricts programs to use this paradigm. The paradigm aims to allow automatic parallelization of the program execution.

1.2 Message-driven programming paradigm

Message-driven (i.e., event-driven) programs periodically receive messages (i.e., events) and adjust global program state according to these messages. Schema of such program, in pseudocode, is following:

```
msg := NIL
while msg != EXIT do
    msg := receiveMsg()
    processMsg(msg)
end
```

Figure 1.1: Schema of the message-driven program

According to the schema, the program is processing each received message. During this processing, the global state of the program is adjusted. This schema is, however, sequential - processing of the message has to end before next message can be processed. This may lead to the state in which messages are waiting even they might be processed in parallel. However, the parallel execution could cause race conditions, because processing of two messages can access the same global variable. Dealing with such parallel access is also often needed during transactions handling of many database systems [3]. It can be resolved in many ways, e.g.:

1. Lock each global variable before accessing it and unlock it after access is not needed anymore. This is very simple to implement, but it does not guarantee that global state is consistent during whole message processing.
2. Lock all needed global variables before starting processing message and unlock them after the message is processed. This solution guarantee that message processing sees the consistent global state. *This is also a solution that will be discussed in this diploma thesis in more details.*

For using the second solution, which allows parallel processing, scheme of the message-driven program can be altered:

```
msg := NIL
while msg != EXIT do
  msg := receiveMsg()
  readVariables := getMsgReadVariables(msg)
  writeVariables := getMsgWriteVariables(msg)
  scheduleMsgProcessing(
    msg, readVariables, writeVariables
  )
end
```

Figure 1.2: Schema of the parallel message-driven program

This schema incorporates the scheduler algorithm which can parallelize message processing based on global program variables which are read and written by messages' processing.

Description of this scheduler algorithm is provided in the *Scheduler chapter*. An algorithm for analyzing program code and extracting a list of read and written variables is described in the *Interpreter chapter*.

1.3 Usage of the message-driven programming

Message-driven programs have some *semantic* or *paradigmatic* restrictions that can make creating of these programs harder for the programmers. Despite that, such programs are currently widely used in the real-world applications of various kinds.

1.3.1 Server-client applications

This kind of applications is waiting for requests from various clients or users. When any of the requests come, it is processed by the application. Processing can involve reading or even updating the internal state of the application. These applications aim to be as fast as possible. Performance can be improved by parallel processing of incoming requests, e.g., by using the schema introduced in the sections above.

When creating a message-driven program for this kind of applications, each request can be viewed as a message. For each of the

messages a list of read and written variables is determined. This is all that scheduler needs to properly parallelize the processing of the messages.

1.3.2 Graphic user interface applications

This kind of applications is typically used by one user that interacts with the application, e.g., by clicking on the controls that are rendered by the application. In addition to processing these user events, messages that ask for application rendering or inform about new data availability can be received. These applications aim to provide smooth user experience, for which rendering at a frame rate of 60 frames per second is needed on current devices.

For this kind of applications, every user event can be viewed as a message for scheduling. Also, each render request and the data message is a message for the scheduling algorithm. Render requests are in most cases read-only, making the potential of parallelization even more significant.

1.3.3 Node.js callback-based applications

Node.js is a framework for creating scalable network programs using JavaScript language [4]. Applications written for this framework are fundamentally asynchronous. All standard input-output functions in this framework take a callback as one of the arguments. This callback is called asynchronously by the framework's runtime system to inform the program about input-output operation progress.

Despite these callbacks are called asynchronously, there are not executed in parallel. Reason for that is an impossibility to determine which portion of the global state is used by the callback. It is impossible due to a massively dynamic character of the JavaScript programming languages.

Using the scheduler and the static analyzer introduced in this diploma thesis it might be possible to create a framework that is similar to the Node.js but also parallelize callbacks execution. This is possible because calling the callback can be viewed as the message processing. This new framework will, however, have to use a programming language that is not so dynamic as JavaScript is.

2 Parallel scheduler for message-driven system

A scheduler is a component, often implemented as essential part of the operating system, which distributes processes' computations on the available computation hardware [5]. In case of this diploma thesis, distribution is restricted by usage of some shared resources - reading and writing global variables.

Before implementing such scheduler, it is beneficial to formally describe message-driven system on which this scheduler will operate. Such formal definition will make it straightforward to define a scheduling algorithm, as well as propose methods for measuring scheduler's performance and potential.

This chapter describes this formal model and whole scheduling algorithm. The provided formal model abstracts away all unnecessary parts of the message-driven system, so explaining the algorithm is as simple as possible.

2.1 Formal model

For the scheduling algorithm description, describing a message-driven system with the following mathematical structures is beneficial:

1. The finite set of all state variables identified for the system:

$$Vars = \{v_1, v_2, v_3, \dots\} \quad (2.1)$$

These are the variables that can be potentially used (read or written) by the message processing procedure.

2. The domain M of all possible messages:

$$M = \{(V_r, V_w) \mid V_r, V_w \in \{0, 1\}^{|Vars|}\} \quad (2.2)$$

Therefore, each message is represented by two binary vectors of the length $|Vars|$. The first vector is for the variables read during message processing, second is for the written variables. If i -th element of these vectors equals 1, it means that i -th variable from

the *Vars* set will be read or, in case of the second vector, written during message processing.

3. The duration function t :

$$t : M \rightarrow \mathcal{N} \quad (2.3)$$

This function assigns a number of time units that processing of the given message takes. This function is used for the simulation to measure scheduler's performance.

4. The possibly infinite sequence of messages:

$$Msgs = m_1, m_2, m_3, \dots ; m_i \in M \quad (2.4)$$

Where the M is the domain of all possible messages, as described above.

Because, during studying scheduling algorithm behavior and performance, we do not need any information about internal structure of the system state, neither about the message processing procedure, we omitted this information from the formal model definition. For our purpose, simulation of the processing, by utilizing the result of the duration function, is enough.

2.2 Scheduling algorithm

With the introduced formal model, it is possible to describe several scheduling algorithms:

1. *Sequential scheduling algorithm* - all messages are processed sequentially. Useful as a reference for performance comparison.
2. *Parallel scheduling algorithm **with** read variables locking* - messages processing is spread across multiple workers if possible. When the message is scheduled, all variables modified by its processing are **locked for reading and writing**. Useful when read-only copies of variables cannot be created or are expensive to create.

3. *Parallel scheduling algorithm **without** read variables locking* - messages processing is spread across multiple workers if possible. When the message is scheduled all variables modified by its processing are **locked only for writing**. Useful when read-only copies of variables are easy and cheap to create.

2.2.1 Sequential scheduler

Sequential scheduling algorithm is straightforward - as soon as message m arrives, process it. During simulation, this processing takes $t(m)$ time units. No variables locking has to be considered because no parallel execution is performed. This scheduler type is used in this diploma thesis as a reference for comparing the performance of the parallel schedulers.

2.2.2 Parallel schedulers

The main problem that scheduler has to solve is to determine if a received message can be processed. Based on the way how is this issue solved, there are two scheduling strategies:

1. When the message m is scheduled for processing, all state variables that will be modified by processing message m will be locked for reading and writing. The message m can be scheduled only if its processing doesn't read any already read-locked variables and if its processing doesn't write any already write-locked variables.
2. When the message m is scheduled for processing, all state variables that will be modified by processing the message m will be locked for writing. The message m can be scheduled only if its processing doesn't write any already write-locked variables. This requires that there is a read-only private copy of each variable for every parallel execution of the message processing.

These two strategies describe two variations of parallel scheduling algorithm - one **with** and another **without** read variables locking.

However, when using the scheduler **without** read variables locking, there is another issue that an algorithm has to be aware of - need

of the system state cloning. Without read variable locking it is not guaranteed that write operations, for the same variable, do not happen in the parallel. This issue can be overcome by maintaining another read-only copy of the whole system state. This copy will be altered only by the main scheduler procedure. For workers, this copy will be read-only, i.e., no parallel writes will be performed.

The scheduling algorithm, which uses N parallel workers for message processing, needs to maintain following data structures to function properly:

1. The queue Q for messages that wait for rescheduling. Initially empty.
2. The binary vector LR_i of the length $|Vars|$ for each worker $i \in [0, N)$ that indicates which variables are currently read-locked by the worker i . Initially contains $|Vars|$ zeros.
3. The binary vector LW_i of the length $|Vars|$ for each worker $i \in [0, N)$ that indicates which variables are currently write-locked by the worker i . Initially contains $|Vars|$ zeros.
4. The binary vector W of the length N that indicates which workers are currently available for processing new messages. Initially contains N ones.

The whole scheduling algorithm can be split into several procedures that are periodically executed to exhibit the expected behavior of the scheduler.

