Abstract

In this report we will present a concurrency oriented real-time language we have created. We will present the parallelization patterns applied, and the structure of the compiler. The language is based on a sub-set of occam's grammar, it includes features such as fine grained parallelism and message passing between processes. During compilation the compiler generates processes, and maps how they depend how they depend on each other.

This is made possible by limiting the ways processes can communication with each other. The language runs on a multiprocessor run-time system. It has one worker thread pr. processor, which executes processes created by the compiler. The processes are scheduled with a multiprocessor earliest deadline first (EDF) scheduler. Two different versions of the EDF scheduler were implemented: One pure EDF scheduler and one modified EDF scheduler that utilize the dependency analysis performed by the compiler. The dependency analysis based scheduler runs the process with the earliest deadline concurrently with the processes it depends on. This enables the workers to work together towards the deadline of the process with the earliest deadline. This is an implementation that enables real-time application to utilize the power of multiprocessor systems.

Performance data and examples is also presented. The examples demonstrate how scheduling based on dependency analysis is able to reach deadlines, which the pure EDF scheduler does not reach. Standard performance tests was performed on our language and several other languages, it showed surprisingly that our system performed as well and even better than some of the other languages. This is likely due to the simple and light structure of our system.

Preface

This report, A concurrency Oriented Real-Time Language, is the written fulfillment of a Master Degree completed at the Department of Engineering Cybernetics, at the Norwegian University of Science and Technology. It describes the language created during this project. Most work was put in to the implementation of the language, and hopefully this report reflects the work done in the implementation.

I would like to use this opportunity to thank some people that supported me during this project.

I would like to thank my supervisors Sverre Hendseth and Martin Korsgaard, for there good advice and criticism. I have to give a special thanks to Martin for always taking the time to help me with Haskell problems.

And a special thanks to my father for supporting and believing in me, during this project.

Jon Tore Hafstad

Trondheim June 5, 2010

Contents

Chapter 1

Introduction

The scope of this report is the present the experimental real-time language we have created. The language is based on known concurrency oriented languages. We will also present the multiprocessor run-time system created to support the experimental language. The run-time system uses two real-time schedulers implemented, where one of the utilizes compile time analysis to improve scheduling. We will present how these two schedules differently, and their supporting features.

The focus of this project has been to let real-time systems utilize multiprocessor systems. As the multiprocessor architectures has become more and more common, and can be found in desktop machines, laptops and even embedded systems. The focus on utilizing the new architecture has grown. Within the real-time community there has also been done more research on this field. Most of the multiprocessor real-time schedulers are still in the experimental phase, and has been tested in experimental operating systems. As far as we could find out there is no languages that support multiprocessor real-time system as a part of their run-time system. There has essentially been little focus on creating tailor made real-time languages.

The scope of this report is to explain how our language and run-time system was designed and built. We will not go into specific details on the implementation, but give an overview and explain the design of different parts of the system. Attached to this report is the complete source code of the compiler and the run-time system, plus a few examples. For further details on mechanisms presented in this report, please see the attached source code.

The report starts by giving a short introduction to compiler techniques, patterns for parallel programming, and an overview of known concurrent lan-

guages new and old. Then a short introduction on real-time multiprocessor schedulers based on our survey on Real-Time multiprocessor schedulers [17]. We will then present the grammar of the experimental language, compiler generated code, and the design of the run-time system. The experimental language will then be demonstrated, and performance results will be compared with existing concurrent languages. At the end we will discuss the results, and what we consider missing features in the language.

1.1 Problem Description

A problem description for our initial goals was formulated:

Create a real-time language focusing on concurrency and parallelization, which can utilize the potential of a multiprocessor system:

(a) Research what language features concurrency oriented language offer, and present them.

(b) Investigate how the design of their run-time architecture support the concurrency oriented language.

(c) Define a subset of the grammar to a concurrency oriented language. Limit the language to only include those mechanism necessary for demonstrating communication and concurrency mechanisms.

(d) Create a compiler based on the defined grammar, and a scalable run-time system.

(e) Investigate if dependency analysis can be utilized by a multiprocessor real-time scheduler. If possible implement a scheduler utilizing these principles.

1.2 Initial Status of the Project

This project is a continuation of our survey on multiprocessor real-time scheduler and porting of the Linux Scheduler Simulator (LinSched). It provided us with vital knowledge about the challenges with utilizing multiprocessor systems in a real-time manner. Our experience with the Linux kernel was also useful when it came evaluating OS performance versus, implementing the mechanisms our self. We wanted to create a simple and effective system, and get away from the complexity of the kernel.

We were also able to base parts of our project on observations and implementation done by Martin Korsgaards in his Time/occam language [**?**]. We where able to use parts of the grammar specified, as we both had occam as bases.

I already had some experience with compiler construction. In an earlier projects in TDT4205, I had build a compiler using the compiler construction tools lex & yacc. These tools however proved them self to be inadequate when it came constructing a more complex compiler. We chose to use some newer tools. They will be presented later in this report. It required that I had to learn new tool (BNFC), and a new programming language (Haskell).

Chapter 2

Background

In this chapter compilers structure and parallelism will be emphasized. We will also introduce multiprocessor real-time schedulers, based on our survey on the subject [17], which contains further details on different algorithms.

In the section 2.1 language grammar, generated code, and compiler generator tools will be presented. Section 2.3 covers parallelization patterns, and presents mechanisms for identifying parallelism at compile time. It also includes a short introduction to a few concurrent programming languages.

The run-time systems of these languages is presented in section 2.4. To be able to appreciate this part of the report, the reader must be familiar with concurrency principles, such as processes, threads, mutual exclusion, race condition (to get up to speed we recommend Patterns for Parallel Programming [25]). The last section is a short summary of the real-time multiprocessor schedulers presented in [17], with some new material.

2.1 The Structure of a Compiler

This section is based on the explanation and terms used in the "dragon book" [5]. The structure of a compiler described in this section is divided into four parts, lexical analysis, syntax analysis, semantic analysis and code generating. This is what is described as a complier front end. We will not focus on the back end which generates computer instructions, as it is a very complex system and not relevant for the rest of the report.

2.1.1 Lexical Analysis

Lexical analysis is the first part of a compiler. It identifies sequences of characters that represent meaningful code, called lexmes. The lexical analyzer or "lexer" generates a token from each identified lexme, where each token holds a value and a type.

```
token <token name, attribute value>
```

The lexical analyzer is based on the grammar specified for the language. For instance, a grammatical rule for declaring a variable:

```
1 int c
```

Listing 2.1: Int variable

In this example int is a type token with `int` as value, and c is of the type variable and `c` is the value. Let us take a look at a bit more complex example:

```
1 c = b + a
```

Listing 2.2: Summation

It consists of two operations; addition of `b` and `c`, and assigning the sum to `a`. To interpret this as intended, we have to define rules for the lexer.

We define the rule for addition to:

```
identifier = identifier + identifier
```

The rule defines that on the left side of the equal sign there has to be an `identifier`, and on the right hand side of the equal sign there must be a summation expression. The summation expression consists of two identifiers on each side of a plus sign. Without rules for what character an `identifier` can consists of the grammar is ambiguous. As an expression such as `a + b` either could be an identifier or an expression for a summation. To avoid this we additionally say that a variable name has to consist of lower case characters from [a-z]. This prevents the grammar from being unambiguous.

When running a lexer with the defined grammar on the addition in listing 2.2, it starts with identifying the `a` as an identifier, and maps it to a token `<id, 1>`, where 1 corresponds to its position in the symbol table. Due to our rule for what characters a identifier can consist of the lexer knows that the equal can't be a part of the identifier. `=` does not have any attributes and is represented by the token `<=>`. Again both a and b is mapped to an id and its symbol table position. Due to our rule for the lexical structure of a identifier the lexer knows what is a lexeme (a group of character that is meaningful). The resulting token sequence generated by the lexer is then

```
1 <id , 1> <=> <id , 2> <+> <id , 3>
```

Listing 2.3: Summation token stream

To define a complete language grammar, rules such as the ones above has to be defined for every single type of statement and expression in the programming language. This can be done together with the syntactical rules, in the grammar. Which we will demonstrate later in this chapter.

2.1.2 Syntax Analysis / Parser

Syntax analysis or parsing uses the token stream generated by the lexer to create a representation of the operations, and groups the statements. This is necessary to get an unambiguous interpretation of source code. The result of parsing is a complete syntax tree and a symbol table. The syntax tree consists of the tokens generated by the lexer grouped together by the parser. The symbol table holds the corresponding values to the ids in the syntax tree.

Let us take a look at a simple subtraction and multiplication, and see why it can ambiguous:

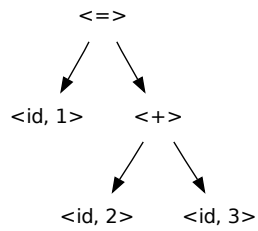

Figure 2.1: Syntax tree produced of the token stream in listing 2.3

```
1 a = a - b * c
```

Listing 2.4: Multiplication and subtraction

There is two way to perform calculation in the code in listing 2.4. Either do the subtraction `a - b` and then multiply `c` with the result from the subtraction; or `b * c` can be calculated first and the result of this multiplication is then subtracted from `a`. These two interpretation will cause different results. A language grammar therefor has to specify which of the operations that has precedence over the other operations. The parser then groups the tokens according to these rules.

Figure 2.1 shows the syntax tree generated by the parser from parsing the token stream from the last section (listing 2.3). Where each inner node in the tree represent an operation, and the children of these nodes represent the arguments to the operation. The syntax tree holds two operations, assign `<=>` and a summation `<+>`. The summation operation is one of the arguments of the assignment. The leaf nodes holds the rest of the arguments. The parsing of the summation is pretty straight forward, because it is only one way the token stream can be read.

When generating the syntax tree for the multiplication and subtraction presented in listing 2.4, the parsers has to use the preference rules to generate the tree in figure 2.4. Multiplication `<*>` has a higher precedence than subtraction `<->`, and is a child node of the subtraction. If the semantic analyzer reads from the bottom right (which is does in this case) the multiplication is the first expression to be read.

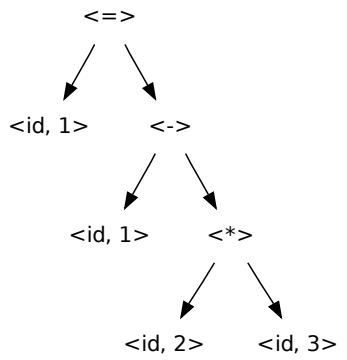

Figure 2.2: Syntax tree produced of the source code from listing 2.4

2.1.3 Grammar

The rules defined for the lexer and parser, creates what we refer to as a grammar. There exists several standards form for expressing the grammar of a language. Some of these can be read by compiler generator tools, that can generates parts of the compiler based on the grammar. The most used standard grammar is called context free grammar, also known as Backus-Naur Form (BNF).

A context free grammar consist of four elements

1. Terminal symbols or tokens, as shown in section 2.1.1. These are the basic elements of the language. Such as numbers, variables, characters etc.

2. Nonterminals are symbols that can be reduced. They can consist of terminals or nonterminals. A nonterminal could for instance be a definition of a list that is defined to consists of digits.

3. Productions, consists of a nonterminal that is the head of the production; the head is placed on the left hand side of an arrow, on the right hand side of the arrow there is a sequence of terminals and/or nonterminals describing what the terminal on the right hand side consists of. The sequence on the right hand side is referred to as the body of the production.

4. A definition of one of the nonterminals as start symbol, the highest level of abstraction.

If we want to put these rules to use we can create a grammar that describes a summation:

```
1 + 2 + 3
```

Listing 2.5: Summation

```
add -> digit + digit
digit -> [0-9]
```

Listing 2.6: Simple context-free grammar

The grammar defined in listing 2.6 is very simple, it only supports summation of two `digit`s. For instance, the operation in listing 2.5 would not be accepted by this grammar, as both the left and right hand side of the `+` must be a digit. To be able to interpret such a summation correctly we have to extend grammar to:

```
statement -> statement + digit | digit
digit -> [0-9+]
```

Listing 2.7: Simple context-free grammar

In this grammar a `statement` can consists of either an `statement` and a `digit`, or a single `digit`. The definition of a `statement` is recursive, which means that a `statement` can be built up from a one or more `statement`s.

Lets take a look at how the code in listing 2.5 will be reduced using the new grammar. If we start reading the source code from the left (left associative) we can see that the first character we encounter is `1`. It matches the definition of a `digit`, and is reduced to a `digit`. A `statement` can consist of a single `digit`, hence it can be further reduced to a `statement`. The next character is `+`, there is no production that matches only a `digit` and a `+`. To be able reduce it to an `statement` we have to shift the next character. The next character is a `2`, which we reduce to a `digit`. What we have so far is then `statement + digit`, which can be reduced to a complete `statement`. The rest of the source code `+ 3` can be reduced in the same manner as the first summation. These reductions is done step by step in listing 2.8, and gives us the syntax tree in figure 2.3

```
1                   shift 1
digit               reduce 1 -> digit
statement           reduce digit -> statement
statement +         shift +
statement + 2       shift 2
statement + digit   reduce 2 -> digit
statement           reduce statement -> statement +
    digit
statement +         shift +
statement + 3       shift 3
statement + digit   reduce digit -> 3
statement           reduce statement -> statement +
    digit
```

Listing 2.8: Deriving listing 2.5 using the CFS in listing 2.7

If we want to expand this grammar to also support multiplication we have to define what operation has precedence over the other in the grammar. So we know if summation or multiplication should be reduced first. In listing 2.9 the grammar from listing 2.7 is modified to accept multiplication and summation, defining multiplication to the highest precedence level.

Associativity is expressed by stating that the nonterminal `expr` must be on the left hand side of `'+'`, and a `digit` on the right, hence left associative.