The first procedure is the $IS_SCHEDULABLE(m)$ which finds out whether the message m can be processed by any worker in the current scheduler state. It can be summarized in the following steps:

1. Make boolean-or product LW of all LW_i vectors:

$$LW = LW_1 \mid LW_2 \mid \dots \mid LW_N \quad (2.5)$$

2. Read vectors V_r and V_w of the message m .
3. Make boolean product L :

- (a) If running scheduler **with** read variables locking:

$$L = LW \ \& \ (V_r \mid V_w) \quad (2.6)$$

- (b) If running scheduler **without** read variables locking:

$$L = LW \ \& \ V_w \quad (2.7)$$

4. Return **true** only if vector L **does not** contain any bit set to one. Or in other words, return **true** only if vector L is all-zero vector.

The second procedure is the $SCHEDULE(m)$ which does the following:

1. Execute $IS_SCHEDULABLE(m)$ to check if the message m is currently schedulable.
 - (a) If the message m is schedulable continue with next steps.
 - (b) Otherwise, add the message m to the waiting queue Q and terminate execution of this procedure.
2. Try to find the i such that i -th bit of the vector W is one.
 - (a) If such i exists:
 - i. Set i -th bit of the vector W to zero.
 - ii. Send message m to the worker i to be processed.
 - (b) Otherwise:
 - i. Add the message m to the waiting queue Q .

The last procedure merges the previous ones and communicates with the workers to provide the desired functionality:

1. Whenever the new message m arrives into the system:
 - (a) Execute $SCHEDULE(m)$ for the arrived message.
2. Whenever the worker i finishes its processing:
 - (a) If running scheduler **without** read variables locking - update the read-only copy of the state to be used by the workers.

- (b) Set the LR_i vector to all zeros.
- (c) Set the LW_i vector to all zeros.
- (d) Set the i -th bit of the W vector to zero.
- (e) For each message m in the waiting queue Q :
 - i. Remove the message m from the queue Q .
 - ii. Execute $SCHEDULE(m)$ for the message.

2.3 Measuring scheduler performance

To measure performance of the schedulers, it is the best to perform simulation which executes described scheduling algorithm and simulate time-intensive message processing by busy-waiting for the amount of time specified by the duration function.

To perform the measuring simulation, it is essential to define generation process of the messages sequence $Msgs$.

Every message m can be viewed as binary vector of length $2 \cdot |Vars|$. Therefore, generating a message can be done by creating a binary vector of length $2 \cdot |Vars|$ and setting some its components to 0 and some to 1. The number of components set to 1 can be determined by specific probability distribution P_1 . After that, the chosen number of 1 components, can be randomly distributed in the vector according to probability distribution P_2 .

The probability distribution function $P_1(x)$ returns a probability of generating a binary vector with the x components set to 1. Many options for this function are possible, e.g.:

1. *Uniform distribution*:

$$P_1(x) = \frac{1}{2 \cdot |Vars|} \quad (2.8)$$

2. *Exponential distribution*:

$$P_1(x) = \lambda e^{-\lambda x} \quad (2.9)$$

Where λ is a parameter carefully chosen based on $|Vars|$. Adjusting this parameter can lead to more precise real-world simulation.

3. Normal distribution:

$$P_1(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (2.10)$$

Where σ and μ is parameters carefully chosen based on $|Vars|$. Adjusting these parameters can lead to more precise real-world simulation. This distribution was used for simulation in this diploma thesis. It is discussed more deeply in the next section.

The probability distribution function $P_2(x)$ returns a probability of the fact that x – *th* component of the binary vector is set to 1. Note that the total number of 1 components in the binary vector is bounded by the first probability distribution P_1 .

Using the normal distribution

The normal distribution is often used in the natural and social sciences to represent random variables whose distributions are unknown. [6] It actually, seems like this distribution (also sometimes called Gaussian distribution) represents many real-world social phenomena. [7]

Therefore, it is natural to use this distribution for generating random messages to simulate the scheduler. In the real-world, messages are created by the actions of the people, so using the Gaussian distribution might lead to the simulation that mimics the real-world more or less precisely.

For the message generation in the system with $|Vars|$ variables following version of the distribution function can be used:

$$P_1(x) = \frac{1}{\sqrt{2\pi|Vars|}} e^{-\frac{(x - \frac{|Vars|}{2})^2}{2|Vars|}} \quad (2.11)$$

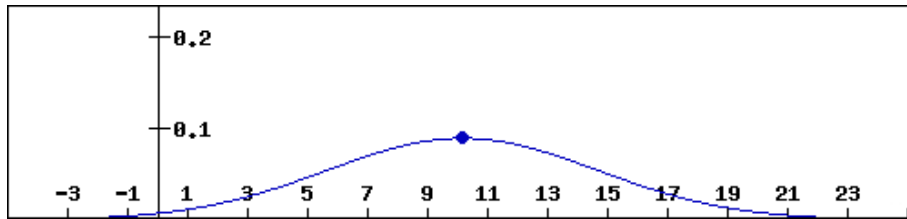


Figure 2.1: Graph of the $P_1(x)$ function for the $|Vars| = 20$

Rate limited message generation

To believably simulate real-world message-driven systems, it is necessary to choose the right time for the message generation. This problem was previously solved for the performance benchmarking of HTTP servers [8] [9]. One of the techniques used in those benchmarks is defining the message rate. The message rate is the real number that denotes the probability of generating given message.

Assume that the generation process has access to the rate r_i for each possible message m_i . The producing of new messages is performed at ticks once per time unit (e.g., one millisecond). The message m_i is produced with the probability of r_i at each tick.

This generation process, when accompanied by appropriately chosen rates r_i , can precisely simulate the real-world behavior.

3 Interpreter for the message-driven code

According to the algorithm presented in the Scheduler chapter and parallel message-driven program schema (figure 1.2) in the Motivation chapter, it is evident that list of the affected global variables by message processing is needed by the scheduler to operate.

In this chapter, the programming language is proposed whose syntax, and semantics is created explicitly to *semantically restrict* all valid programs to the message-driven programming paradigm. These restrictions aim to allow static analysis of the program code which automatically generates *read/written variables extraction function*. This function, based on the message, determines global variables that will be read or altered during program execution on the given message.

3.1 Programming language proposal

Following sections contain the syntax and the semantics of the programming language, which will be used in the experiments presented in this diploma thesis [2]. This language is more high-level than simple assembly, so it is not that hard to write programs in it. However, it is still quite low-level, so it is trivial to implement, and it is simple to analyze statically.

3.1.1 Main concept and philosophy

First of all, the proposed programming language has to be *Turing complete*. Main needed feature for *Turing completeness* is some form of the looping [10]. For this purpose, recursive calls, and therefore function declaring, will be possible in the proposed programming language.

Secondly, static analysis, which is needed to extract altered global variables, has to be possible. This is a reason why some low-level features have to be excluded, e.g., dynamic controlling of the program flow. Therefore, no dynamic address jumps, which are possible in the assembler languages, will be possible.

It is also desirable to increase the potential number of variables that will be identifiable in the programs. To achieve this the special built-in

type is present in the programming language - *object type*. This type allows the creation of the nested structures and has built-in syntactic support in the programming language:

```
global.obj.subObj.subSubObj.field = 10
```

Figure 3.1: Syntactic support for the object type

This is syntactically valid statement, which creates three objects, *obj*, *subObj* and *subSubObj*, and assign integer value to the field named *field* of the object *subSubObj*. Thanks to the fact that this is the only syntax for creating objects, it is possible that nested fields can also be identified as separate variables that are passed to the scheduler.

There are only two predefined top-level objects scoping all variables that make the syntax clear and explicit:

1. *global* - the object that represents global program state which is preserved across processing of multiple messages. The static analyzer analyzes access to this object and creates function, which returns list of *paths* (e.g. *obj.subObj.subSubObj.field*) that are *read* or *altered* by processing of specific message.
2. *local* - the object that is specific to each function call. Every function call creates new *local* object for the function and assigns arguments to the object's fields. Other fields can be created during function execution.

Every message is processed by the *main* function which is defined by the programmer. This function has only one argument of the *object type* - actual message to be processed. The return value of this function is also of the *object type* and contains information about the action that should be done by the runtime system, e.g., logging, HTTP request sending or writing to the file. This function can produce side-effects in terms of global variable assignments. This function can call other, potentially recursive, functions that can also have side-effects.

3.1.2 Syntax and semantics

Function declaration

To allow code reusing and recursive calls declaring named functions is possible with the following syntax:

```
def func(arg1, arg2)
  {Assignment statements / Control-flow statements}
end
```

Figure 3.2: Schema of the function declaration

This declaration creates function named *func* which takes two arguments. These arguments can be accessed by the *local.arg1* and *local.arg2* names. Every function has a return value, so it is not explicitly stated in the declaration. No types are declared for the function, however, each value has its type in the language. These types are checked dynamically during execution when built-in functions are called.

Literal and field name statements

Constant literals are the only way how to create new values in the programming language:

```
42
100.0
true
false
'c'
"string"
```

Figure 3.3: Examples of the constant literals

These literals can be used in more complex statements of the programming language. Exact usage is shown in the following sections.

Despite creating new values, there is also another way how to reference values in the programming language - *field names*:

```
local.x  
local.x.y  
local.x.y.y  
global.x  
global.x.y  
global.x.y.y
```

Figure 3.4: Examples of the field names

Nested structure of the field names is a very important feature of the programming language. It allows creating the nested structures that are essential for many algorithms. These structures are possible because each value under fields with the name starting with *local.x*. are contained in the *local.x* object, i.e., all these values are copied when the object *local.x* is copied.

Assignment statements

Assignments are a driving mechanism of the program computation. There are three types of the assignments:

1. *The constant assignment* which allows assignment of any valid constant literal to specific global or local field:

```
{Field name} = {Constant literal}
```

Figure 3.5: Schema of the constant assignment statement

2. *The copy assignment* which copies the value of any global or local field to another global or local field:

```
{Field name} = {Field name}
```

Figure 3.6: Schema of the copy assignment statement

3. *The function call assignment* which calls the function and copies its result to the specified global or local field:

```
{Field name} = func({Constant literal / Field name},  
                    {Constant literal / Field name})
```

Figure 3.7: Schema of the function call assignment statement

This statement calls a function named *func* with two arguments and copies its result into the *{Field name}* field.

Control-flow statements

Controlling the execution flow of the program is the essential requirement for each programming language. Without controlling execution flow, each program run will be almost the same even on different input data, and that is not enough for most of the algorithms.

For controlling execution flow, two types of statements are usable:

1. *The condition statement* which branches the execution based on field's boolean value:

```
if {Field name}  
    {Assignment statements / Control-flow statements}  
else  
    {Assignment statements / Control-flow statements}  
end
```

Figure 3.8: Schema of the condition statement

This statement checks value of the *{Field name}* field. If it is *true* first set of statements is executed, if it *false* second set of statements is executed. Any other value leads to a program termination with an error. After execution of the appropriate set of statements after the condition is executed.

2. *The return statement* which stops the execution of the current function and returns the value to the caller:

```
return {Field name / Constant literal}
```

Figure 3.9: Schema of the return statement

If the function doesn't have *return* statement or its execution doesn't reach any, special value *null* is returned. This value is discussed in the section below.

3.1.3 Built-in data types

Language provides few built-in data types, their usage is implicit and no special declaration is needed to specify usage of concrete type. Values of these types are created only by constant literals. Exception is the *object type* which allows on-demand creation by the syntax introduced in the Figure 3.1.

Table 3.1: Built-in data types

Name	Possible values	Default value	Literal examples
Object	List of variables	null	null
Boolean	true, false	false	true, false
Integer	from $-2^{63} - 1$ to $2^{63} - 1$	0	-1, -2, 1, 2
Float	approx. from $-1.7 \cdot 10^{-308}$ to $1.7 \cdot 10^{308}$	0.0	-1.5, -10.0, 1.5, 10
Char	arbitrary unicode character	U+0000	'c', '0'
String	sequence of <i>Char</i> values	""	"Hello", "World"

3.1.4 Built-in functions

In addition to built-in data types, many built-in functions, which operate on values of these types, are provided. These functions can be divided into several groups.

First of all, there are functions for an equality checking. These can operate on any given built-in type except the *object type*.

Table 3.2: Built-in functions for an equality checking

Signature	Description
<code>_eq(val1, val2)</code>	Returns <i>true</i> if both values have the same type and value, <i>false</i> otherwise.
<code>_neq(val1, val2)</code>	Returns <i>true</i> if both values doesn't have the same type or their values differ, <i>false</i> otherwise.

For scalar types, i.e., integer, floating-point and char types, value comparing is also possible. These comparing functions are defined only for arguments of the same scalar type. All other types of arguments lead to program termination with an error.

Table 3.3: Built-in functions for scalar types comparing

Signature	Description
<code>_lt(val1, val2)</code>	Returns <i>true</i> if <i>val1</i> is lower than <i>val2</i> , <i>false</i> otherwise.
<code>_gt(val1, val2)</code>	Returns <i>true</i> if <i>val1</i> is greater than <i>val2</i> , <i>false</i> otherwise.
<code>_leqt(val1, val2)</code>	Returns <i>true</i> if <i>val1</i> is lower or equal than <i>val2</i> , <i>false</i> otherwise.
<code>_geqt(val1, val2)</code>	Returns <i>true</i> if <i>val1</i> is greater or equal than <i>val2</i> , <i>false</i> otherwise.

For the integer type, special modulo function is provided. This function can be called only with both arguments set to integer values. Using any other argument types leads to a program execution error.

Table 3.4: Built-in functions for the integer type

Signature	Description
<code>_mod(val1, val2)</code>	Returns value of <i>val1</i> % <i>val2</i> .

3. INTERPRETER FOR THE MESSAGE-DRIVEN CODE

Numeric types can be used in basic arithmetic functions. These functions can operate only if both arguments are of the integer type, or both are of the floating-point type. Any other types terminate a program execution with an error.

Table 3.5: Built-in functions for integer and floating-point arithmetic

Signature	Description
<code>_add(val1, val2)</code>	Returns value of <i>val1</i> + <i>val2</i> .
<code>_sub(val1, val2)</code>	Returns value of <i>val1</i> - <i>val2</i> .
<code>_mul(val1, val2)</code>	Returns value of <i>val1</i> * <i>val2</i> .
<code>_div(val1, val2)</code>	Returns value of <i>val1</i> / <i>val2</i> .

For integer and boolean types, it is possible to define boolean logic functions. On booleans their behavior is apparent. On integers, they compute the result in the bit by bit manner. If these functions are used on any other type, a program is terminated with an error.

Table 3.6: Built-in functions for a boolean logic

Signature	Description
<code>_and(val1, val2)</code>	Performs and boolean operation between the <i>val1</i> and <i>val2</i> and returns the result.
<code>_or(val1, val2)</code>	Performs or boolean operation between the <i>val1</i> and <i>val2</i> and returns the result.
<code>_xor(val1, val2)</code>	Performs exclusive-or boolean operation between the <i>val1</i> and <i>val2</i> and returns the result.
<code>_neg(val)</code>	Flips each bit in the <i>val</i> and returns the result.

String type is unique and needs some functions for manipulation. Executing these functions on non-string type terminates a program execution with an error.

Table 3.7: Built-in functions for a string manipulation

Signature	Description
<code>_len(str)</code>	Returns length of the string <i>str</i> as an integer value.
<code>_ch(str, index)</code>	Returns character of the string <i>str</i> at position <i>index</i> as a char value
<code>_con(str1, str2)</code>	Returns concatenation of two strings.

In addition to all these functions, some more are provided for value conversions and computation simulations.

Table 3.8: Built-in utility functions and functions for conversions

Signature	Possible arguments types	Description
<code>_integer(val)</code>	Integer, Float, Boolean, Char	Returns <i>integer</i> representation of the given value.
<code>_float(val)</code>	Float, Integer	Returns <i>float</i> representation of the given integer value.
<code>_sleep(duration, val)</code>	Integer and Any	Returns <i>val</i> and simulates computation taking <i>duration</i> milliseconds.

3.2 Static analysis of the program code

Static program analysis is a process which is very often used in software verification tools [11] to catch many potential errors before a program is even executed. Some forms of the static program analysis are also used in compilers to optimize code execution [12], e.g., death code elimination.

Even though it is not apparent, optimization of the code execution is also the final aim of static analysis described in this diploma thesis. It is not evident because the static analysis is used to create extra code that will be used by the specified scheduler to parallelize a program execution.

Several steps have to be done by the analyzer to create this *guide code* for the scheduler. These steps are described in the following sections, and there are possible only because the semantics of the programming language was carefully chosen, as was shown in the previous thesis parts.

The algorithms described in the following sections are based on ideas often used in *formal verification* techniques [11] as well as in *compiler optimization* methods [12]. The algorithms' creation was mainly influenced by the following techniques:

1. *Symbolic execution* which is often used in formal verification [13].
2. *Shape analysis* which is used primarily in formal verification [14].
3. *Abstract interpretation* which is used in formal verification and compiler optimization [15].

Detection of recursive functions

Recursive functions are essential for many algorithms. However, a static analysis of such functions is, in general case, impossible. To avoid executing analyzer on recursive functions, it is needed to detect whether the function is recursive or not.

This detection can be done by a *cycle detection* algorithm [16] in the directed graph that is created according to the following description:

1. Add a vertex for each declared function.
2. Create a directed edge from the vertex of the function **f** to the vertex of the function **g** for each *function call assignment* of the function **g** in the function **f**.