The precedence level is expressed by defining two nonterminals `expr` and `term`. A `term` must either consume a multiplication or the nonterminal `factor`, which is either a `digit` or the nonterminal `expr` wrapped in parenthesis. The nonterminal `expr` can either consume a summation, or a `term`. This production state that a `expr` must be reduced before a `term` can be reduced, hence multiplication has the highest precedence.

```
statement -> expr | term
expr -> expr '+' term | term
term -> term '*' factor | factor
factor -> digit | (expr)
digit -> [0-9+]
```

Listing 2.9: Grammar for multiplication and summation, with precedence and associativity defined

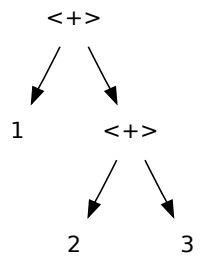

Figure 2.3: Syntax tree produced when doing the reduction in listing 2.8

2.1.4 Semantic Analysis

Semantic analysis takes in the syntax tree produced by the parser and the symbol table, and then check that if it corresponds to the languages semantics. The most important part of the semantic analysis is usually type checking. Depending on the language the type check can be strict or less strict. A language that guarantees that a complied/type-checked program will run without any type errors is a "strongly" typed language. The opposite of a "strongly" typed language is a "weakly" typed language, where type errors are allowed and can potentially crash the program. An additional classification for type checking is "dynamic" and "static" type checking, which is specified based on when the type checking is executed. If the type checking is done at compile time (during compilation) it is "statically" typed. If type checking is done at run-time it is a "dynamically" typed language. Table 2.1 shows different languages classified by the classification mentioned, the table is based on information from [35].

Type check is an analysis of variables and values in statements, to check if they corresponds to the semantic rules in the language. For instance, when a variable `int a` is assigned through `a = b + 10`; `b`'s type also have to be of the type `int`. It could be that `b` is a string, and then the addition would not make any sense. Thus a type check has to performed. Other typical semantic rules would be; a break statement is enclosed in a `while`, `switch` or an `if - else` statement.

For the examples presented earlier in figure 2.2 and figure 2.1, the type check

	Strong typing	Weak typing
Static typing	Pascal, Java, Modula-3 and ML	C and C++
Dynamic typing	Scheme, Postscript and Smalltalk	assembly code

Table 2.1: Type checking classification

would depend on the type of the variable in `<id, 1>`. In most languages they would have to be of the same type, if not the semantic analyzer would report an error or warning.

2.1.5 The Intermediate Code Generator

The intermediate code generator is the piece of the complier that turns the syntax tree into code. Most modern compilers generates C code at this stage, and uses known C compilers to generate machine code. C compilers provide a stable and well known platform. C compilers are available for several architectures, and it would require a great deal of work to generate efficient machine code without a finished back end.

How advance the intermediate code generator is depends on the difference between the language used when generating code and the language of the source code. For some languages the level of abstraction between the languages can be high. The code generator can than be complex as a lot of "work" has to be done to transform the code. This is especially true if the defined language and the language of the generated code is from two different language families, such as functional and imperative (more details on this later).

Of the syntax trees in figure 2.2 and figure 2.1 we can generate one operation for each operation in the syntax tree, and assigned the result to a temporary variable. For the syntax tree in figure 2.2 we would get the code in listing 2.10

```
1 t1 = id3 * id2
  t2 = id1 - t1
3 id1 = t2
```

Listing 2.10: Intermediate generated code of the syntax tree in figure 2.1

This form is called three-address code, where each operation corresponds to one statement.

2.1.6 Optimization

Generated code by the intermediate code generator is usually non-optimal, and often a bit clumsy. Hence, many compilers has a code optimizers that optimizes the code generated by the intermediate code generator, before it is "sent" to the back end. The optimizer analyzes the generated code, and applies optimization techniques to improve the code, either architecture specific optimization or general techniques. For instance, it can remove sequences of code that is unnecessary, or replaces it with code that would run faster, or take up less memory, all depending on what the goal of the optimization is.

One of the most common optimization techniques is dead code elimination. The first step of dead code elimination is to analyse the flow of data in the execution path. Based on the result from the analysis unused variables and "dead" (can't be run) execution branches can be found. An example of a "dead" execution path in its simplest form is an `if` where the branch condition always is false. The `if` block is then never executed and can be removed without having any effect on the result of the program. A dead variable is a variable that does not effect the program in any way.

For instance, the code sequence in listing 2.11 has two dead variables. `a` is assigned twice in line 1 and in line 4. The first assignment have no affect as it is overwritten before it is used. The same is true for `b`, which is assigned in line 2 and in line 3. We can then remove the first assignment of both `a` and `b`. The source code can then be reduced to the form in listing 2.12.

```
1 a = c + 3
  b = 2
3 b = 2 * c
  a = 3
5 d = a + b
```

Listing 2.11: Dead variable it does not affect the program

```
1 b = 2
  a = 3
3 d = a + b
```

Listing 2.12: Dead variable it does not affect the program

Most compilers does this optimization as a part of the type checking and gives a warning when a unused variable is declared, or if an `if` statement

is always false. There are several other optimization techniques that are much more complex than data-flow analysis. In section 2.3.5 we will briefly look at some optimization techniques that tries to identify code that can be parallelized.

2.1.7 Run-time Environment

A run-time environments is created together with the compiler and is the environment where the compiled code is executed. It handles memory management, input/output, access of variables, communication etc. Exactly what mechanisms a run-time has depends on the features of the compiled language. We will take a look at the runtime for languages that focuses on concurrency in section 2.4

Summary

In this section we have presented the structure of a compiler front end. It consists of four parts :

- lexical analysis reads the source, identifies tokens, and generates a token stream
- syntax analyzer reads the token stream,"sorts" the program, and generates a syntax tree
- semantic analyzer reads the syntax tree and performs checks, such as type checking on the tree
- intermediate code generator reads the syntax tree and generates code based on it

We also described how language grammar can specified on Backus-Naur Form, and how this form is read.

2.2 Compiler Generator Tools

Creating a hand written compiler is a lot of work, big parts of the complier is also quite static. Instead of rewriting the standard structure (presented earlier in this chapter) the framework of a compiler can be auto generated by using compiler generator tools. In this section we will take a look at such tools.

lex and yacc

Lex is a tool for building a lexical analyzer, and yacc is a tool for creating a parser (This section is based on the book lex & yacc [20]). With lex we can specify a lexical definitions which are mapped to a specified block of code. There exists different lex tools for different language, so the code block can be either Haskell (Alex), Java (JLex) or C (flex). The lexical specification are defined with a regular expression. In listing 2.13 we have written a lex specification that accepts numbers.

```
1 [0-9+] {yylval = atoi(yytext); return NUMBER;}
  [\t] ; /*ignore whitespace*/
3 \n return 0; /*logical EOF*/
  . return yytext[0];
```

Listing 2.13: Lexical specification for lex for numbers and varibales used in listing2.14

The first regular expression `[0-9+]` accepts one or more digit in the range $0-9$. It then converts the character string accepted (`yytext`) to an integer, stores it in `yylval`, and returns the token for digits `NUMBER`.

In Yacc the syntactical rules for the language is defined. The syntactical specifications uses tokens that corresponds to the tokens used in the lexical specifications. The style used in yacc looks a lot like CFS (see section 2.1.3). As with lex, each rule is linked to a block of code. In listing 2.14 we have written simple yacc grammar supporting summation and multiplication, such as the CFS grammar presented in listing 2.7. As we can see, it uses the tokens specified used in the lex specification earlier. (Both examples (2.13 & 2.14) is based on examples in [20]).

```
%token NUMBER
%left '+'
%left '*'
%%

statement: expression {prinf("Sum %d\n", $1)};
expression: expression '+' expression { $$ = $1 + $3;
   }
          expression '*' expression { $$ = $1 * $2;
  | NUMBER                          { $$ = $1; }
;
```

Listing 2.14: Yacc grammar for a simple calculator

In the beginning of the yacc specification we specify the tokens, followed by definition of associativity, and precedence level of `'+'` and `"*"`. These definitions are necessary to avoid ambiguity, as described in section 2.1.3. Yacc has its own syntax to refer to tokens in the token stream; `$$` refers to the value on the left hand side of the `:` (nonterminal), `$1` the first argument on the right hand side, `$2` is the second argument etc. The grammar in listing 2.14 specifies that a statement consists of an expression, and when this reduction is done it should print out the value of `$1`. `expression` is specified to either consist of two `expressions` which is added, or just a `NUMBER`.

The lexer and parsers generated will analyze and calculate a simple summation. In listing 2.15 we demonstrate how the compiler reduces a simple summation of $5 + 6 + 7$.

```
--- CHARACTER INPUT -----
"5 + 6 + 7"

------ LEXING ------
NUMBER "+ 6 + 7"
<NUMBER, 5> '+' <NUMBER, 6> "+ 7"
<NUMBER, 5> '+' <NUMBER, 6> '+' "7"
<NUMBER, 5> '+' <NUMBER, 6> '+' <NUMBER, 7>

----- PARSING ------
<NUMBER, 5>                    | shift NUMBER
```

```
12 expression                    | reduce expression ->
     NUMBER
   expression '+'                | shift '+'
14 expression '+' <NUMBER,6      | shift NUMBER
   expression '+' expression     | reduce expression ->
     NUMBER
16 expression                    | reduce expression ->
     expression '+' expression
   expression '+'                | shift '+'
18 expression '+' <NUMBER, 7>    | reduce expression ->
     NUMBER
   expression '+' expression     | reduce expression ->
     expression '+' expression
20 expression                    | reduce statement ->
     expression
   statement
```

Listing 2.15: Lexing and parsing of a simple summation

The BNF Converter

The BNF converter is a tool that generates the front-end of a complier, based on a specified grammar like lex and yacc. BNFC uses LBNF (labeled BNF) which is based on CFS or BNF, presented in section 2.1.3, with some additional rules, such as each rule has to be given an unique label. These labels are used when generating the syntax tree, and can also be used by the intermediate code generator to identify nonterminals and productions. BNFC can generate a fronted written in C/C++, Java or Haskell.

In LBNF both lexical and grammatical specification has to be defined. The lexical specification is written with regular expression similar to lex. There are, however, several predefined tokens which can be used when specifying the grammar such as; `Integer`, `Double`, `Char`, `String` and `Ident`. They are all converted to the corresponding type in the language converted to, except `Ident` where a data type has to be defined (see [24] for more details). In contrast to yacc where precedence and associativity is stated separately and in the beginning of the grammar (see [20] p.61) precedence is expressed through index variants in LBNF. Such as in listing 2.16, where multiplication has a higher precedence than summation.

```
1 EInt.   Exp3 ::= Integer;
  EMult.  Exp2 ::= Exp2 "*" Exp3;
3 EPlus.  Exp  ::= Exp "+" Exp2;
  coercions Exp 3;
```

Listing 2.16: LBNF precedence and associativity defined for multiplication and summation

In the grammar each production is given a label and a number. The number is used in the definition of the rest of the expression to describe precedence. In this example a `EMult` has to have a `Exp3` on the left hand side, which is an `EInt`. A summation `EPlus`, has to have a `Exp2` on its right hand side, which is a `EMult`. This is the same way to describe precedence as we saw earlier in listing 2.9. The keyword `coercions` binds all the indexed `Exp` together, and allows translation between the precedence levels.

Haskell

Haskell is a general purpose lazy functional programming language. It is used for creating all types of applications, but in this section we will focus on using haskell to write a compiler (This section is based [29]).

Haskell's specification was published in 1990 by a committee of researchers. It is one of the few functional languages from the nineties that survived, and is still maintain today [29].

Functional languages is based on functions that take in values as input, and gives out values as output. Instead of statements that modifies data, as "conventional" imperative languages. A haskell program is a collection of functions that are linked together to give an specific output. For a programmer used to "conventional" programming languages this way to design languages can be a bit abstract.

Haskell is also additionally a lazy language, which means that it does not execute anything that its not necessary. From a programmers perspective this means an infinite behavior can be used without it ever being executing. That might be a bit abstract. A typical example (by David Turner [36]) for this is an list containing prime numbers. The list in infinite, but a calculation never happens until the program "asks" for a number in the list.