Any declared function is recursive only if there exists a cycle starting in the function's vertex.

This static analysis is possible only because all function calls are semantically restricted to be static, i.e., no jumps to the computed program addresses are possible, as it is possible in the low-level assembler languages.

When dealing with the complexity of this algorithm, let the n to denote the count of declared functions in the program. If so, the created

graph will have exactly n vertices and at most n^2 edges. The simple *depth-first search* algorithm, with the time complexity $O(|V| + |E|)$, can perform the cycle detection. Therefore, the worst-case time and space complexity of this static analysis is $O(n^2)$ [17].

Creating list of read and written global fields

Each declared function (recursive or not) can potentially read or alter some global variables. Static analysis of the source code can create a list of these variables for each function.

The whole list of the variables can be obtained using *breath-first search* or *depth-first search* algorithm [16] in the directed graph that has three different types of vertices and is created by the following process:

1. Add a *function vertex* for each declared function.
2. Add a *global read vertex* for each field name, starting with the *global.* string, that appears in the source code as a condition argument or as a function argument.
3. Add a *global write vertex* for each field name, starting with the *global.* string, that appears in the source code in the left side of an assignment statement.
4. Create directed edge from the *function vertex* of the function **f** to the *function vertex* of the function **g** for each *function call assignment* of the function **g** in the function **f**.
5. Create directed edge from the *function vertex* of the function **f** to the *global read vertex* of the global variable **v** if the variable **v** is used as a *condition argument* or as a *function call argument* in the source code of the function **f**.
6. Create directed edge from the *function vertex* of the function **f** to the *global write vertex* of the global variable **v** if a variable **v** is used as a *left-side* of any assignment in the source code of the function **f**.

List of read global fields for any declared function **f** contains fields whose *global read vertex* is reachable from the *function vertex* of the **f**.

List of write global fields for any declared function f contains fields whose *global write vertex* is reachable from the *function vertex* of the f .

This static analysis would be impossible to perform if the semantics of the assignment statements would define assignment by reference. However, all assignment statements, as well as function call arguments passing, perform copying of the values. Therefore, all global fields assignments are static.

For the complexity analysis of this algorithm, let the n to denote the number of statements in the program plus the number of declared functions in the program. Therefore, number of vertices in the created graph can be bounded by the $O(n)$, and number of edges by the $O(n^2)$. The simple *depth-first search* algorithm, with the time complexity $O(|V| + |E|)$, can test reachability in the graph. Consequently, the final worst-case time complexity of the algorithm is $O(m \cdot n^2)$ where the m denotes the number of declared functions. The worst-case space complexity is $O(n^2)$ [17].

Creating boolean expression for each global variable read and write

Each global variable read and write can be located in the branch of zero or more conditions. These branches can also be nested in each other. For the scheduler, it is necessary to extract all possible conditions that lead to reading or writing of any given global variable.

The boolean expressions represent such conditions. However, they might be undetermined if they depend on the global variable itself, and not only on the incoming message. Such dependency is undesirable because using global variables during scheduling can lead to an inconsistent state of the system. Therefore, these undetermined expressions have to conclude in the global variable being treated as always read or written.

All conditions in the proposed programming language consist of the testing particular variable which has to be of the *boolean type*. Value of such variable is created by a sequence of *assignment statements*. This sequence can be easily translated into the expression which has a tree structure. Expression trees of each variable can be created by the following recursive process performed on the code of the program:

1. On a *constant assignment statement* (see figure 3.5) change the current expression for the target variable to the specific constant.
2. On a *copy assignment statement* (see figure 3.6) change the current expression for the target variable to the current expression of the source variable if it exists, or to the source variable read expression if source variable has no current expression.
3. On a *call assignment statement* (see figure 3.7) action depends on the type of the function that is being called:
 - (a) If it is a *built-in function* change the current expression for the target variable to the *function-call expression*, also replacing function's arguments with the corresponding expressions.
 - (b) If it is a *non-recursive function* recursively run this process and change the current expression for the target variable to the expression returned by the function being called.
 - (c) If it is a *recursive function* which **does not** read or write any global variable change the current expression for the target variable to the *function-call expression*, also replacing function's arguments with the corresponding expressions.
 - (d) If it is a *recursive function* which **does** read or write some global variables change the current expression for the target variable to the special *undetermined expression*.
4. On a *condition statement* (see figure 3.8) the process has to be split to determine expressions for *then* and *else* branch independently. Then, to merge the split, each changed expression is replaced by the special *if expression* which consists of the three expressions:
 - (a) *The condition expression* - expression of the variable that was in the former condition.
 - (b) *The 'then' expression* - expression that belongs to the variable in the *then* branch.
 - (c) *The 'else' expression* - expression that belongs to the variable in the *else* branch.

5. On a *return statement* (see figure 3.9) set the current expression for the current function's return to the current expression of the returned variable or to the constant expression of the returned constant. However, return statement is special because it can alter execution flow of the program. Therefore, return expression has to be guarded by the condition expression of the return's location in the code. Also, if function has multiple return statements these have to be merged by another condition expression.

This process creates trees that have leaves with *constants*, with *variables reads*, or with *undetermined expressions*. If leaf contains the *global variable read* or the *undetermined expression*, whole expression tree has to be considered as *undetermined* - such expressions are considered to be always *true* by the scheduler.

When there exists the expression for each defined variable at each point in the code, it is easy to determine when are specific global variables read or written. It can be done by traversing the graph that is created by the following procedure. The procedure is performed on the code which is obtained by *inlining* each *non-recursive side-effect* function call:

1. Add a *root vertex* which holds an *always true* expression.
2. Add a *condition vertex* for each condition that appears in the *in-lined program code*. This vertex contains the expression determine by the previously described process.
3. Add a *global read vertex* for each global variable that is read by any of the statements.
4. Add a *global write vertex* for each global variable that is assigned by any of the assignment statements.
5. Create a *then directed edge* and a *else directed edge* from each **top-level** *condition vertex* into the *root vertex*.
6. Create a *then directed edge* from each **not-top-level** *condition vertex* **v** into the *condition vertex* of the condition which contains the condition **v** in its *then branch*.

7. Create an *else directed edge* from each **not-top-level** *condition vertex* v into the *condition vertex* of the condition which contains the condition v in its *else branch*.
8. Create a *then directed edge* from each *global read or write vertex* v into the *condition vertex* of the condition which contains operation of the v directly in its *then branch*.
9. Create an *else directed edge* from each *global read or write vertex* v into the *condition vertex* of the condition which contains operation of the v directly in its *else branch*.

The boolean expression for any given *read or write vertex* v can be obtained by finding the path from the v to the *root vertex*. The final boolean expression has to satisfy each condition expression or its negation, depending on the type of the incoming edge of the condition vertex:

1. Incoming edge is *then edge* - the condition expression has to be satisfied.
2. Incoming edge is *else edge* - negation of the condition expression has to be satisfied.

With this process multiple boolean expressions are created for each *global variable read* and multiple boolean expressions are created for each *global variable write*. If all these boolean expressions evaluate to *false* it is guaranteed that message processing will not read or write given variable. This means that all global variables are implicitly locked by the scheduler, there are unlocked only if all their boolean expressions evaluate to *false*.

To analyze the complexity of the algorithm, let the n to denote the number of statements in the program. Therefore, the number of vertices of the graph is bounded by the $O(n)$, and number of edges by the $O(n^2)$. Finding the path between two vertices can be performed by the simple *depth-first search* algorithm. Hence, the total time complexity of the algorithm is $O(m \cdot n^2)$ where the m denotes the number of global variables identified in the program code. The worst-case space complexity is $O(n^2)$ [17].

3.3 Compiling from a high-level OOP programming language

The programming language presented in this diploma thesis is low-level, and does not contain many features that are expected in user-friendly languages. This is intentional because it was designed to make the implementation of the interpreter and static analyzer as simple as possible. However, this approach made the usage of the language very impractical for the programmer. Although, it is still usable as a target of a compilation from another, more high-level language.

It is even possible to create a compiler for an object-oriented programming language (OOP). There is, however, one limitation that has to be overcome to allow polymorphism feature found in the OOP languages [18]. This limitation is the lack of dynamic jumps. Polymorphism is often implemented by some virtual function table which holds in-memory addresses of the functions. These addresses are used by the function call to execute right instructions based on the polymorphic object type [19].

In general case, programs consist of several separately compiled modules. This separation of compilation makes it impossible to overcome the mentioned limitation. However, if whole program code is compiled together, these polymorphic calls can be transformed into the conditions which statically check the type of the object and make a static function call [19].

4 Implementation

This chapter describes implementations of the algorithms introduced in the previous parts of the thesis.

In the first section tools used for scheduler implementation are introduced, as well as some basic overview of the implementation is provided.