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p
      /= 0]
```

Listing 2.17: The classic Turner's sieve [36]. A list containing every possible prime number

The `primes` function in listing 2.17 does not begin to calculate the list before it is called. The function is recursive and hence when looking for the 100th prime number it has to calculates the 99th first. `primes` calls the function `sieve` with a list from $2 \rightarrow \infty$. The function `seieve` takes in the list and divides it into two parts, the first element and the remaining piece of the list. It then builds a list of the first element, and the output of a recursive call of itself. The input to the recursive call is the list generated by the list comprehension `[x | x <- xs, x 'mod' p /= 0]`, `x` is the last output, it consists of elements from the list `xs` (`x <- xs`) that has a remainder when divided by the first element in the list (`x 'mod' p /= 0`). The function will recursively run through the whole list. What we are left with is a list of all the prime numbers.

As we have seen earlier in this section a compiler describes the transition from one level of abstraction to another. From a language written with a predefined grammar, to machine code that runs on a specified architecture. Lets take a look at how a functional language would describe such a transition:

In a functional language the programmer writes function taking in a input, and giving out an output. The same input always gives the same output. If the input to the function is the assignment of a variable, the output is generated code for the assignment. In a complier the transition from assignment to generated code would not only be one function, but a number of functions: first the lexical analyzer, the parser, the code generator etc. The complete set of functions can been seen on as one function describing the complete translation from source code to generated code. With a functional language such as Haskell, such translations can be described quite naturally. A combination of the tools mentioned earlier and Haskell is a powerful combination for creating compilers. For additional details on Haskell and building Haskell compilers we recommend "Real World Haskell" [29], which this section is based on.

Summary

In this section we presented some tools for creating a complier, lex & yacc and BNFC. They auto generate lexers and parsers. In the last section Haskell was briefly presented, and a short introduction to how it can be used to build a compiler was presented.

2.3 Parallelization

In the previous section the structure of a compiler was presented. The compiler can generate code depending on the architecture of the target machine. However, it can not always generate concurrent code that can utilize the multiprocessor, and it is the programmer that essentially has to write so it can run efficiently.

Today most mainstream CPUs is multicore, which is one microchip with several cores on. The introduction of these processors in to the mainstream market marks a paradigm shift in hardware and software development. Suddenly problems and techniques that only where relevant for systems running server farms and super computers, where relevant for programmers working on games, web-browsers mainstream application. As a result new languages and run-time systems has been introduced (see 2.4). In this section we will give an overview of some well known parallelization techniques, and parallelization patterns, which enables programmers and compilers to harness the power of multiprocessor architectures. The patterns and definitions in this section are based on patterns presented in "Patterns for parallel programming" [25].

2.3.1 Exploitable Concurrency

The first step to writing parallel code is to identify patterns and tasks, which has the potential to be done concurrently. A problem has exploitable concurrency if it can be divided into subproblems, which can be executed simultaneously. To be able to run simultaneously the subproblems has to be isolated problem that does not interfere with any other of the subproblems. A typical example of exploitable concurrency are for/while loops where an section of code are done repetitively. In many cases the iteration in the loop does not dependent on each other, hence it does not matter if the 100th iteration is done before the first, or at the same time as the first. Identifying such exploitable concurrency requires a "trained" eye, so we will not go into detail on techniques on how to identify exploitable concurrency, but present design patterns which can utilizes it.

2.3.2 Amdahl's and Gustafson's Law

When talking about parallelism it it natural to describe the limitations of parallelization. There are two laws formulated which describes the limita-

tions. The best known is probably Amdahl's law.

Amdahl's law state that; the performance improvement gained by parallelization, is limited by the amount of sequential code. For instance, if half of the code can be parallelized and distributed over n processors, and the other half has to be run sequentially. The speedup will be limited by the sequential code, no matter how many processors the parallel code can run on the sequential part has will not run any faster.

$$\begin{aligned} \text{Speedup} &= \frac{s+p}{s+p/N} \\ &= \frac{1}{s+p/N} \end{aligned}$$

Where s is the sequential part of the program, p the parallel part of the program, and N the number of processors in the system. So when N increases and $p/N \to 0$ the speedup goes towards $1/s$ [26].

Amdahl's law is often criticized for being pessimistic, due this limiting effect. Amdahl made two assumptions when formulating his law; the sequential part and the parallel part consists of a constant number of operations independent of the number of processors, and the input size is fixed [26]. Later Gustafson observed that these assumptions where inappropriate for massive parallel systems. He observed that the problem size could grow, and that the sequential fraction of the program was dependent on the number of processors in the system. He defined a new law, Gustafson's law:

$$\text{Scaled speedup} = N + (1 - N) * s'$$

Where s' is the serial fraction of the time spent on a parallel system [26].

This law is much less pessimistic, as the scaled speedup is not limited by the sequential piece of the code. It has later been proved by Shi [32] that Gustafson's and Amdahl's law are actually the same law, with different definitions of the sequential part s. These laws can show the limitation in performance gain for parallelized systems, and is hence important to consider before applying a parallelization pattern.

2.3.3 Concurrency in a Language vs Concurrency in a OS

Although concurrency in programming languages and operating system is essential the same, there are some important differences.

For an operating system it is important to keep the processor busy, and it is the schedulers task to feed it with jobs to keep it busy (see section 2.5). If a task is waiting on reading from I/O it is usually preempted, and a new process is run. These processes has to be run in a safe environment, so if one process crashes the rest of the processes running on the OS is not affect.

Handling concurrency in a programming language is a bit different. A programmer writing concurrent programs, want to minimize the code run sequentially, and try to run as much code in parallel as possible. They are not concerned with the other programs running in the OS. A programming language can not guarantee that the programmer does not write unsafe code. What it can do is to offer safe mechanisms and tools, that makes it easier write safe and parallel code. However, it is in the end the programmers responsibility to write "good" code.

In essence a programming language puts the power in the hands of the programmer, while an OS tries to keep everything safe. To keep everything safe requires a lot of checks and clean up mechanisms, which introduces overhead in the OS (see [25] for more details).

2.3.4 Parallelization Patterns

With mainstream programming languages such as C, Java, python the challenges is to identify code, which can be parallelized. With concurrent programming languages, which strives to be parallel, concurrency and parallelization occurs "naturally" through language mechanisms. We will take a closer look at some of these languages is section 2.4. In this section we will take a look at some of the most used design patterns enabling programmers to write concurrent code (all of based on the pattern presented in [25]).

Single Program, Multiple Data (SPMD)

SPMD distributes one copy of the source code to each unit. In a cloud/server farm this would mean a machine/unit, or on multiprocessor system one per CPU. The code is executed independently on each unit with small variation, often using their ID to differentiate their behavior. This can be done through branching condition, where the ID triggers different paths through the source code. The stages of SPMD program are usually:

- Initialize; load program, usually also establishing connection between the units

- Obtain unique identifier; usually a unique thread id, which can later be used to determine the execution path of the program.
- Run the program; each unit runs the program using the ID to differentiate its behavior.
- Distribute data; distribute the data that the source programs takes in out to the units, and retrieve calculated data later.
- Finalize, clean up and shut down the programs on the different units.

This is a technique that easily can be transferred to very large systems such as super computers, where its necessary to have highly scalable structures. The type of programs running on such computers are often independent intensive computations, and can be distributed among the units in the computer.

SPMD is often criticized for its complexity, especially when it comes to defining behavior based on a unique id. It can make the source code complex and it can be difficult to identify the sequential algorithm parallelized.

Master/Worker

The master worker principle is based on delegating tasks. The master creates tasks and distribute these among a set of workers. In many cases there are functions in the program that are time consuming, and not necessarily needs to be done in a particular order. With a master worker pattern such a task can be identified, and run in the background.

Let's say we get an sample of data as input. It consist of an array with different measurements, which all have to be analyzed. The analysis is independent, and is only based on a single measurement. So each of the analysis can then be run as an independent tasks asynchronously. The master then creates a task for each analysis, and queues them. The worker created by the master, executes the queued tasks, and when the queue is empty it notifies the master, which wakes up and continues (figure 2.4 illustrates the master/worker behavior).

This pattern enables the jobs to be run concurrently and distributed among the processors in the system, instead of the master running all the analysis on one processor. There are several different languages and run-time system, which uses this pattern in their implementation, some examples of this we will come back to in section 2.4.

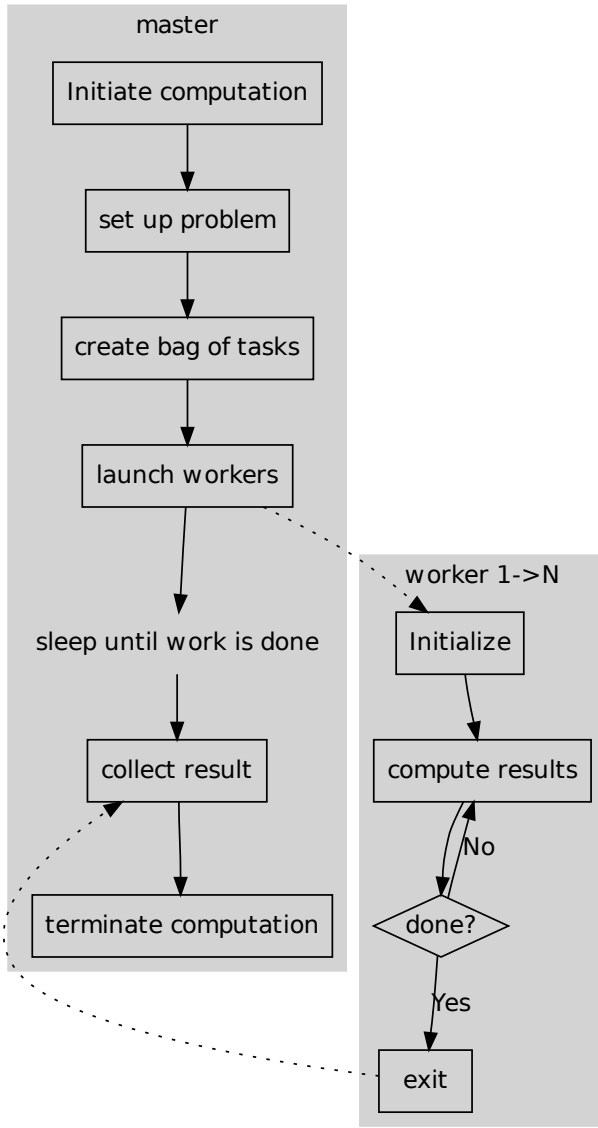

Figure 2.4: Master worker pattern

The master worker pattern has a high grade of scalability, as long as the number of tasks exceeds the number of workers. However, a known bottleneck to this pattern is a shared data between the workers. Such as the shared task queue, which may become a bottleneck if the number of workers become large. A solution to prevent such problems is to have several queues, and distribute the tasks among these queues, which is a well-known technique used in OS schedulers. This is something we will come back to in the section on multiprocessor schedulers (section 2.5). Other shared data may also become bottlenecks. Many of the systems based on the master/worker pattern usually also support some sort direct communication between the workers to prevent communication through shared data, and hence prevent bottlenecks and consistency problems due to the shared data (see section 2.3.7).

Loop Parallelism

Most of the execution time of a program is usually spent in a loop, hence also an place of interest to" unleash" exploitable concurrency. In many cases the iterations of the loops are independent from each other, and could be run on different processors. The first step when using loop parallelism is to identify the loops in the sequential program. Where most of the execution time is spent. Such an analysis can be done with a profiling tool (times the loops in the program), or through using debugger tools and experience. If each iteration is independent of each other the pattern can be applied. If not check if the dependent part of the code can be moved out of the loops body. In many cases such read/write dependencies can be moved to before the loop. It is also necessary to evaluate if the sequential work done in each iteration is intensive enough to compensate for the overhead of the loop parallelism pattern.

When a suitable loop is identified it can be divided into parts, and run concurrently on the processors in the system. For instance, lets say we have a for loop iterating from $0 \rightarrow 100$ where loop parallelism can be applied. It is run on a system with four processor, each processor runs 25 of the iterations concurrently.

Memory locality can affect the performance of loops which are parallelized negatively. Due to that most modern architectures has its memory structured as a hierarchy, the access time for each of the processors are not the same in the shared memory. In other words, the one that has the "copy" of shared data (such as an array where each is manipulated pieces of it) closest

has a faster access time than the others. Due to this effect systems using the loop parallelism pattern gives each unit a copy of the shared memory in the innermost loop.

For instance, let's say that the loops manipulates an array. The array is the located on a chunk of memory, which all the concurrent loops has to access. The memory is then moved around between each processors cache, and that the caches is constantly invalidated. To prevent this, process can use a temporary array, and copy it to the original array when finishing the loop, which is an easy way to avoid most of the negative effect due to shared memory.

Fork/Join

Another and commonly used parallelization pattern is the fork/join pattern, which is based on dynamic concurrency. Where conditions determined at run-time, decides how much exploitable concurrency there is. These conditions can be branch conditions in simple if/else branches, where each branch can be done concurrently. Or an array with variable length, which determines how many threads can work on the array concurrently. Listing 2.18 is an example of a program that utilizes an if/else branch to do some work concurrently.

It forks (creates a clone of the thread) the main thread. The child thread created does merge sort on the lower half of the array, while the main thread does merge sort on the upper half. When they are finished with their sorting the threads are then joined again.

```
fork();
if(child process) {
    sorts the lower half of the array;
}
else {
    sorts the upper half of the array;
}

join();
```

Listing 2.18: Pseudo code sorting an array

One of the pitfalls programmers fall in when using the fork/join pattern is the running costs of the OS mechanisms used to implement a fork/join

pattern. If the work done in each branch of the fork is to small compared to the job introduces trough creating and joining, then the pattern may not increase efficiency at all. To avoid the overhead from the OS, some run-time system provides a pool of worker OS-thread that can perform the forked tasks. The threads in the pool runs queued processes created by in a fork/join pattern, and returns to the pool when done executing the process. This usage of thread pools can be seen on as a combination of the master/worker pattern and fork/join. In many of the run-time systems built to enable concurrency based on such a thread pool (for more details see section 2.4).

2.3.5 Automatic Parallelization

Automatic parallelization is a term used to describe compiler optimization, which identifies exploitable concurrency in sequential code, and generates concurrent code. First an analysis is done at compile time, and then parallelized code is generated. There are different types of analysis. It can either be based on language specific keywords, which tells the compiler that this section is independent and can be run in parallel. Or analysis done at compile time, which maps dependencies in the code, and uses this to identify exploitable concurrency.

Most of the compilers that support automatic parallelization, generates code based on the patterns presented in the previous section. Later in this chapter we will present some of these languages and their run-time systems.

The most used pattern of the one presented is probably loop parallelism, mainly because loops are common and is where a program spend most of its run-time. In many cases iterations is independent from each other, or it can easily be modified to be independent. However, automatic parallelization is complex and time consuming process. Additionally, there is no guarantee that the run-time of the program is improved. In many cases this is due to the overhead introduced through parallel patterns, as mentioned earlier.

We will now take a brief look at how exploitable code can be identified, and how the code generated can look like. This section is based on the “dragon book’[5], where more detailed exploration of automatic parallelization can be found.