The second section contains the description of the interpreter and static analyzer implementation. Used tools and libraries are listed, an overview of the implementation is provided, and some further implementation optimizations are discussed.

4.1 Scheduler

The whole scheduler is implemented in the *C++ programming language* [20] using the standard library that is part of the *C++14 specification* [20]. Thanks to this, the implementation is portable and can be used on every platform supported by the *C++14 standard* - including Unix and Windows.

Because scheduling algorithm is inherently parallel, some parallelization construct has to be used. For this purpose, system threads feature [21], e.g., Unix POSIX threads, is used. One thread is used for the main scheduler procedure, and one additional thread is used for each scheduler's worker.

As many parallel threads are used during algorithm's execution, some synchronization of threads has to be involved. Consumer-producer queues showed up to be an appropriate data structure for such synchronization. These queues are implemented to be thread-safe by using standard *semaphores* and *conditional variables* [21].

Various control messages are being sent to the queues. These messages are then received and processed by the queue's owner thread. Every sent message has an assigned priority, which alters the order of the message processing - messages with the higher priority are processed earlier.

Following *control messages* are exchanged by the worker threads, the main scheduler thread and the *outside world*, which is generating the *messages* that should be scheduled and processed by the scheduler:

4. IMPLEMENTATION

1. Main scheduler thread receives the *control message* from the outside world that carries the *message* that should be scheduled and processed.
2. Main scheduler thread receives the *control message* from itself that carries the *message*, previously received from the outside world, that should be rescheduled. This *control message* has lower priority than *schedule control message* above.
3. Main scheduler thread receives a *control message* from the worker that informs the scheduler about finished processing and entering the idle state. This *control message* has the highest priority.
4. Main scheduler thread sends a *control message* to the worker that carries the *message* that worker should immediately start processing.

In addition to exchanging these messages, the main scheduler thread keeps track of idle workers and locked variables by different workers, i.e., information needed to determine which *outside world messages* are schedulable at any given time. Also, there are some extra *control messages* related to the terminating of the main scheduler thread, as well as worker threads. These are not essential for scheduler functionality, so there are not described here.

4.2 Interpreter

An interpreter is a program which takes a source code of the particular programming language as an input and performs semantic actions based on the source code, that should be performed according to the programming language specification.

Every interpreter generally consists of several parts:

1. *Lexer* which transforms the source code into a stream of tokens.
2. *Parser* which converts the stream of the tokens into an abstract syntax tree (AST).
3. *Executor* which traverses the AST and performs corresponding semantic actions.

The interpreter implemented as part of this diploma thesis is using the *ANTLR* [22] library to create the *lexer* and the *parser*. This library takes lexer and parser definition files and generates set of C++ classes that handle tokenization and parsing of the input source code into the AST. This AST is then executed by the *executor* which is also implemented in the C++ programming language. The parser generated by the *ANTLR* tool performs the parsing using the *LL(*)* algorithm [23]. This algorithm is based on *formal language* and *finite-state automata* abstraction and formalism [24].

The interpreter is also designed to run in an analysis mode. In this mode, special program static analysis is performed, as described in the Interpreter chapter. This analysis is also performed on the AST, and its result is set of boolean expressions that are used to determine read and written variables during the particular message processing.

For the experiments presented in the Results chapter, it was important to merge the interpreter implementation with the scheduler implementation. Both were implemented in the C++ programming language, and the scheduler implementation was designed to be as modular as possible. Thanks to this, merging both components was smooth and efficient.

The implementation was done to allow comparing performance impact of the different workers count. Therefore, it was not necessary to optimize every single part of the interpreter. Several aspects were not optimized and should be if the interpreter will be used in the real service:

1. *Object copying* is performed on each assign. This is a required semantic restriction of the programming language which makes the code analysis possible. However, this is very inefficient because many assignments create an object that is just read-only. To mitigate this inefficiency, the copy-on-write algorithm [25] might be implemented for object assignments.
2. *Boolean expressions evaluation* is performed to determine variables that are read or written by each messages processing. The boolean expressions are created by the analyzer and are always side-effect free. However, there might not be optimal and often can be simplified. The aim of this simplification is a decrease

4. IMPLEMENTATION

of the evaluation time. To achieve this improvement boolean minimization procedures [26] might be applied.

3. *Boolean expressions caching and reusing* can lead to improved performance. Generated boolean expressions often share some parts whose evaluation might be cached and reused instead of evaluating it again when it is needed. Whole expressions result might be cached for the messages that will be rescheduled in the future.

5 Results

This chapter sums up results of the experiments performed as part of this diploma thesis.

First set of experiments is related to studying performance and potential of scheduler algorithm described in the Scheduler chapter. For these experiments simulation, which tries to mimic real-world scenarios, is used.

The second set of experiments is studying performance and behavior of the program static analysis introduced in the Interpreter chapter. For these experiments, some real programs, written in the proposed programming language, are used.

5.1 Performance of the scheduler

To measure performance more precisely, real scheduler's implementation was used for scheduling randomly generated list of messages. Processing of each message was set up to take a specific number of milliseconds. Messages from the list are being sent to the scheduler in the specified interval. To compare the impact of a different number of workers and different scheduler types, the specific number of messages were fed to the scheduler and total processing time of all messages was measured.

Message generation

The scheduler operates on the system with $|Vars|$ variables. Processing of each generated message for the simulation reads and writes some of these variables.

The probability that processing of the message reads or writes x variables is following:

$$p = \frac{1}{\sqrt{2\pi|Vars|}} e^{-\frac{(x - \frac{|Vars|}{2})^2}{2|Vars|}} \quad (5.1)$$

This probability is computed based on the Gaussian distribution, as was discussed in the Scheduler chapter. Point of choosing such

5. RESULTS

complicated probability function is to simulate the real-world behavior as precisely as possible.

Processing time of the message

When $|M_R|$ and $|M_W|$ denotes the number of randomly chosen variables that are read and written during message processing, following equation specify the duration of simulated processing in milliseconds:

$$t = 2 \cdot \max\{1, |M_R|\} + 4 \cdot \max\{1, |M_W|\} \quad (5.2)$$

Therefore, it means that writing is more expensive operation than reading, that also mimics real-world behavior. Thanks to this, the simulation may be more precise.

Comparison tables

Simulation described above was run with different parameters - variables count, workers count (N) and scheduler type. Following tables are comparing total simulation time. All simulations with the same variables count were run over the same messages sequence. Therefore any random anomalies across simulation runs are not present. All simulations were run on the system with **one CPU with four cores**.

The tables are showing total *processing time* in milliseconds for the simulation runs with the different parameters. Smaller *processing time* values means better performance. The tables contain a *performance gain* information, which compares specific scheduler's processing time with the processing time of the scheduler with just one worker, i.e., sequential scheduler. Higher *performance gain* means the better performance of the particular scheduler.

From the results, it seems that performance gain is stable and does not depend on the number of processed messages. Using two workers instead of just one has increased performance gain significantly. However, adding more workers does not have such significant impact. Impact of more workers is even decreased when variables count is increased.

Table 5.1: Comparison table for processing 1000 messages

Type	N	Processing time	Performance gain
Sequential	1	5 variables - 9495 10 variables - 15459 20 variables - 29592	reference values
Read-write locking	2	5 variables - 5699 10 variables - 10942 20 variables - 24223	5 variables - 39.53% 10 variables - 29.22% 20 variables - 18.14%
Read-write locking	4	5 variables - 4970 10 variables - 10329 20 variables - 23824	5 variables - 47.26% 10 variables - 33.18% 20 variables - 19.49%
Write locking	2	5 variables - 5288 10 variables - 9115 20 variables - 19970	5 variables - 43.89% 10 variables - 41.04% 20 variables - 32.52%
Write locking	4	5 variables - 4301 10 variables - 8131 20 variables - 19365	5 variables - 54.36% 10 variables - 47.40% 20 variables - 34.56%

Table 5.2: Comparison table for processing 5000 messages

Type	N	Processing time	Performance gain
Sequential	1	5 variables - 46868 10 variables - 77173 20 variables - 148457	reference values
Read-write locking	2	5 variables - 28085 10 variables - 53137 20 variables - 118816	5 variables - 40.08% 10 variables - 31.15% 20 variables - 19.97%
Read-write locking	4	5 variables - 24634 10 variables - 50242 20 variables - 117840	5 variables - 47.44% 10 variables - 34.90% 20 variables - 20.62%
Write locking	2	5 variables - 25777 10 variables - 46015 20 variables - 97542	5 variables - 45.00% 10 variables - 40.37% 20 variables - 34.30%
Write locking	4	5 variables - 21248 10 variables - 41014 20 variables - 94227	5 variables - 54.66% 10 variables - 46.85% 20 variables - 36.53%

5. RESULTS

Table 5.3: Comparison table for processing 10000 messages

Type	N	Processing time	Performance gain
Sequential	1	5 variables - 93813 10 variables - 155388 20 variables - 297084	reference values
Read-write locking	2	5 variables - 55823 10 variables - 105807 20 variables - 236148	5 variables - 40.49% 10 variables - 31.91% 20 variables - 20.51%
Read-write locking	4	5 variables - 49031 10 variables - 100972 20 variables - 233787	5 variables - 47.74% 10 variables - 35.02% 20 variables - 21.31%
Write locking	2	5 variables - 51554 10 variables - 90769 20 variables - 194338	5 variables - 45.05% 10 variables - 41.59% 20 variables - 34.58%
Write locking	4	5 variables - 42016 10 variables - 81417 20 variables - 185631	5 variables - 55.21% 10 variables - 47.60% 20 variables - 37.52%

According to the results, *read-write* scheduler has worse performance than scheduler which utilize only *write* locking. This is expected behavior, but in real programs, this difference might vanish because of added cloning overhead. As stated in the Scheduler chapter state cloning has to performed only for the *write* locking scheduler.