```
1 for(i = 0; i < n; i++) {
      Z[i] = X[i] - Y[i];
3     Z[i] = Z[i] * Z[i];
  }
```

Listing 2.19: A for loop with exploitable concurrency (example 11.1 in [5])

In listing 2.19 a for loop doing array manipulation is presented. The block in the for loop consists of two simple assignment, where one depend on the other. However, there are no dependency between the iterations of the for loop.

By using dependency analysis techniques such as data-flow analysis (see section 2.1.6) the exploitable concurrency in the for loop can be identified. The loop can then be parallelized easily using SPMD and loop parallelization. First lets take a look at the loop parallelized by applying SPMD:

```
  b = ceil(n/M);
2 for ( i = b*p; i < min(n, b*(p+1)); i++ ) {
      Z[i] = X[i] - Y[i];
4     Z[i] = Z[i] * Z[i];
  }
```

Listing 2.20: for loop parallelized with SPMD

In line `1` in listing 2.20 `ceil` determines the number of iteration each instance of the SPMD program will do. It is then used together with the process id `p` to define where the loop starts and the upper limit. To prevent the "last" iteration from going over `n`, the minimum of `n` and the calculated limit (based on process id and `b`) is found. The for loop can then be run separately on multiple processors, and when they are all done the array can be "glued" together.

It is not necessary to divide the for loop using process id dependent code. It can also be parallelized using the loop parallelization pattern, the result will be the same and the code is quite similar.

```
1 void independentCalc(int i) {
      Z[i] = X[i] - Y[i];
3     Z[i] = Z[i] * Z[i];
  }
5
  for( i = 0; i < n; i++) {
7     run_par(&indepentdentCalc, i);
  }
```

Listing 2.21: A for loop parallelized with loop parallelization

In listing 2.21 a function is created with the body of the for loop. In the for loop a function from the run-time system is called with the generated and `i` as argument.

The function is then run in parallel in the background by the run-time system (see section 2.4 for more details on how such a run-time system are implemented). It generates `n` process that can be run concurrently. We could also modify the code to run parts of the for loop concurrently, instead of every single iteration. It would be a good idea, as the cost of generating a process is high, and the work done in each process is small.

Pure automatic parallelization is difficult, so most concurrent languages and run-time systems uses a combination of grammatical rules and libraries to help simplify parallelization.

2.3.6 Dependency Analysis

As mentioned in the last section, dependency analysis is an important tool when it comes to identifying exploitable concurrency. There are three types of data dependency [5]:

- True dependence, a write is followed by a read.
- Antidependence, a read is followed by write to the same location.
- Output dependence, two writes to the same location.

A true dependency is what we usually consider a dependency; first a write followed by a read that reads the written data. Antidependence is used to check if it is possible to change the order, if there is an antidependence

the read must always happen before the write, or else the behavior of the program might change. This is important to know for parallelization techniques that might move dependent code, in order to enable a parallelization pattern to be applied. Such techniques also has to be aware of output dependency, as changing the order of two writes can change the behavior of the program.

Data dependencies are especially important when it comes to parallelizing loops. If each iteration is not completely independent, it might be that a set are independent, and can be run in parallel.

```
float Z[100];
for(i = 0; i < 10; i++) {
    Z[i + 10] = Z[i];
}
```

Listing 2.22: A for loop which can be run parallel

As we can see in the for loop in listing 2.22 the iterations are dependent on each other. However, there are still exploitable concurrency which can be harnessed.

By using a theory called "affine transform theory" (see chapter 11.1.5 in [5] for details) the compiler can determine that this loop can be run in parallel. Let us look at an example where it can be used:

Due to the for loops upper limit of `i < 10` there are no variables that are dependent on each other. The reason for this is that the position of the assigned element in the array has an offset of ten on `i`, and hence none of the elements assigned is used again in a later iteration. But if the upper limit is changed to `i < 100` there are dependencies between iterations, and it is then necessary to map these. With `i < 100` as upper limit for the for loop every tenth iteration is dependent on each other. Iteration `0` has to run before iteration `10`, which again has to be run before iteration `20` etc. Affine transform theory can map such dependencies, which enables the compiler to try to take advantage of it.

2.3.7 Communication Between Concurrent Components

Communication between the components can divide into two different classifications:

- Shared state communication.

- Message passing communication.

(Defined in [2])

Most conventional programming languages supports shared state communication. A typical example of an implementation that uses shared state are processes that share variables, and by writing and reading the shared variables the processes can communicate with each other. Shared state communication is criticized for creating bottlenecks and difficult to keep safe. As multiple concurrent components can manipulate the shared state, it is important to make sure that only one writes at the time, so the data is kept consistent. There are many different patterns and techniques to create process safe data structures. In many cases such structures becomes bottlenecks in the system as the only allow one process to access it at the time.

Programming language, which support message passing communication, does usually not allow shared state communication. This means that the usage of shared variables is restricted, and in some cases not allowed at all (such as in occam [21]). Instead they support communication between the processes through messages, and there is usually grammar defined for sending and receiving messages.

In some languages such as in Occam and GO the programmer can define channels that messages can be sent through (see section 2.3.8for further detail). The processes then listens or sends over these channels. In other programming languages message passing is built in to the structure of a process. Such as is Erlang, where each process has a mailbox, which other processes can send messages to. When a new message arrives in its mailbox, the process is notified [2]. The process then has to copy the message to a variable if it wishes to use it further. Most run time systems that implement such a behavior copies the message data from the senders stack to the receivers, to prevent that there are any shared data among the processes. Then their is no need for mutable data structures, and hence less chance of creating data structure that becomes a bottleneck.

Message passing can be either asynchronous or synchronous. Synchronous communication is blocking, which means that the sender and receiver must execute the channel communication at the same time. Most programming languages built on message passing offers synchronous and asynchronous communication, through channels and buffered channels. The buffer holds the message sent until a receiver comes and picks it up. If the channel can't buffer messages, the communication is synchronous. With asynchronous

communication parallelism improves, as processes can't be blocked. There are, however, several issues with asynchronous messaging, as the sender never knows if it was actually received. To make sure critical messages comes through, a three way handshake protocol has to be implemented, which introduces extra communication. But with synchronous communication a process might be blocked while waiting on a non critical message [11].

2.3.8 Concurrency Oriented Programming Languages

As mentioned earlier in this section, there are several concurrent programming languages. That means that they are designed let the user easily write concurrent programs without too much hassle. Later in the report (section 2.4) the run-time systems that support the language features is presented. In this section we will present three concurrent languages, the new language GO, and two old timers Erlang and Occam.

Erlang

Erlang is a language created by the telecom company Ericsson in 1986, and a open source version was released in 1998. In many cases Erlang has set the standard for what a concurrent language should offer and focus on. There is later built several clones of Erlang, based on different languages, such as:

- Scala; running on the java platform
- Erlectricity; Ruby
- Retlang; C#
- Candygram; Python

Ericsson based their language on a combination of functional languages (Haskell) and existing concurrent languages (such as Ada, Chill). They designed a concurrent programming language, with asynchronous messaging, which where robust, and ran on a virtual machine [8]. Erlang's focus on concurrency and scalability has made it popular for distributed systems (running on several machines, and often with a varying number of machines depending on the load on the system).

They argument strongly for choosing message passing over shared state communication: channel communication limits the possibility of interference

between processes, and reduces the chance of operations having unwanted side-effects. It also help improve efficiency on a multiprocessor system. According to "Programming Erlang" [2] a multiprocessor program has to fill the following criteria to be effective:

- Create/use lots of processes
- Avoid side effects
- Avoid bottlenecks
- Send a small amount of message compared to the amount of computations

The first goal means that the program has to be fine grained, and only have a small pieces of sequential code. For instance, a program consisting of one heavy processes doing a lot sequential computations, and a large number of light processes "feeding" the sequential process. Will not be able to utilize a multiprocessor system, as most of the light processes will be done quickly, and just be waiting on the heavy process running on one processor. By creating small and intensive processes of about the same size, it easier to distribute the computational load over all the processors in the system.

Side effects can occur through shared state communication. Erlang does not allow any shared data, all communication has to be performed through messages to other processes. Which also reduces the chance of having bottlenecks in the system, as each process has its own separate mailbox where it keeps its messages. The language encourages programmers to note create shard structures which can be become bottlenecks.

In most languages and systems based on message passing, communication is fast but expensive compared to reading/writing to shared data. That means that system that are message intensive can be slow. But by creating processes that have a lot of computation compared to the number of message helps reduce the load on message passing mechanisms. This problem is closely connected to the first one discussed in this section. By focusing on the parallelization techniques presented in section 2.3 such problems can be avoided.

Ericsson designed Erlang for their own products such as tele switcher. It was important for them that these systems where robust. For instance, if an error occurs during a phone call, it is important that not the whole tele systems crashes and has to be rebooted. For a large company using an Ericsson

system for there internal phone network, a system where the whole system crashes from time to time would be unacceptable. Therefor, in Erlang a process can register as "dependent" on each other. If one process crashes the "dependent" processes are notified, and can do correct measurements to prevent further errors.

In the following example (listing 2.23 and listing 2.24) communication example from [2] is presented. The server (listing 2.23) waits for an incoming message from a client that is asking for the area of an object. It then responds with the calculated area of the object. The client (listing 2.24) starts by spawning the server, which then runs in parallel with the client. It then asks for the area of a square with `10` sides. It sends its own PID in the message as the return address. The PID is retrieved by the function `self()`.

```
-module(area)
-export([loop/1])

loop(Tot) ->
    receive
        {Pid, {square, X}} ->
            Pid ! X*X;
            loop(Tot + X*X);
        {Pid, {rectangle, [X, Y]}} ->
            Pid ! X*Y,
            loop(Tot + X*Y);
        {Pid, areas} ->
            Pid ! Tot,
            loop(Tot)
    end
```

Listing 2.23: Erlang server in a client server example. The client asks the server for the area of a square, rectangle etc (presented in [2]).

```
Pid = spawn(fun() -> area:loop(0) end),
Pid ! {self(), {square, 10}},
receive
    Area ->
        ...
end
```

Listing 2.24: Erlang client in a client server example. The client asks the server for the area of a square (presented in [2]).

occam

Appeared in 1983, and was originally created by IMNOS to run on their transputer microprocessor. It has later been adapted to run on conventional microprocessors, and there is currently several different "new" compilers for occam [21]. occam is based on message passing concurrency through CSP (communicating sequential process), which is a language for formally describing interaction patterns in a concurrent system. Due to occam's close relation to CSP occam compilers generates code that runs on the CCSP run-time system. The CCSP systems can also be used as an C library, to implement CSP like features in C (more details on this is 2.4).

occam is very different then conventional languages, such as Java and C. For instance, shared variables are not allowed. Data can only by shared through messages. Unlike Erlang where each process has a mailbox, occam has its own communication type `CHAN`(channel). A channel can either by synchronous, or asynchronous. This is done by declaring if a channel is buffered or not. A channel is also restricted to send one type of messages, like only `INT` variables. occam's strict rules when it comes to shared data helps programs achieve the criteria for an effective multiprocessor program, avoid side effects and avoid bottlenecks. As with Erlang this is achieved through message passing concurrency.

occam also have some unique features, such as the `PAR` and `SEQ` keyword. All statements has to be put in a statement block with a order stated; `SEQ` the statements are run sequentially, as in an "ordinary" language, `PAR` the statements are run concurrently. By running statements concurrently creates a fine grained parallelism, which can easily be distributed. If we look at the last two criteria for an effective multiprocessor program: create/use lots of processes, and send a small of messages compared to the amount of computations. The fine grained concurrency helps meet the first criteria,

as many processes are created as a consequence of the language. However, occam's fine grained concurrency may not help to meet the last criteria. The cost of message passing is, however, low compared to other languages supporting message passing, which is something we will come back to later in the report (chapter 6).

These are only a couple of the unique features occam support, for more details on occam please see [21]

A typical of occam design involves a client and a server, communication over a channel. In the example in listing 2.25 there are two procedures, which is run in parallel. The client sends the number 2 to the server, which receives it.

```
PROC Client(CHAN OF INT c)
  INT x:
  SEQ
    ch ! 2
:

PROC Server (CHAN OF INT ch)
  INT x:
  SEQ
    ch ? x
:

PROC Main()
  CHAN OF INT s:
  PAR
    Client(s)
    Server(s)
:
```

Listing 2.25: Occam client server example. Main starts two the client and server as two processes. They communicate over the channel s, client sending 2 over the channel to the server.

The Go Programming Language

GO was launched in October of 2009, and was created by Google. Its main target was application running on their massive server farms. There

goal was to create a lightweight fast language, with good support for parallelism. Google felt that none of the existing languages offered the support they where looking for. By picking the best from different languages they defined a language that support CSP channel communication, light weight asynchronous parallel processes, and fast compilation [30]. This section will be a short introduction to the language features, and in section 2.4.1 how the run-time system works will be presented.

As Erlang, GO is based a message passing concurrency, and offers both synchronous and asynchronous communication. This is done through channels, much like occam's channel. They can either be buffered (asynchronous) or unbuffered (synchronous). It does not have any parallelization keyword, and is neither as restricted as Erlang and Occam. However, functions can be run concurrently through launching them with the `Go` keyword. They are then run independently of the processes launching it (see [30] for further details). Except from these two features, GO looks remarkable like any other programming language. It utilities its strong run-time environment, which enables light concurrent processes to be run efficiently. GO programs helps meet the criteria for multiprocessor programs, through message passing and light concurrent processes.

```
1 c := make(chan int);
  func wrapper(a int, c chan int) {
3     result := longCalculation(a);
      c <- result; \\sends the result over the channel
5 }

7 go wrapper(17, c); \\runs the wrapper function in the
     background

9 //do something for a while; then ...
  x := <- c \\reads the result form the channel
```

Listing 2.26: Go routine using a wrapper function to perform a heavy calculation which sends the result over a channel (presented in [30])

Summary

In this section has focused on concurrency in programming languages, and how exploitable concurrency in a program can decrease of a program on a

multiprocessor system. A few parallel programming patterns was presented, along with a brief introduction to how compilers can automatically generate simple parallelization of the code.

In the last section three concurrent programming languages was presented, Erlang, Occam and GO. We will in the next section present the design of their run-time systems.

2.4 Parallel Run-Time Environments

In this section we will present how processes are created and run in different run-time systems. Some of them are run-time systems that supports a compiler such as Erlang, GO and Occam. Or through an API such as JIBU and CCSP. Most of this section is based the run-times source code, as all of the systems discussed are open-source, and there are few or no article published on their architecture. We also got some help from the communities, for instance, GOs mailing list. We will not go into detail on how the systems are implemented, but give an overview of how processes are created and run, and how communication between them is made possible.

2.4.1 GO's Run-Time System

The GO run-time system is built on the master/worker pattern (see section 2.3.4). The run-time has a set of worker threads, which is defined defined by the architectures it runs on. There should be one worker for each processors in system. When a function is called with the GO keyword, a process is created and queued. The worker threads then proceeds to execute the queued processes (see function `scheduler` line 455 in `src/pkg/runtime/proc.c` available at [9]).

The worker threads operates on a scheduler, which is non-preemptive. This means that the GO processes will run until they voluntary yield. In GO this happens on blocking systems calls such as a reading or writing to a channel, and they can also be made to yield explicitly. GO has focused on light weight process. For instance, by keeping information and process switching at a minimum. This is done through having simple data structure containing information about the process, which the worker thread can easily access.

The worker threads are OS-threads, and the reason for limiting the number of OS-threads is due to there overhead and expensive context switches. Instead it has its own processes with separate stacks and a scheduler.

Channel communication is either asynchronous through buffered channels, or synchronous through non buffered channels. For Go synchronous communication means that a process is blocked until the receiver/sender is ready on the other side with their message. Message passing is done through calling the processes send/receive function, which stores the message as an process argument in the processes stack, which then can be retrieved by processes (see function `chanrecv` and `chansend` at line 275 and 172 in

`src/pkg/runtime/chan.c` available at [9]). The authors of GO argues that CSP's high level interface enables simpler code and better scalability [10].

GO's architecture is similar to the other run-time systems discussed in this section. However, it distinguishes itself through its simplicity and straight forward approach in its implementation. GOs developer team is planning on doing changes to the run-time environment to increase performance and scalability.

2.4.2 Erlang's Run-Time System

In the same way as Java, Erlang code runs on a virtual machine, which means that Erlang code can be run on multiple platforms.

As multiprocessor architecture has become more and more popular Erlang has launched a virtual machine that supports multiprocessor systems. This version has one scheduler for each CPU. Currently they are sharing a run queue containing all the Erlang processes.

The scheduler is a round-robin scheduler operating with four priority levels max, high, normal and low. It is preemptive, as a processes is suspended if it waiting for a receive statement and their is no matching message in the message queue. And waken up as soon as possible when a new messages arrives [37]. It can also be interrupted if a message come through while executing.

The Erlang team in Ericsson has proposed some modification to the multi-processor virtual machine ([23]); partitioned scheduling with one run queue pr scheduler. Using work stealing and a distribution algorithm to keep the run queues balanced, this is due to that a shared run queues can become a bottleneck with more than four schedulers [23]. With partitioned run queues, this won't be a problem anymore.

Channel communication in Erlang is done through stack manipulation. Each processes has its own separate heap, stack and header. When a process sends a message to another process a message and the receivers id has to be specified. The virtual machine first checks that pid is valid, it then proceeds to copy the message onto the process' stack, and links it to the end of the processes message-list. The virtual machine does not allow there to be any pointer that points to other processes stacks. Hence if the message contains any pointers the data that it points to is also copied [7].

2.4.3 JIBU's Run-Time System

JIBU[1] is a run-time system that offers a concurrency library for Java, C++, .NET, and DELPHI. JIBU offers features like parallel execution , CSP channels, mailboxes etc. In this section we will focus on how processes is executed, and the architecture of the run-time environment.

JIBU's API functions generates processes, which is put in separate queues. Each of the OS-thread executes processes from multiple separate queue [3]. If one of the worker threads runs out of tasks, it will start executing tasks from one of the other worker-threads queue. This type of scheduler is known as a partitioned work stealing scheduler, as it has a partitioned run queue and the worker steals processes from each other. The schedulers is a non-preemptive scheduler, which means that a process is not interrupted when it first starts to execute (see function `TaskScheduler::run()` in `jibu_c_1.0.0/jibu_1.0.0/libjibu/task_scheduler.cpp` and function `ThreadScheduler::createTaskScheduler` in `../thread_scheduler.cpp` available at [4]).

What differentiate JIBU from the other run time systems presented, is its thread pool. It initially creates one OS-threads for each CPU, but more threads is created if their is demand for it. These threads behaves just like the worker threads created initially, executes task from the run-queues and return to a thread pool when there is nothing to do.

Jibu also has support for synchronous channel communication, as well as asynchronous communication like Erlang through processes mailboxes. These communication tools enables the user to move away from concurrency based on the shared state model in their chosen language.

2.4.4 occam's/CCSP's Run-Time System

Most of this section is based on the CCSP version presented in [27]. CCSP is a run-time environment based on the communication language presented by Hoare in [18]. It is a C run-time environment that supports channel communication and concurrent processes execution. And is used as the run-time environment for the occam compiler KROC (Kent Retargatable Occam Compiler). In contrast to the SPOC (Southhamptons portable occam complier) it does not use a compiler generated scheduler, but uses a

[1] At the time of writing, JIBU's website is no longer available. Hopefully it will be up again soon. If not, please contact the author to retrieve JIBU's source code. June 2010

predefined structure to create processes, and schedule them in a efficient manner.

To avoid stack manipulation through assembly instruction CCPS implements its own stack. All the C function that is generated by the Occam complier has to be wrapped by a stack function that uses the rules defined by CCSP. By avoiding stack manipulation CCSP reduces the cost of context switching. This is done through declaring explicit a list of input parameters, output parameters, and a list of corrupted registers. The run-time system can then reserve a specified amount of registers, and know which register that is corrupted after the function has executed. Enabling safer stacks. In addition to having its own stack structure, each function is split into parts where each part is labeled by a C label. The Complier then generates a table containing the labels, and can use these labels to jump back and forth in a function, rather than saving the state of the function on a stack.

It is necessary for CCSP to have fast context switches due to the large number of processes, which can be generated. For instance, Occam code like