To sum up, the performance of the scheduler, based on this simulation, seems to be promising. Based on these results it looks like message-driven programs executed by the proposed scheduler can exhibit significant performance improvement. However, this is still a theoretical simulation. More realistic results are provided in the next sections, where real programs are studied.

5.2 Accuracy of the static code analyzer

To make the parallelization process automatic, it is necessary to automatically extract boolean expressions for each global variable used in the program. This extraction is performed by the static code analyzer described in the Interpreter chapter. This analyzer cannot always

extract side-effect free boolean expressions. For such cases, analyzer assumes that given variable is always read or written. This section shows some general instances in which the extraction is or isn't entirely possible.

Simple message-dependent conditions

Very often, processing type that program has to perform entirely depends on the data carried by the message. Example of such program follows:

```
def main(msg)
  local.test = _eq(local.msg, "type1")
  if local.test
    global.type1data = "data"
  else
    global.type2Data = "otherData"
  end
end
```

Figure 5.1: Example of the code using message-dependent conditions

For this program, it is expected that static analyzer will generate boolean expressions that distinguish between the message of type "type1" and of another type. Such expressions were really correctly generated for the *type1data* and *type2data* variables. In lisp notation:

```
type1data: (<_and> true (<_eq> local.msg "type1"))
type2Data: (<_and> true (<_neg> (<_eq> local.msg "type1")))
```

Figure 5.2: Boolean expressions generated for the message-dependent conditions

Side-effect free, recursive call dependent conditions

Programs often declare some utility functions that check some condition about retrieved message data. This condition can be quite complicated, even recursive. Provided that this condition check function

5. RESULTS

does not read or write any global variables (i.e., is side-effect free) it is also analyzable by the implemented static analyzer. Following program shows usage of this feature:

```
def check(str, ch, index)
    # Code that can recursively call 'check'
    # function but cannot read or write any
    # global variable (or call function that does).
end
def main(msg)
    local.test = check(local.msg, 'g', 0)
    if local.test
        global.type1data = "data"
    else
        global.type2Data = "otherData"
    end
    return local.msg
end
```

Figure 5.3: Example of the code using side-effect free, recursive call dependent conditions

Boolean expressions generated for this code can freely use the *check* function because it has no side effects. The implemented static code analyzer generates such expressions:

```
type1data: (<_and> true (<check> local.msg 'g' 0))
type2Data: (<_and> true (<_neg> (<check> local.msg 'g' 0)))
```

Figure 5.4: Boolean expressions generated for the side-effect free, recursive call dependent conditions

Global variable dependent conditions

Limitation of the implemented message-driven parallelization approach is the fact that parallelization decision has to be based only on the message data. If the boolean expression needs data from the

global state, the scheduler cannot use this expression. The global state may be used by the worker, so reading the data from it is generally dangerous. Therefore, programs that contain such conditions have to be detected, and the condition has to be rewritten to be always **true**. Example of such program:

```
def check(data)
  local.test = _eq(local.data, global.data)
  return local.test
end
def main(msg)
  local.test = check(local.msg)
  if local.test
    global.type1data = "data"
  else
    global.type2Data = "otherData"
  end
  return local.msg
end
```

Figure 5.5: Example of the code using global variable dependent conditions

For this code, it was correctly identified that global variables' conditions depend on the global variable, even though dependency is indirect through the function call. Identified boolean expressions for the variables are following:

```
type1data: true
type2Data: true
```

Figure 5.6: Boolean expressions generated for the global variable dependent conditions

Recursive side-effect function dependent conditions

Another limitation of the static code analysis is a dependency on the recursive functions that are reading or writing some global variables.

5. RESULTS

If the result of the boolean expression is found to use the return value of such function, whole value of the expression has to be treated as always **true**. Example of the code that is analyzed this way is following:

```
def check(data, index)
  local.test = _eq(local.index, 0)
  if local.test
    return false
  end
  local.n = _add(local.index, global.n)
  local.test = check(local.data, local.n)
  return local.test
end
def main(msg)
  local.test = check(local.msg, 5)
  if local.test
    global.type1data = "data"
  else
    global.type2Data = "otherData"
  end
  return local.msg
end
```

Figure 5.7: Example of the code using recursive side-effect function dependent conditions

For this code analyzer correctly identified that the *check* function is the problematic one, because it reads *global.n* variable. Final boolean expressions for the *type1data* and *type2data* variables are, therefore, always **true**:

```
type1data: true
type2Data: true
```

Figure 5.8: Boolean expressions generated for the recursive side-effect function dependent conditions

5.3 Performance of a scheduled processing by an interpreted code

To measure scheduler's performance impact on real-world applications, interpretation of the real code is considered. This code was written in the programming language proposed in the Interpreter chapter. Static code analysis, also described in the previous chapter, was performed on the code to find out influenced global variables.

Experiments for two different programs were performed as part of this diploma thesis. Both experiments consist of several steps:

1. Parsing the program code written in the programming language introduced in the Interpreter chapter.
2. Analyzing the parsed AST with the static code analyzer described in the Interpreter chapter.
3. Starting the scheduler, which was proposed in the Scheduler chapter, with N workers. Workers are set-up to interpret the parsed code. The scheduler is set-up to use boolean expressions provided by the static code analyzer to determine locked global variables.
4. Schedule messages periodically over time period of T seconds. Messages are generated individually for each of the studied applications.

During the experiment execution, statistics about the number of processed messages are gathered. The performance impact of the scheduler can be determined based on the statistics.

Following sections describe programs and their messages, sum up the results of the scheduler's performance impact, and discuss precision of the static code analyzer.

5.3.1 Server-client application

This application simulates the server-client behavior of the simple database system. Such systems typically store and retrieve data from tables, or collections, or generally, so-called, *storage buckets*. In many

5. RESULTS

use cases of such systems, the retrieve requests are more frequent and sophisticated than the store requests.

The created implementation simulates the mentioned behavior in the following way:

1. Maintains **4** *storage buckets* in separate global variables. In the real application there might be real persistent data structures under these variables, for this simulation just empty strings are used and reads/writes are simulated by busy waiting.
2. Can process *read messages* that can inform program to retrieve data from 1, 2, 3 or 4 *storage buckets* at once. Reading one *storage bucket* is simulated to take *10 milliseconds*.
3. Can process *write messages* that can inform program to store data to one of the *storage buckets*. This processing is simulated to take *50 milliseconds*.

Implementation code is included in this diploma thesis (see appendix D). The code is written in the way that incoming message completely determines which *storage buckets* will be used during processing. Therefore, static code analyzer managed to correctly create boolean expressions (see chapter 3) for each global variable.

Simulation of the application load is performed by the following message generation process:

1. Every *10 milliseconds* message that reads from **1** randomly chosen *storage bucket* is generated and scheduled.
2. Every *20 milliseconds* message that reads from **2** randomly chosen *storage buckets* is generated and scheduled.
3. Every *30 milliseconds* message that reads from all *storage buckets* is generated and scheduled.
4. Every *50 milliseconds* message that writes to **1** randomly chosen *storage bucket* is generated and scheduled.

The described simulation was performed using the scheduler *with* read variables locking. Results of two simulation runs, for a different duration, are presented in the following tables:

Table 5.4: Server-Client application results table for a 50 seconds simulation

Workers count	Average processed messages count per second	Performance gain
1	45.56	reference value
2	80.9	77.57%
4	93.66	105.58%

Table 5.5: Server-Client application results table for a 100 seconds simulation

Workers count	Average processed messages count per second	Performance gain
1	43.99	reference value
2	79.51	80.75%
4	96.89	120.25%

The results show that performance gain is stable and does not depend on the length of the simulation. Also, adding more workers increases the performance of the system. The performance gain will stabilize at some number of workers - based on the number of *storage buckets* and mainly the number of processors available on a test machine. Simulations were run on the test machine with a dual-core processor with a hyper-threading feature, i.e., it had, theoretically, four processor cores.