```
PAR i = 0 FOR 10
```

where ten processes is created. In a traditional C program ten threads would be many. While Occam program can consist of hundred to thousands of processes. It is clearly necessary that the cost of a switch between processes is considerable lower in CCSP than for an OS-thread. CCSP hence has fast context switches, fast communication operations and spawns processes fast.

The developers of CCSP has focused on non locking mechanisms, and advance pointer and assembly manipulation, to optimize critical mechanisms. They also improve efficiency through better cache utilization, which is done through batching process together and run them in groups (see section 2.5.4 for more details). CCSP has a non-preemtive scheduler. The scheduler is based on that processes labels descheduling points, where it can safely be descheduled. The advantage is that is reduces the overhead introduces through a preemptive scheduler, and allows the complier to stay in control of the amount of constants that need to be stacked when descheduled.

CCSP uses a similar approach as the other run-time kernels discussed in this section. It has queues containing pending and waiting process, which are executed by worker threads. As the other run time systems, CCSP offers message passing. Messages can either be put on buffered and sent asynchronously, or copied synchronously. The synchronous communication

methods are blocking, and the process was to wait until the processes it is communicating with is ready. When they are both at the synchronization point, the message is copied from the sender to the receiver atomically. This solution is similar to how it is implemented in GO.

However, a lot of effort is put into making channel operation fasts. For instance, in GO a channel is a struct holding a list of senders/receivers, the interface of the element etc (for more details see `go/src/pkg/runtime/chan.c:40` [9]). While in CCSP a channel is represented by a single machine word. It uses the lowest bit to carry information, and holds a pointer to a process descriptor (see [31] for more details). This is just one piece that has been minimized, and shows the effort put in to optimizing the run-time system.

Summary

In this section we presented the design of four concurrent run-time system. Two languages run-time system, and two APIs which can be used in other languages (CCSP is used as run-time system for the Occam compiler KROC). Most of them use the master/worker design pattern, and has one worker pr. processor. This is due to large overhead on OS-threads. Most of them also have separate memory for processes in the run-time system. Communication is often done by copying data between the processes stacks.

2.5 Multiprocessors Scheduling

In this section we will present the architecture of two multiprocessor schedulers, some techniques to help improve efficiency, and lastly two of the best known real-time schedulers. This section is mainly based on our survey on real-time multiprocessor schedulers [17].

Scheduling for a multiprocessor system raises a set of new challenges, which uniprocessors scheduler can not meet. For a uniprocessor scheduler the only focus is what process to run. However, a multiprocessor scheduler does not only have to decide what process to run, it also have to utilize all of the processor and distribute processes among them. The two most common design for multiprocessor schedulers are: *partitioned* scheduler, and the *global* scheduler.

2.5.1 Partitioned Schedulers

In a partitioned scheduler there is more than one run queue. Tasks are distributed among these run queue when they arrive. Assignment is done by evaluating the tasks and the state of the processor the run queue belongs to. The goal is to distributed the tasks as fairly as possible.

The advantage with this approach is that it reduces multiprocessor scheduling to a set of uniprocessor scheduling problems. Enabling the usage of known optimal uniprocessor scheduling algorithms (can guarantee 100% utilization). The distribution of tasks is a NP-hard problem, bin-packing. Distribution of tasks can thus be an expensive operation. The extra cost due to distributions can decrease the performance of such schedulers [28].

Load balancing is an additional problem for partitioned schedulers. The number of tasks in each run queue has to be evenly distributed. When the system has ran for a while it might be that some of the run queues are empty, while others are full. So sometimes redistribution/load balancing might be necessary. It is a time consuming operation, and the system has to wait until the balancing is complete [28]. Global schedulers avoid this altogether by using a different approach.

2.5.2 Global Schedulers

Global schedulers has one global run queue. A task from the global run queue is assigned to a processor when it is idle, or when preemption occurs.

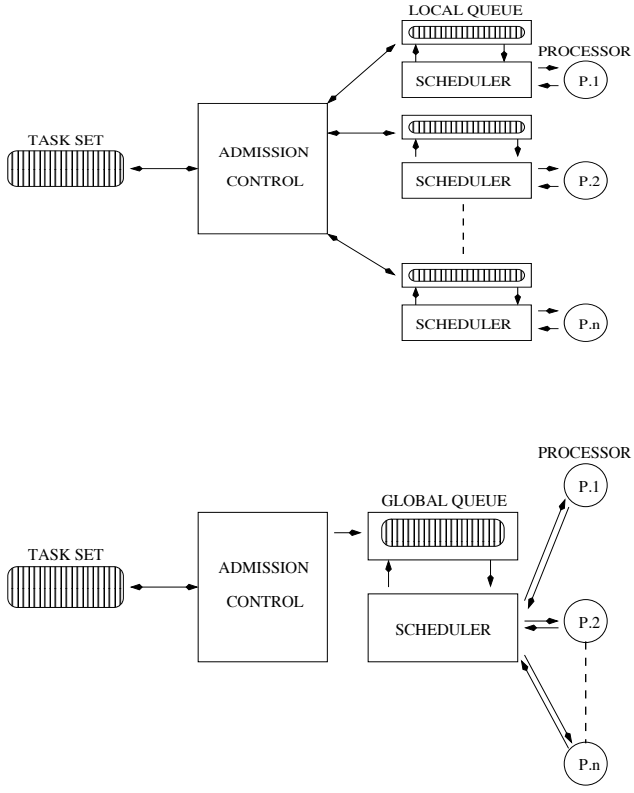

Figure 2.5: (upper) Partitioned scheme and (lower) Global scheme. Illustration from [28].

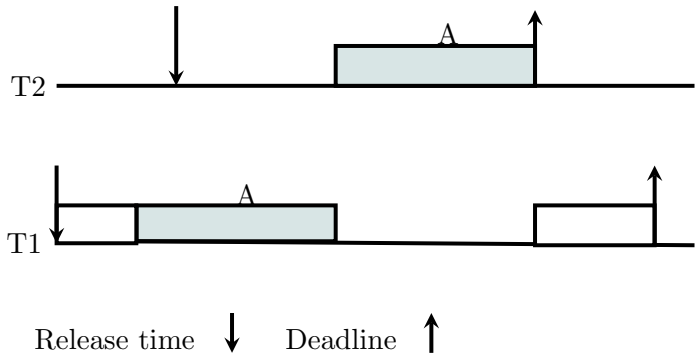

Figure 2.6: Process T_2 must wait until T_1 releases resource A, even though T_2 has higher priority

This scheme is much less complex then the partitioned scheduler. The simplicity of the design reduces the overhead compared to a partitioned scheme.

With a global scheduler there is no need for redistribution of tasks among the run queues, because each processor now only have one task at the time.

However, the scheduler now have to consider what all the processors are doing at once. Known optimal uni-processor schedulers are no longer suitable [28].

The global run queue is a shared resource between the processors. As mentioned is the section on parallelism (section 2.3) shared resources can become bottlenecks in parallelized systems. The global run queues has to guarantee exclusive access. This means that some times processors will be idle while waiting to access the run queue. To avoid this as much as possible complex analyzing mechanism can be used. However, they increase the overhead and complexity of the scheduler.

2.5.3 Interlocking Protocol

Interlocking protocols are techniques to prevent "important" tasks from being blocked by "non" important tasks holding a resource it requires.

Interlocking can occur when two tasks share the same resource.

If T_1 holds resource A and has a lower priority then T_2, but T_2 needs resource A to run, we have a interlocking problem. The high priority task T_2, must wait until a low priority task T_1 releases resource A (see figure 2.6). T_1

might even be preempted to allow higher priority tasks to run, and make T_2 wait even longer.

One of the best known interlocking protocols (priority inheritance protocol PIP [16]) uses priority inheritance to solve this problem, by giving T_1 the same priority as T_2 for as long it holds resource A. It enables T_1 to finish faster and release A so that T_2 can run. Guaranteeing that no task with a priority higher than T_1s initial priority can interrupt.

For a partitioned scheduler the interlocking problem is quite complex, due to that their is no global priority. The tasks are run individually on each processor, with its own run-queue. When a task needs a resource locked by a task running on a different processor, all the run-queues has to be evaluated. This is a complex and time consuming task. As interlocking happens fairly often it can effect the overall efficiency of the scheduler. Handling interlocking for global schedulers is easier, as the scheduler only have to analyze the tasks lying in the one run queue, and the domain of the priorities are global (More details on interlocking protocols can be found in [16]).

2.5.4 Batch Scheduling

Batch scheduling is a technique to improve efficiency for schedulers scheduling fine grained parallelism. With fine grained parallelism a large number of light processes are created, which does a small amount work. This puts more pressure on the scheduler and the run-time, because much more of the execution time is now spent in scheduler and in run-time system mechanisms. Such as context switches, process creation, process destruction and communication between the threads.

The idea behind batch scheduling is to better utilize cache. For instance, a process that run repeatedly on the same processor has less chance for cache miss, because less processes has run in between each time it executes. To exploit this the scheduler groups processes together based on what variables they share, and whom they communicate with. The groups are then run on separate processors.

Batching of the processes requires dependency analysis to be performed [31], to map the dependencies between the processes. Grouping based on dependencies are, however, not a requirement for this technique to be efficient. As long as there are processes which access the same memory repeatedly this technique can help improve efficiency. The run queues holds the batches

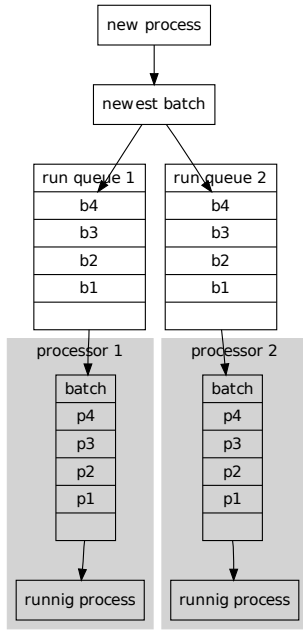

Figure 2.7: Partitioned batch scheduler with one run queue pr processor. Each run queue holds a set of batches.

and each batch is run until preemption or a task is blocked. Batch scheduling can be used together with ordinary scheduling algorithms, such as work stealing. The difference is that a "unit" is not an individual process but a batch, as illustrated in figure 2.7.

2.5.5 Real-Time Schedulers

In this section we will present two of the best known real-time schedulers. They schedule the processes based on a processes deadline, which is usually specified explicit or based on analysis. And tries to schedule the processes so they all meet their deadline. If it can guarantee that they all meet their deadline it is known as an optimal scheduler.

Rate Monotonic (RM)

Rate monotonic is a non-optimal uni-processor algorithm presented by Liu and Layland [22]. RM assumes that deadlines are equal to the period of the task, and assigns priority according to the length of the period. A task with a short period will get a high priority assigned. This is done under the assumption that there is static priorities (the task with the highest priority is always allowed to run), context switches times and other thread operations are free (not causing any additional delay).

RM schedulability test is defined as:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1) \tag{2.1}$$

Where C_i is the computation time, T_i is the release period, n is the number of task to be scheduled.

RM has been a popular uni-processor algorithm due to its simplicity, which means that the overhead is low as the analysis necessary done for each scheduling is low. In many cases this makes a RM scheduler faster and more efficient then an optimal scheduler with large overhead.

It does, however, not scale well. Zapta and Alvarez did an extensive comparison of global and partitioned RM against partitioned and global EDF schedulers [28]. Their study show that EDF generally outperforms RM both partitioned EDF vs partitioned RM and global EDF vs global RM.

Earliest Deadline First (EDF)

Earliest deadline algorithm was presented by Liu and Layland [22]. Its a simple and optimal uni-processor algorithm. As long as the total utilization does not exceed a hundred percent of the system capacity the task set is schedulable:

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.2}$$

Liu and Layland proves EDFs optimality by showing, that, if there exists a set of task $\{\tau_1, ..., \tau_n\}$, each characterized by arrival time T_i, computation time C_i and deadline D_i. If this task set can be scheduled by any algorithm

such that no deadlines are missed, it can also be scheduled by the EDF with no deadline missed.

EDF assigns the task with the least time to its deadline the highest priority and is allowed to run first. Hence the name earliest deadline first. It then continues to schedule tasks according to their deadline. EDF implementations usually handles sporadic tasks (tasks that does not have a fixed arrival time, not reoccurring) by giving them a static portion of the resource, making the schedulability analysis easier.

The analysis and verification of schedulability is simple, as long as the utilization is below 100% it is schedulable. This makes EDF a simple and optimal uniprocessor scheduler.

A known issue is that EDF does not scale well, and is not optimal for a multiprocessor systems. This was demonstrated by Dhall and Liu [15], and the limiting effect is thus knows as "Dhall's effect". Due to combination of both heavy and light tasks in scheduling the utilization bound is limited. Andersson et al shows in [6] that the maximum utilization of global or partitioned multiprocessor EDF scheduler is 50%

They show that if we have a task set $\tau = \{(T_1 = 2L - 1, C_1 = L\}, (T_2 = 2L-1, C_2 = L), ..., (T_m = 2L-1, C_m = L)), (T_{m+1} = 2L-1, C_{m+1} = L)\}$ to be schedulable on m processors, and all the tasks arrives at the same time the utilization for this task set will be $L/(2L-1)+(L/(2L-1)/m$ and they proceed to show that static priority scheduler such as global or partitioned EDF will not be able to schedule it. If τ is run in a global static priority scheduler where m high priority tasks $T_i..T_m$ will run in parallel and execute in L time. There will then be $L-1$ time left for the low priority task $m+1$, which is not enough as its execution time is L.

There exists several modified versions of EDF algorithms that tries to avoid Dhall's effect, by taking advantage of assumptions that there are always light weight tasks and heavy weight tasks. They try to schedule the heavy tasks individually, hence avoiding Dhall's effect.

Summary

This section was a summary of our survey on real-time multiprocessor scheduling [17]. Two different multiprocessor designs was presented; partitioned and global. We also explained how shared resources can cause problems for schedulers. Towards the end of the section two well known real-time schedulers was presented, and a technique to improve cache reuse.

Chapter 3

Experimental Language Grammar

In this chapter we will present our experimental concurrent real-time language. We focused on creating a language where real-time requirements could be described, and enable users to easily write concurrent code. Our language supports a subset of occam's grammar and is also extended to support new statements. The grammar is written in BNF, which was presented in 2.1.3. It is used in BNFC to auto generate a Haskell lexer and parser. The intermediate code generator was created entirely by us, and the code generated based on the grammar is the subject of the next chapter.

The complete BNF grammar can be found in appendix A.

3.1 Language family

As mentioned earlier we wanted to create a language focusing on two features; real-time support and concurrency. Our language is based on a subset of occam's grammar; it includes the features from occam we saw best fit for our goals. We has excluded other features, either due to their complexity or that they where unnecessary for demonstrating the principles of the language.

occams fine grained concurrency was one of the features that was attracting. This is something that can be utilized in a multiprocessor real-time scheduler (P-fair see [17] for more details) to avoid Dhall's effect, by only having light processes there are no heavy blocking processes. With occam there

is no guarantee that heavy process is not created. But it makes it easier to avoid it, if the user is aware of the problem. In occam it is easy and natural to write concurrent code, due to it strict rules on shared variables and keywords for expressing parallel code. Hence, reducing the sequential part of the program. Amdahl and Gustaffson (see section 2.3.2) laws show how the sequential part of the code affects the scaleup possibel through parallelisme.

Synchronization between the fine-grained processes is done through communication over channels. Channel communication in combination with strict rules on how shared variables, is an attractive feature for a real-time language. As channel communication is the only way to share data, it described the dependencies between the processes. Clearly this is something that can helpful when scheduling the tasks.

We used the BNF converter to generate an Alex lexer and a Happy parser. Our BNFC grammar is based on the BNFC grammar of Martin Korsgaards Time/occam language [19].

In this section we will describe the grammar of the language. First it is nessary to clearify the relationship between occams description in the occam 2,1 reference manual [21] and our. In occam/CSP simple statements such as an assigment is called a *primitive process*. More complex statement, which is constructed from other statements, is called *constructed processes*. To avoid confusion between by using process is in the language as describing a statement, and using it when talking about the the run-time system system another way. We will be using statement about what Occam calls *processes*.

3.2 Layout

Occam has a strict policy when it comes to indentation, exactly two spaces extra for every nested block. Due to the implementation tools, we used `C` like syntax on blocks, opening and closing brackets at the start and end of a block. Every statement is ended by a semicolon. Indentation has no effect, but is recommended.

3.3 Hello World

Every program has to at least consist of a main procedure, which is the first procedure to be run. A procedure is declared with the keyword `PROC`,

followed by the procedure name `Main`, where the first letter has to be capitalized. Every procedure also has to have a deadline specified, which is used when scheduling the procedure. The procedure declaration is followed by a block that starts with a order keyword, which declares if the statement should by run in parallel or in sequence. The statement run in "sequence" is a print statement printing the string `"Hello World"`. Later in this section we will go more into detail on all the components in the hello world program.

```
PROC Main : 4S : {
    SEQ {
        Print "Hello World";
    }
}
```

Listing 3.1: Hello World

3.4 Procedure Declaration

As shown in the Hello World example, a procedure has to be defined with the `PROC` keyword, a name, and a deadline. As mentioned above, every program must at least have a `Main` procedure which is the "start" procedure. Every other procedure can be named anything as long at it starts with a capital letter and uses only letters from the latin alphabet, digits and under score.

The regular expression used to describe the rules for a procedures name is: $\langle upper \rangle (\langle upper \rangle \mid \langle digit \rangle \mid \text{`\_'}) * \langle lower \rangle (\langle letter \rangle \mid \langle digit \rangle \mid \text{`\_'}) *$

A procedure declaration must also contain the procedures deadline; it is specified after the procedure name with a colon as separator. A deadline has to be specified with an integer, and the resolution of the time. The time can be specified in resolution from seconds to nano second. The full list of keywords is presented in listing 3.2.

```
S - Seconds
Ms - Milli seconds
Us - Micro seconds
Ns - Nano seconds
```

Listing 3.2: Timer resolution keywords

The deadline and the procedure block is also separated by a colon. The reason for adding time to an occam procedure is due to the real-time aspect of the language. Instead of, expressing how important a task is through a priority, developers can explicitly say that; this procedure must finish before this deadline from the moment it is initiated. To think that all procedures can be given a deadline no matter what, might be a bit naive. However, we choose to disregard "general" procedures that might have no deadlines, and solely focus on a concurrent real-time language.

The procedure grammar written on BNF:

⟨*GlobDecl*⟩ ::= `PROC` ⟨*CapIdent*⟩ `:` ⟨*DeadLine*⟩ `:` ⟨*SingleBlock*⟩
| ⟨*Chan*⟩ `;`

Where ⟨*Deadline*⟩, ⟨*CapIdent*⟩, and ⟨*SingleBlock*⟩ is a nonterminals. ⟨*SingleBlock*⟩ is a special procedure block. We will described how the specification for this nonterminal in the next section.

3.5 ⟨*SingleBlock*⟩/Procedure Block

The block in a procedure declaration is different from a statement block, such as a for loops. A procedure block is divided into two parts; a variable declaration at the beginning, and a single statement block.

Written on BNF:

⟨*SingleBlock*⟩ ::= { ⟨*ListVar*⟩ ⟨*StmtCompound*⟩ }

⟨*ListVar*⟩**/Declarations** To simplify the language all variables are integers in our language. A variable is declared with the keyword `VAR`.

In beginning of the statement block a list of variables and/or arrays can be declared. They can not however be assigned. Assignment is considered a statement, and has to be executed inside the statement block.

There were several reasons for limiting the variable types and declaration. For instance, by limiting the amount of types both stack handling and communication is easier, as variables can only by one size (see chapter 5). By only having one type of variables we could also limit the expression and statement necessary. This resultes in a simpler compiler, enabling us to focus on concurrency and communication mechanisms.

```
PROC Main : 4S : {
    VAR var;
    VAR array[10];
    SEQ {
        TestProc1;
        TestProc2;
    }
}
```

Listing 3.3: Decleration of a variable and an array, and a sequential statement block where two procedurees are initiated.

⟨*StmtCompound*⟩**/Statement Compound** After the variable has been declared, the procedure block requires a statement compound. A statement compound is a block with a list of statements. For every statement compound there has to be specified the order the statements in the compound are to be run in. This is specified through the order keyword, which specifies if the statement is to be run sequential or in parallel. A typical statement compound can be found in listing 3.3, where there are two procedure call statements: first, `TestProc1` is called, and secondly `TestProc2`.

3.6 Expressions

Expressions are quite similar to C expression, but again the number of operators is very restricted. Our language supports subtraction and addition. The reasoning behind restricting the type expression, was that they where not vital to demonstrate the concurrency features of the language.

⟨*Expr1*⟩ ::= ⟨*Expr1*⟩ + ⟨*Expr2*⟩
| ⟨*Expr1*⟩ − ⟨*Expr2*⟩
| ⟨*Expr2*⟩ `[` ⟨*Expr2*⟩ `]`
| ⟨*Expr2*⟩

3.7 Statements

Our languages statement is also limited, but supports enough statements to demonstrate concurrency and real-time features. We chose to disregard

conditional statements such as `WHILE` and `IF/ELSE`. The argument for limiting the number of statements is the same as for limiting the variable types. By disregarding them the compiler implementation is simplified, and they are not necessary to demonstrate the concurrency and real-time behavior of the language.

The BNF grammar for the included statement is:

⟨*Stmt*⟩ ::= ⟨*LowIdent*⟩ ⟨*AssignOp*⟩ ⟨*Expr*⟩ `;`
| ⟨*LowIdent*⟩ ⟨*Array*⟩ ⟨*AssignOp*⟩ ⟨*Expr*⟩ `;`
| `DeSchedule ;`
| ⟨*CapIdent*⟩ `;`
| ⟨*Expr*⟩ `;`
| `Work ;`
| ⟨*StmtCompound*⟩
| `Print` ⟨*String*⟩ ⟨*ListExpr*⟩
| ⟨*ChanCom*⟩ `;`
| ⟨*Stmt*⟩ `;`

Statement Compound/⟨*StmtCompound*⟩

As mentioned statement compounds is a vital part of a procedure, and is really where behavior of the program is written. In our language there are two types of statement compounds, which both run statements in the specified order. The first type is a statement block, and the second a for loop.

The BNF description of statement compounds:

⟨*StmtCompound*⟩ ::= ⟨*Order*⟩ `FOR` ⟨*LowIdent*⟩ ⟨*AssignOp*⟩ ⟨*Integer*⟩ `TO` ⟨*Integer*⟩ `:` ⟨*StmtBlock*⟩
| ⟨*Order*⟩ ⟨*StmtBlock*⟩

Statement Block (SEQ and PAR)/⟨*Order*⟩ ⟨*StmtBlock*⟩

In occam `SEQ` and `PAR` are placed in front of a statement block. The keywords implies in what order the list of statement inside the block are to be ran in. If it is a `PAR` block all the statements are run in parallel. The statements in a `SEQ` block are run sequentially. This makes it possible to create fine grained parallelism. A statement block is also a statement, so a statement block can consist of several nested statement blocks.

For Loop/⟨*Order*⟩ `FOR` ⟨*LowIdent*⟩ ⟨*AssignOp*⟩ ⟨*Integer*⟩ `TO` ⟨*Integer*⟩ `:` ⟨*StmtBlock*⟩

For loops is just like for loops in any other language: where a counting variable has to be declared an assigned to a value, and a limit to iterate to. Except from it to that the initial counter value, and upper limit has to be declared with integers. This is due to stack inheritance, which we will come back to in the next chapter.

```
SEQ/PAR FOR i = 0 TO 10 : {
  ...
}
```

Listing 3.4: For loop declared

An order has to be specified at the beginning of a for loop declaration. It specifies the order the for iterations of the loop is executed in, which is different from a statement block where it is the order of each statement it specifies. For instance, in a parallel for loop each of the iteration is run concurrently. The body of the for loop can be seen on as a procedure. This differentiates it self from pure parallel statemnts block where the statements are run in parallel. This separates our language from occam, where a for loop is followed by a procedure block. As before, this change is due to simplification, and that this is a more natural way of describing concurrency. With such a for loop, the parallelization pattern "loop parallelism" is easily described (see chapter 2.3). Of course it is also possible as Occam to use a single statement or statement blocks in the block of a for loop.

Procedure Call/⟨*CapIdent*⟩

When a procedure has been declared, it can be called from inside a statement compound. The procedure is then started and ran. If a procedure is called within in a `PAR` compound, the procedure is run concurrently with other statements in the block. A procedure can not take any arguments; hence a procedure can be called by its name only, such as `TestProc1;` in listing 3.3.

Assignment/⟨*LowIdent*⟩⟨*AssignOp*⟩⟨*Expr*⟩

Variables and arrays have to be assigned inside a statement block. Assignment in our language is just like in C. On the left hand side of the equal sign is the variable that is assigned, and on the right hand side an expression.

```
1 a = 3;
  a = 2 + a;
```

Listing 3.5: Variable being assigned

De Schedule/`DeSchedule`

In our language we have also introduced a new statement, `DeSchedule`, which de schedules the process currently running. It was necessary to have such as a statement when the scheduler is non-preemptive, as our is (see section 5.1). It enables the programmer to explicit tell when a procedure can de scheduled. A typical usage of the statement can be if procedure comes to a computing intensive part of the code, which is not a critical computation. By using the deschedule statement the scheduler can let time critical processes run instead, before returning to the descheduled procedure.

```
  PROC Main : 4S : {
2     Var a;
      SEQ {
4         Work;
          a = 3;
6         DeSchedule;
          Work;
8     }
  }
```

Listing 3.6: Descheduing a procedure "working"

Work/`Work`

Additionally to the deschedule statement we added a work statement. The work statement is directly linked to a work function in run-time system, which does a computational intensive task. The statement is useful in an experimental language such ours, enabling simulation of intensive algorithms, without having it to implement again in every test case. It is especially useful when creating a number of intensive task that interferes with time critical processes, which will be demonstrated in chapter 6.

Print/`Print`

⟨*String*⟩ ⟨*ListExpr*⟩ The last statement we added was `Print`. A `Print` statement has to be followed by a string, or a list of expression or both (demonstrated in listing 3.7). Each variable is separated by a space and a comma, and ended with a newline. It simply utilizes print functions in the generated language, and prints to the standard output.

```
1 Print "Hello World";
  Print "Hello World" var;
3 Print "Hello World" var, 3;
```

Listing 3.7: Print statement printing a string and expressions

Channel Communication/⟨*ChanCom*⟩

Channels are declared globally and shared by the procedures. They can, however, only be used in two procedures. There are two symbols reserved for channel communication send `!` and receive `?`. A procedure which uses a channel can both send and receive on that channel multiple times, but it can only communicate with one other procedure through this channel. The reason for limiting channel communication is due to the dependency analysis done at compile time. For the dependency analyzer it is necessary that it can at compile time map which procedures depend on whom. When a channel is shared by more than two procedures, sent messages has to be broad casted to all the procedures sharing the channel. A procedure waiting on receiving on a channel with multiple senders *might* depend on multiple procedures, and we can never know which one will be the actual sender. Such a solution will then result in a more complex dependency analysis.

The BNF description of channel communication is based on CSP channel communication. Reserving `!` and `?` for send and receive messages over a channel:

⟨*ChanCom*⟩ ::= ⟨*ChanExpr*⟩ `!` ⟨*Expr*⟩
| ⟨*ChanExpr*⟩ `?` ⟨*Expr*⟩

```
1 CHAN chan_a;

3 PROC Test1 : 1S : {
    VAR var1;
```

```
5       SEQ {a
            var = 44;
7           chan_a ! var1;
        }
9   }

11  PROC Test2 : 1S : {
        VAR var2;
13      SEQ {
            chan_a ? var2;
15      }
    }
```

Listing 3.8: Two procedurees communication over a channel

In listing 3.8 there are two procedures communicating over a channel. `Test1` assigns the variable `var1`, and then sends it over the channel `chan_a`. `Test2` waits on receiving on `chan_a`, and then reads the data sent over the channel into the variable `var2`.

It is also possible to declare channel arrays, and then use an element in the array to communicate over. However, an element in a channel array can only be referred to by the loop counter or a constant. The reason for this restriction is that by allowing any type of expression as reference, the dependency analysis gets very complex. Something we will come back to in the next chapter.

Summary

In this chapter the grammar of our experimental languages has been presented. The grammar is written on BNF (Backus-Naur Form) and is based on Martin Korsgaard Time/Occam grammar [19].

occam requires that statement is grouped together in blocks, and the order the statement can be executed in. By `SEQ` the statements are executed sequentially, `PAR` the statements are executed in parallel.

Additional to ordinary occam statements such as, assignment, print, procedure call, our language has the statement created for testing purposes, `DeSchedule` and `Work`. `DeSchedule` stops and deschedules the process, and `Work` triggers a work function in the run-time that does a computational intensive task.

Last channel communication was described. Channels has to be declared globally, and message can be sent on a channel with the keyword `!` and received with they keyword `?`.

Chapter 4

Compiler Generated Code

In the previous chapter we presented the grammar of our language. In this chapter we will present the code generated by the intermediate code generator in the compiler. Our compiler is written in Haskell and the lexer and parser is generated with help of BNFC, and the grammar presented in the previous chapter. The Haskell code generator reads the syntax-tree (see section 2.1.2) and generates C code as output[1]. The generated code can then be compiled with a C compiler, such as gcc. In some cases the relationship between our language and C is close; this helps us to easier describe the transition between them. For instance, an assignment statement `a = 2` would be exactly the same in C. However, some of the features in our language, such as `PAR` bocks, channel communication, concurrent execution of processes; require more complex code to be generated.

4.1 Processes Generation

A process is piece of the program that can run on its own. Processes can be generated by two statements in the language, procedures and `PAR` blocks. Both statements are restricted by the grammar to only interact with other processors in particular ways. For instance, in a `PAR` block a variable can be read by all the concurrent processes, but only one of the processes can write to it. A procedure on the other hand, can only interact with other processes through channel communication.

[1]Please see the attached source code for specific details on the code generators implementation

Let's take a look at how `PAR` generates processes. In a `PAR` block each statement in the block is run in parallel, so each statement gets a C function and a separate process that executes the C function. In a `PAR FOR` the statement block is implicitly interpreted as a sequential block, where each iteration of the loop is run parallel. Hence, each iteration is run as a separate process, and a C function is generated based on the statement block in the for loop.

4.2 Duff's Device/Co Routines

To enable a processes to be run concurrently and be able to be descheduled while running. It is necessary to know where the process stopped, so it can continue from that point when run again.

An example of where a process is descheduled is; when a process is waiting for a incoming message on a channel, it then has to wait on another process to transmit the message before continuing execution. It is easy enough to block a process if each process was run on an independent OS-thread, and we let the operating system handle scheduling. However, if we want control scheduling and process execution it is necessary to enable a blocked process to be deschudeled until it is ready to execute again.

The problem is then: how do process know the state it was in when it was descheduled? What where the values of its local variables? Which instruction was it blocked on ? etc. The traditional solution to this problem is to have a stack where the data is kept. The run-time system can pop and push values and pointers that needs to be stored until the process is ready to run again. However, a stack increases the compliers complexity, and a stack must be used with care as the portion of memory it uses grows for each time new data is added.

Duffs device A different approach to remember the state the process is by using a design pattern known as Duff's device (presented in [34]). It utilizes high-level language mechanism to get a stack like functionality, without the hassle of memory management.

```
2 int fuction() {
      static int i = 0;
4     static int pc= 0;
```

```
6       switch(pc) {
            case 0:
8               i++;
                pc = 1;
10              return i;
            case 1:
12              i--;
                pc = 2;
14              return i;
            case 2:
16            ....

18      }
    }
```

Listing 4.1: Duffs device

So we wish to enable an implementation where we can deschedule a function, and later return to the point where we last left of. The function above does some simple incrementation and decrementation on a single variable.

A switch case such as the one in the example, "fall trough" and continues to execute the next case if not "breaked". We can use this to split our function into parts. At a point where we wish to yield we simply save the `pc` of the next case and return.

To store the local variables they can be defined as static, if the Duff's device is only used by one thread at the time. If the function is executed in parallel/"shared", the local variables and process counter has to be saved externally. A good solution is to create a struct holding all the variables and information of the Duff's device, such as, pc and function pointer. All references to variables in the function then have to be to the variables stored in the struct.

CCSP/Kroc has a design similar to Duff's device. It is a jump table, consisting of pointers to labels in the function. The labels point to different parts of the function. For instance, a label is placed where the process may be blocked. If the process is blocked, it can jump directly there next time it is run (for more details please see [27]).

In contrast to CCSP, JIBU has no deschudelings point, so once a process is executed in will execute until it is done. It is up to the operating system to schedule the different OS-threads created (see section 2.4).

GO uses a stack to hold process information. The scheduler in GO is non-preemptive, so a process has to be yield to stop execution. When the process is yield the program counter is set and stacked. The scheduler can then retrieve another process. The stack given to each process is minimal only a few kilobytes, but can be extended as the process needs more. By giving each process a minimal stack the cost of context switch is less. GOs also use assembly instructions to switch between processes, which helps decrease the cost of a context switch (see our discussion with Russ Cox on GOs stack [14]).

Our implementation To avoid the complexity of complete stack manipulation, we chose to use Duff device when implementing processes in the run-time system. Variables and data is kept in a struct we call a stack. It holds the information about what state the current processes is in, and its variables. However, it is important to note that it is not a stack in a traditional sense. The two main reasons for choosing this implementation were; it reduces complexity, and it makes stack inheritance between processes easier.

As described earlier the compiler generates C function that processes runs. During compilation when the compiler comes across a statement where the process can be descheduled it generates a new `case` in Duff's device. A typical example of such statement is channel communication. Additionally they can be explicit introduced by the programmer trough the `Deschedule` statement:

```
1 DeSchedule;

3 \\Generated code

5 stack->pc = 1 ;
  procStatus = ProcReady ;
7 goto finalize ;
  case 1 :
```

Listing 4.2: The generated code generated of a DeSchedule statement

We will in the remaining part of this explain some of the code generated to support the concurrent execution of the processes.

4.3 The Stack

As mentioned earlier, a stack struct holds all the variables and the information about the state of a process. At compile time when a procedure is parsed the number of variables declared is counted. A variable count variable are then generated together with the C function. Both the function and the variable is used as argument when a new process is created.

The variable count is necessary when the size of the processors stack is allocated. The variables are stored in an array of the specified size. Each variable name is then linked to its position in the array. The first variable declared lies in position one, the second in position two etc. To avoid explicitly link each variable name to their number in the generated code, the compiler keeps a list over the names and numbers, and inserts the array reference where the variable is used. The stack also contains the process counter, which controls which `case` to run in the co routine generated, and a pointer to a copy of its parent's stack.

4.4 Stack Inheritance

As mentioned earlier, a process is generated for each statement in a `PAR` block, and each iteration in a `PAR FOR`. These statements are parts of a procedure and can access the variables declared in the procedure. Hence, a process has to be able to access its parent's variables. Each process is therefor given a copy of the parent process. Processes generated of `PAR` statements is handed a copy of the stack of the processes they are in.

`PAR` block can be nested, a `PAR` inside a `PAR` block. Therefor, it is necessary to keep track of on what level the variable are declared, and on what level they are used. The compiler does this by having a status variable for each block. The status is changed each time a new block inside the original is parsed, and when it is done. Each time a variable reference occurs; the level the variable is declared on, and the current level is checked. It then uses the difference between the levels to generate a reference to the correct parent stack at the correct level.

For instance, in a procedure with two nested block, as the one in listing 4.3.