5.3.2 Graphical user interface application

This application simulates the behavior of the graphical user interface (GUI) program. The primary task of these programs is to render the interface and respond to the user events such as a mouse move and mouse clicks. They also, typically, parse some data and show them in the graphical interface.

The experimental implementation simulates this behavior with the following process:

5. RESULTS

1. Maintains *GUI state* under one global variable. In real applications, there will be complex data structure under this variable or even more global variables. For this simulation, just one global variable is used.
2. Maintains *data* and *real-time data* under two separate global variables. In real applications, there will be complex data structure under these variables. For this simulation, just simple counters are used.
3. Can process *gui update messages* that inform the program to update the GUI state. Processing of this message is instant, i.e., without any busy waiting simulation.
4. Can process *data update messages* that inform the program to update the *data* variable. This update is simulated by busy waiting that lasts *50 milliseconds*.
5. Can process *real-time data update messages* that inform the program to update the *real-time data* variable. This update is simulated by busy waiting that lasts *10 milliseconds*.
6. Can process *render messages* that inform the program to render whole application. Rendering reads all global variables in the system. Rendering is simulated by busy waiting that lasts *10 milliseconds*.

Implementation code is included in this diploma thesis (see appendix E). Static code analyzer managed to correctly identify which global variables are read and written by any of the messages.

Simulation of the application load is performed by the following message generation process:

1. Every *1 millisecond* message that updates *GUI state* is generated and scheduled.
2. Every *500 milliseconds* message that updates *data global variable* is generated and scheduled.
3. Every *50 milliseconds* message that updates *real-time data global variable* is generated and scheduled.

4. Every *16 milliseconds* message that causes rendering of the graphical user interface is generated and scheduled.

The simulation was performed using the scheduler **without** read variables locking. This scheduler is especially suitable for this kind of the applications because of their relatively small internal state that can be easily cloned. Results of two simulation runs, that show count of all processed messages and number of processed render messages, are included in the following tables:

Table 5.6: GUI application results table for a 50 seconds simulation

Workers count	Average processed messages count per second	Performance gain
1	all - 495.94 render - 30.42	reference values
2	all - 875.9 render - 57.06	all - 76.61% render - 87.57%
4	all - 978.12 render - 61	all - 97.23% render - 100.53%

Table 5.7: GUI application results table for a 100 seconds simulation

Workers count	Average processed messages count per second	Performance gain
1	all - 501.74 render - 30.72	reference values
2	all - 864.18 render - 55.51	all - 72.24% render - 80.7%
4	all - 988.32 render - 61.12	all - 96.98% render - 98.96%

The results show that performance gain is stable and does not depend on the duration of the simulation. Adding more workers increased the overall performance. In this experiment, it was also important to monitor *render messages*. The frequency of these messages was set to one per *16 milliseconds* because this frequency is needed to get smooth *60 frames per second (FPS)*. The results show that using

5. RESULTS

4 workers helped to achieve this goal. Only 1 or 2 workers were not sufficient to maintain the smooth FPS.

Conclusion

This diploma thesis aimed to address an issue of computation performance increasing limitations. Increasing working frequencies of computation units cannot further improve a computation performance. Therefore, parallelization has to be involved. Parallelization process is in most cases complicated and has to be done by programmers.

To leave out the programmer from the parallelization process the special programming language was created. This programming language allows creating message-driven applications that can often solve real-world problems. These applications are automatically parallelizable by the special scheduling algorithm. The scheduler is guided by the information about the code that are extracted by the static code analyzer.

The performance impact of the auto-parallelization process was measured in this diploma thesis. It showed up as quite significant. Implemented static analyzer tools managed to identify parts of the program that do not depend on each other. Proposed scheduler was able to correctly distribute the computation process across multiple workers.

The implemented tools are just the simple proof-of-concept. Their implementation was ineffective in some places. These concerns were discussed in the Implementation chapter. To use these tools in real applications, it would be necessary to solve these issues. It could also be interesting to extend the tools and create a programming language that is more resemble to popular object-oriented languages and uses the philosophy of the Node.js framework (see the Motivations chapter).

Bibliography

1. AMBLER, Allen L.; BURNETT, Margaret M.; ZIMMERMAN, Betsy A. Operational versus definitional: A perspective on programming paradigms. *Computer*. 1992, vol. 25, no. 9, pp. 28–43.
2. WATT, David Anthony. *Programming language syntax and semantics*. Prentice Hall PTR, 1996.
3. *ACM Transactions on Database Systems (TODS)*. New York, NY, USA: ACM. ISSN 0362-5915.
4. TEIXEIRA, Pedro. *Professional Node.js: Building Javascript Based Scalable Software*. Wrox, 2012. ISBN 1118185463.
5. LIU, C. L.; LAYLAND, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*. 1973, vol. 20, no. 1, pp. 46–61. ISSN 0004-5411. Available from DOI: 10.1145/321738.321743.
6. CASELLA, George; BERGER, Roger L. *Statistical Inference*. Duxbury Press, 2001. ISBN 0534243126.
7. LYON, A. Why are Normal Distributions Normal? *The British Journal for the Philosophy of Science*. 2014.
8. BARFORD, Paul; CROVELLA, Mark. Generating representative web workloads for network and server performance evaluation. In: *ACM SIGMETRICS Performance Evaluation Review*. 1998, vol. 26, pp. 151–160. No. 1.
9. MOSBERGER, David; JIN, Tai. httpperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*. 1998, vol. 26, no. 3, pp. 31–37.
10. TELLER, Astro. Turing completeness in the language of genetic programming with indexed memory. In: *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. 1994, pp. 136–141.
11. WALLACE, Dolores R.; FUJII, Roger U. Software verification and validation: an overview. *Ieee Software*. 1989, vol. 6, no. 3, pp. 10–17.

BIBLIOGRAPHY

12. BARRETT, William A; COUCH, John D. *Compiler construction: theory and practice*. Science Research Associates Chicago, 1979.
13. KING, James C. Symbolic execution and program testing. *Communications of the ACM*. 1976, vol. 19, no. 7, pp. 385–394.
14. SAGIV, Mooly; REPS, Thomas; WILHELM, Reinhard. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 2002, vol. 24, no. 3, pp. 217–298.
15. COUSOT, Patrick; COUSOT, Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.
16. DIESTEL, Reinhard. *Graph Theory*. Springer, 1997. Graduate Texts in Mathematics, no. 173.
17. COOK, Stephen A. An overview of computational complexity. *Communications of the ACM*. 1983, vol. 26, no. 6, pp. 400–408.
18. SMITH, Ben. Object-oriented programming. In: *Advanced Action-Script 3*. Springer, 2015, pp. 1–23.
19. HARPER, Robert; MORRISETT, Greg. Compiling polymorphism using intensional type analysis. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 130–141.
20. ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2012. Available also from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.
21. ARPACI-DUSSEAU, Remzi H.; ARPACI-DUSSEAU, Andrea C. *Operating Systems: Three Easy Pieces*. .910th ed. Arpaci-Dusseau Books, 2015.
22. PARR, Terence. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. ISBN 1934356999.

BIBLIOGRAPHY

23. SIPPU, Seppo; SOISALON-SOININEN, Eljas. *Parsing Theory: Volume II LR (k) and LL (k) Parsing*. Springer Science & Business Media, 2013.
24. ROZENBERG, Grzegorz; SALOMAA, Arto. *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer Science & Business Media, 2012.
25. RODEH, Ohad. B-trees, Shadowing, and Clones. *Trans. Storage*. 2008, vol. 3, no. 4, pp. 2:1–2:27. ISSN 1553-3077. Available from DOI: 10.1145/1326542.1326544.
26. DUŞA, Adrian. A mathematical approach to the boolean minimization problem. *Quality & Quantity*. 2010, vol. 44, no. 1, pp. 99.

A Running manual

The executable program was implemented as part of this diploma thesis. Source code files for this program are delivered on the memory medium that comes with the diploma thesis, or can be downloaded from the following GIT repository:

`https://github.com/w0301/dp\_thesis`

To build the executable, following steps are needed:

1. Change current working directory to the main source directory located on the memory medium:

```
cd "interpreter/"
```

2. Create the build directory and enter it:

```
mkdir -p "build/" && cd "build/"
```

3. Run *CMake* to generate makefiles:

```
cmake ..
```

4. Run *make* to compile the source codes:

```
make
```

5. Sometimes, the first build fails because of some *CMake* dependencies issues. If it happened rerun *CMake* and *make* again.

```
cmake .. && make
```

The executable allows to perform all experiments that were presented in this diploma thesis:

1. To perform simple scheduler performance experiment:

```
./build/interpreter --test-scheduler \  
  <messages_count> <variables_count>
```

Here the *<messages_count>* and the *<variables_count>* are integer parameters.

2. To perform server-client application test:

```
./build/interpreter --test-server <duration>
```

Here the *<duration>* is an integer parameter.

3. To perform GUI application test:

```
./build/interpreter --test-gui <duration>
```

Here the *<duration>* is an integer parameter.

4. To interpret specific source code file:

```
./build/interpreter <path_to_file> <arg>
```

Here the *<path_to_file>* is a path of the file that will be interpreted and the *<arg>* is a string argument that will be passed to the main function.

B ANTLR lexer definition

The ANTLR library was used to generate the programming language lexer. Following code was used as an input for the ANTLR tools to generate C++ lexer classes:

```
lexer grammar LangLexer;  
  
INTEGER  
  : [1-9] [0-9]*  
  | [0]  
  | [\ -] [1-9] [0-9]*  
  ;  
  
FLOAT  
  : [0-9]+ '.' [0-9]+  
  | [\ -] [0-9]+ '.' [0-9]+  
  ;  
  
BOOL  
  : ('true' | 'false')  
  ;  
  
CHAR  
  : '\ ' . '\ '  
  ;  
  
STRING  
  : '"' .*? '"'  
  ;  
  
OBJ  
  : 'null'  
  ;  
  
EQ  
  : '='  
  ;
```

B. ANTLR LEXER DEFINITION

```
DOT
:  '.'
;
```

```
COMMA
:  ','
;
```

```
COLON
:  ':'
;
```

```
L_PAREN
:  '('
;
```

```
R_PAREN
:  ')'
;
```

```
DEF
:  'def'
;
```

```
IF
:  'if'
;
```

```
ELSE
:  'else'
;
```

```
RETURN
:  'return'
;
```

```
END
    : 'end'
    ;

// identifier tokens comes last!
NAME
    : [a-zA-Z_] [a-zA-Z0-9_]*
    ;

GLOBAL_OBJ_PATH
    : 'global' (DOT NAME)*
    ;

LOCAL_OBJ_PATH
    : 'local' (DOT NAME)*
    ;

// skipping comments
COMMENT
    : '#' ~( '\r' | '\n' )* -> channel(HIDDEN)
    ;

// skipping whitespaces
WHITESPACE_SKIP
    : [ \t\r\n]+ -> channel(HIDDEN)
    ;
```


C ANTLR parser definition

The ANTLR library was used to generate the programming language parser. Following code was used as an input for the ANTLR tools to generate C++ parser classes:

```
parser grammar LangParser;  
  
options {  
    tokenVocab = LangLexer;  
}  
  
integerValue  
    : INTEGER  
    ;  
  
floatValue  
    : FLOAT  
    ;  
  
boolValue  
    : BOOL  
    ;  
  
charValue  
    : CHAR  
    ;  
  
stringValue  
    : STRING  
    ;  
  
objValue  
    : OBJ  
    ;  
  
constantValue  
    : integerValue
```

C. ANTLR PARSER DEFINITION

```
| floatValue
| boolValue
| charValue
| stringValue
| objValue
;

globalIdentifier
: GLOBAL_OBJ_PATH
;

localIdentifier
: LOCAL_OBJ_PATH
;

identifier
: localIdentifier
| globalIdentifier
;

value
: constantValue
| identifier
;

functionName
: NAME
;

functionArgName
: NAME
;

functionArg
: value
;
```

```
functionCall
    : functionName L_PAREN
      (functionArg (COMMA functionArg)*)? R_PAREN
    ;

assignmentStatement
    : identifier EQ value
    | identifier EQ functionCall
    ;

conditionStatement
    : IF identifier thenStatements END
    | IF identifier thenStatements ELSE statement
    | IF identifier thenStatements ELSE elseStatements END
    ;

returnStatement
    : RETURN value
    ;

statement
    : assignmentStatement
    | conditionStatement
    | returnStatement
    ;

thenStatements
    : statement+
    ;

elseStatements
    : statement+
    ;

statements
    : statement+
    ;
```

C. ANTLR PARSER DEFINITION

```
function
: DEF functionName L_PAREN (functionArgName
    (COMMA functionArgName)*)? R_PAREN statements END
;

file
: function*
;
```

D Server-Client application

Following code was used by the implemented interpreter in one of the experiments presented in the Results chapter:

```
def simulateWrite(db)
    local.db.writes = _add(local.db.writes, 1)
    local._ = _sleep(50, 0)

    return local.db
end

def simulateRead(db)
    local.db.reads = _add(local.db.reads, 1)
    local._ = _sleep(10, 0)

    return local.db
end

def isBitSet(bits, n)
    local.nCh = _ch(local.bits, local.n)
    local.test = _eq(local.nCh, '1')
    return local.test
end

def main(msg)
    # initialization
    local.test = _eq(local.msg.type, "init")
    if local.test
        global.db1.data = ""
        global.db1.reads = 0
        global.db1.writes = 0

        global.db2.data = ""
        global.db2.reads = 0
        global.db2.writes = 0

        global.db3.data = ""
    end
end
```

D. SERVER-CLIENT APPLICATION

```
        global.db3.reads = 0
        global.db3.writes = 0

        global.db4.data = ""
        global.db4.reads = 0
        global.db4.writes = 0
    end

    # do real work
    local.test = _eq(local.msg.type, "work")
    if local.test
        # doing required writes
        local.writeDb1 = isBitSet(local.msg.writes, 0)
        local.writeDb2 = isBitSet(local.msg.writes, 1)
        local.writeDb3 = isBitSet(local.msg.writes, 2)
        local.writeDb4 = isBitSet(local.msg.writes, 3)

        if local.writeDb1
            local.newDb = simulateWrite(global.db1)
            global.db1.data = local.newDb.data
            global.db1.writes = local.newDb.writes
        end

        if local.writeDb2
            local.newDb = simulateWrite(global.db2)
            global.db2.data = local.newDb.data
            global.db2.writes = local.newDb.writes
        end

        if local.writeDb3
            local.newDb = simulateWrite(global.db3)
            global.db3.data = local.newDb.data
            global.db3.writes = local.newDb.writes
        end

        if local.writeDb4
            local.newDb = simulateWrite(global.db4)
```

```
        global.db4.data = local.newDb.data
        global.db4.writes = local.newDb.writes
    end

    # doing required reads
    local.readDb1 = isBitSet(local.msg.reads, 0)
    local.readDb2 = isBitSet(local.msg.reads, 1)
    local.readDb3 = isBitSet(local.msg.reads, 2)
    local.readDb4 = isBitSet(local.msg.reads, 3)

    if local.readDb1
        local.newDb = simulateRead(global.db1)
        global.db1.reads = local.newDb.reads
    end

    if local.readDb2
        local.newDb = simulateRead(global.db2)
        global.db2.reads = local.newDb.reads
    end

    if local.readDb3
        local.newDb = simulateRead(global.db3)
        global.db3.reads = local.newDb.reads
    end

    if local.readDb4
        local.newDb = simulateRead(global.db4)
        global.db4.reads = local.newDb.reads
    end
end

return local.msg
end
```


E Graphical user interface application

Following code was used by the implemented interpreter in one of the experiments presented in the Results chapter:

```
def simulateGuiStateUpdate(guiState)
  local.guiState.writes =
    _add(local.guiState.writes, 1)
  return local.guiState
end

def simulateDataUpdate(data)
  local.data.writes = _add(local.data.writes, 1)
  local._ = _sleep(50, 0)
  return local.data
end

def simulateRealTimeDataUpdate(data)
  local.data.writes = _add(local.data.writes, 1)
  local._ = _sleep(10, 0)
  return local.data
end

def simulateRender(guiState, data, realTimeData)
  local._ = _sleep(10, 0)
  return local.data
end

def main(msg)
  # initialization
  local.test = _eq(local.msg.type, "init")
  if local.test
    global.guiState.writes = 0

    global.data.writes = 0

    global.realTimeData.writes = 0
  end
end
```

```
# handling update messages
local.test = _eq(local.msg.type, "gui")
if local.test
    global.guiState = simulateGuiStateUpdate(
        global.guiState
    )
end

local.test = _eq(local.msg.type, "data")
if local.test
    local.newData = simulateDataUpdate(
        global.data
    )
    global.data.writes = local.newData.writes
end

local.test = _eq(local.msg.type, "realTimeData")
if local.test
    local.newData = simulateRealTimeDataUpdate(
        global.data
    )
    global.realTimeData.writes = local.newData.writes
end

local.test = _eq(local.msg.type, "render")
if local.test
    local._ = simulateRender(
        global.guiState, global.data, global.realTimeData
    )
end

return local.msg
end
```