```
PROC Test : 4S : {
    VAR a;
    PAR {
        PAR {
            a = 2;
        }
    }
}

\\ Generated variable assignment

parent -> parent -> vars[2];
```

Listing 4.3: Compiler generates reference to the parent stack at procedure level

Where the assignment of `a` is inside two nested block, hence the correct parent stack is two levels up.

Each process is given separate copies of the parents stack. The reason for this is that; with no pointer interaction there is no unwanted side effects, and no need for guards to protect the memory. It however, requires synchronization of the memory when all the children processes are done executing, which is something we will come back to in section 5.2.

4.5 Process Status

To tell the run-time system what state the processes is in when the C function returns. We have defined a set of statuses a processes can have (see listing 4.4).

```
enum ProcStatus {
    ProcWaiting = 0,
    ProcReady,
    ProcRunning,
    ProcDone,
    ProcMainDone,
};
```

Listing 4.4: Processes statuses

If the process is blocked, either due to waiting on synchronous communication, or on children processes to complete. It returns the status `ProcWaiting`.

If it is descheduled voluntarily through a `DeSchedule` statement it has the status `ProcReady`. A process should never return with the status `ProcRunning`, it is only used in the run-time, which is only set when a process is run.

When a process is complete it returns `ProcDone` or `ProcMainDone` depending on if its the `Main` procedure or not. When the process is done it triggers cleaning mechanisms in the run-time system, cleaning up the memory after a process.

4.6 Process Initiation

We have now explained how processes are created, but not how they are initiated. A new processes is initiated by a procedure call or through `PAR` statement compounds. A procedure call results in that the run-time process creation function is called, with the function name, variable count, deadline, and dependency functions, and the process that created it as arguments. This process is then run as an individual process, and scheduled based on its own deadline.

```
1 newProc (Test, Test_var_count, Test_var_deadline,
    stack, Test_chan_dep, proc);
```

Listing 4.5: A new process is created

As mentioned processes are also created for `PAR` statement blocks, the process is then initiated where the `PAR` statement is is the code. It uses the same run-time function, but with the auto generated function as argument.

4.7 Channel Communication

Our language support synchronous communication. In listing 4.6 we can see code generated from a channel send statement. It first checks the status of the channel. If there is a waiting process it sends on channel, and notifies the waiting process, if not it register as a waiting process and returns.

The code would look almost the same for a receive statement, except using the run-time receive functions.

The example in listing 4.6 is a bit simplified; the generated code would also have to use a channel lock to make sure that the channel is not compromised. We will in the next chapter explain how the run-time system is designed to support synchronous communication.

```
chan_a ! a;

\\Generated code

int chan_status ;
chan_status = channelReady (chan_a);
if (chan_status){
    sendOnChannelSynch (a, chan_a);
}
else {
{
    sendOnChannel (a, chan_a);
    deschedule and wait;
}
```

Listing 4.6: Simplified compiler generated C code of a channel send statement

4.8 Dependency Analysis

Besides generating code our compiler also does dependency analysis to improve deadline misses. Due to the strict language grammar and communication, can almost only occur through channel communication. The result of these strict rules is that a process is directly dependent on others through channel communication occurs. By analyzing and mapping channel communication, the compiler can get a complete "picture" which process depends on whom.

There is, however, one other way a process can be dependent on other processes, this is through parent child relationship. As explained earlier, if a process contains a `PAR` block it has to wait for all the children process to complete before it is complete. This dependency relationship can not be exploited to improve efficiency, as a `PAR` block can not be executed earlier than when it is called inside the parent process. Such a behavior could

potentially change the behavior of the program, which has to be avoided when optimizing the code (see section 2.3.6).

The compiler saves the procedures using a channel, on channel communication statement. It has a list over all the channels declared in in program, and saves the two belonging procedures when the code is compiled. The compiler then generates dependency functions for each process, using the information. The generated function is called when the processes is initiated. The function creates a list over which channels a process depend on, and a list of the processes that uses a channel (Listing 4.7 is an example of such a auto generated function). These lists are used by the scheduler, as we will see in the next chapter.

```
void Proc0_chan_dep (struct Proc * proc){
    addProcAsDep (chan_a, proc);
    addProcAsDep (chan_b, proc);
}
```

Listing 4.7: An auto generated channel dependency function

Summary

In this chapter the code generated by the compiler was presented. First how a process is structured was presented. How Duff's device enables processes to continue where they left of after being descheduled. The compiler also generates a variable count, which determines the size of a process' stack. We briefly explain how the complier from a channel communication statement generates a channel check to determine if there is a waiting process on the other end of the channel, and sends or receives on the channel.

In the last section we presented how dependency analysis is performed was. It maps the usage of the channels in the program, and generates functions adding the according processes to the channel they use. The functions are run when the process is created.

Chapter 5

The Run-Time System

When the compiler has generated runnable code of source code it is run in the run-time system. Its job is to perform language mechanisms such as channel communication, memory handling, creating, destroying and scheduling processes. Our run-time system utilizes parallel programming patterns (presented in section 2.3), and mechanisms from the run-time systems to known concurrency oriented languages such as GO, Erlang, and occam (see section 2.4).

In this section we will present how the scheduler works, how stack inheritance is performed and how we organize processes.

5.1 Scheduler

The run-time system is built around a multiprocessor EDF scheduler. We based our choice of scheduler on our conclusion from our real-time multiprocessor scheduling survey [17]. The scheduler is a plain EDF scheduler which later can be extended to a modified EDF scheduler, which is discussed in chapter 8. For testing purposes two versions of it was implemented; one using the dependency analysis performed during compilation (see section 4.8), and one scheduling only based on deadlines. We will first present the pure EDF scheduler.

Pure EDF Scheduler The scheduler is based on the master/worker pattern. The number of worker threads is specified in a macro and has to be one pr. CPU. This is due that each worker is locked to a CPU with Linux

specific mechanisms. The worker threads retrieves the process with the earliest deadline from a shared run queue (ready queue) holding all the ready processes. It then executes the process generated C function. Depending on the process status returned by the function, the process is either put back on the ready queue (`ProcReady`), put on the waiting processes queue (`ProcWating`), or cleans up after the process (`ProcDone`) by freeing used memory etc.

When a process is complete all processes which are waiting on it are notified. A process may wait on more than one process, and each process has counter that is decremented each time a process it depends on notifies it. When the counter reaches zero, the process is moved from the waiting queue to the ready queue, and retrieves the changes done in the stack copies the children process held. How this mechanisms work we will come back to in next section.

When the worker has completed the task related to the processes status returned, it retrieves the next process with shortest deadline from the ready queue.

Dependency Based EDF Scheduler The dependency based multiprocessor EDF scheduler is based on the pure EDF scheduler, and uses the same framework. It starts by choosing the processes with the lowest deadline, as the pure EDF would. Based on the dependency lists generated by the compiler a separate run queue is filled with the dependent processes of the process with the earliest deadline. These processes is then run by the workers in the system.

As long as this run queue contains processes the workers execute them, as illustrated in figure 5.1. When the run queue is empty, and there are no dependent processes it starts over; by retrieving the next process with the lowest deadline from the ready queue.

Each process contains a list of the channel they use and depend on. The list is a FIFO list, which means that the first channel in the list is the first channel used in the process. Each of these channels contains a list over the processes which use it. By combining the information in both these list the scheduler can map all the processes the current process depends on, *directly* and *indirectly*. The worker then starts to execute all the dependent processes concurrently, working "together" to meet the earliest deadline.

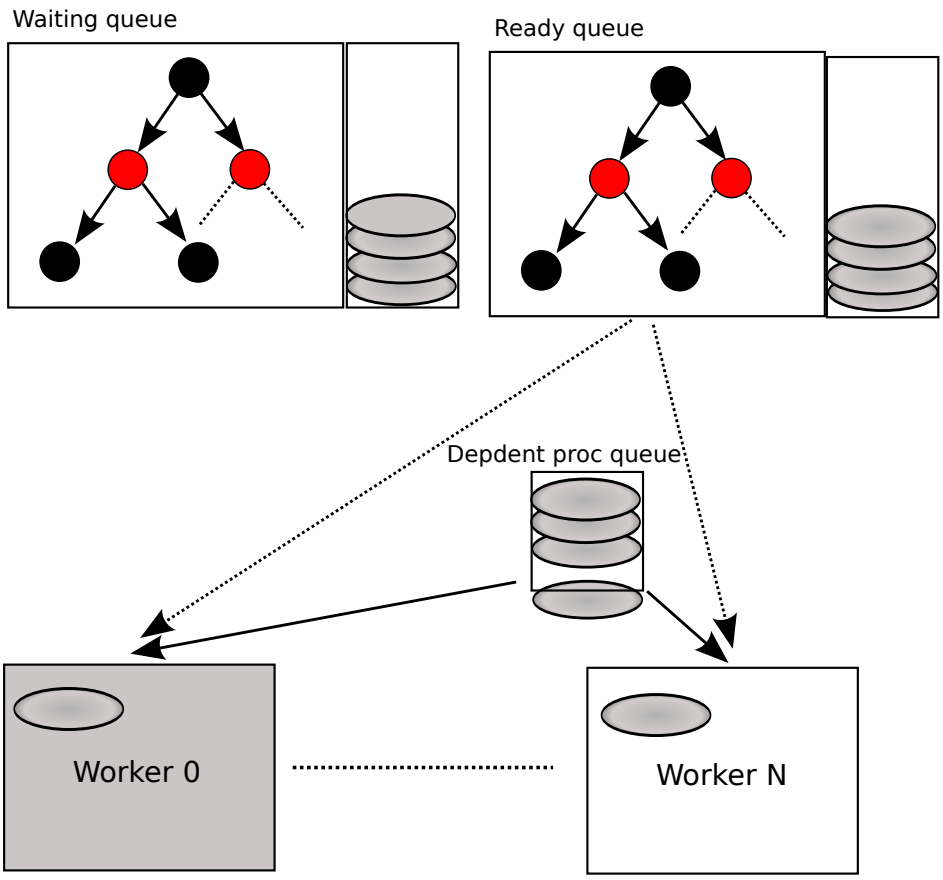

Figure 5.1: Worker and process queues

5.2 Stack

The stack holds information about the state of a process, and all the variables used in the process. As mentioned in the previous chapter (see section 4.3), each process generated from a `PAR` block is given a copy of its parent stack, enabling concurrent read and write. Because each child process has a copy, it is necessary to synchronize them with the parent when they are done. So the parent can preserve the changes done in the child processes. When a child process is done, the parent is notified, and queues the child's copy of its stack. Then as the "last" child process completes stack merging is triggered. All the children stacks are compared with the parent stack, and changes copied. Due to the strict grammar, not more than one of the child process can write to a variable, which makes the merging easier.

These rules would also make it easier to hand each child process only the pointer to the parents stack instead of a copy. However, separate copies means that there is no need for locks in the stacks, and a "shared" stack will not become a bottleneck.

Separate memory is also beneficial for multiprocessor systems. As memory does not need to be synchronized across cache (please see [17] for more details).

In all the run-time systems presented earlier in section 2.4 processes have separate stacks, and does not allow stacks to contain pointers to other stacks. There arguments for this is the same as our. Shared data increases complexity and can become bottlenecks (as argued by Armstrong in [2]).

5.3 Channel Communication

Channel communication is made possible through run-time mechanisms, and compiler generated code. The compiler generates code that checks the state of the channel, and uses run-time functions to perform the communication (see section 4.7).

There are two different versions of each of the communication functions, one pure that copies the message, and one that additionally synchronizes with the waiting process. These functions are simple, because most of the work is either done in the compiler, or in scheduler.

Let us take a look the code generated from a channel communication statement:

```
int chan_status ;
chan_status = channelReady (chan_a);
if (chan_status){
    sendOnChannelSynch (a, chan_a);
}
else {
{
    sendOnChannel (a, chan_a);
    deschedule and wait;
}
```

Listing 5.1: Simplified compiler generated C code of a channel send statement

Both of the functions take the message and the channel as argument. The channel holds the data to be sent, and a pointer to the sender and receiver process. It also contains a flag used to notify if there is any processes waiting for channel communication, and a list over the process using the channel.

If there is no waiting process on the "other" side of the channel, the process writes its message to the channels data variable, sets the waiting flag, and sets its status to waiting and deschedules. The scheduler then puts the process in the waiting queue, and retrieves a new process.

However, if there is a waiting process on the "other" side of the channel, the process has to be notified that a message is ready to be sent. The process then uses the `Sync` version of the sender function, which stores the message and than moves the process waiting from the waiting queue to the ready queue. The process that was waiting to receive on the channel is then scheduled normally. And retrieves the message next time it is run.

5.4 Advanced Data Structures

In our run-time system we have implemented two different queue structures; a linked list, which can hold both channels and processes, and a red-black tree that holds all the processes in the run-time system. In this section we will present how they are used in the run-time system, but not present how they are implemented. This is because they both are based on well known designs.

Linked List

Linked-lists are used in dependency analysis to hold processes using channels, and the channels a process uses. These lists are only used to hold data where the run-time system only accesses the first element. Queuing and dequeuing can be done in constant time $O(1)$ disregarding the size of the list [13]. Searching a linked list can be done in linear $O(n)$ disregarding if it is sorted or not. This is why such lists are only used in a FIFO manner. Searching the list with many elements can quickly become time-consuming.

Red-Black Trees

To avoid the search time of linked lists we chose to implement red-black trees to hold the process in the wait queue and ready queue.

A red-black tree is a tree that is always balanced. It guarantees that the length of the left sub-tree of a node is always ± 1 the length of the right sub-tree. This enables such trees to have a worst case run-time for insert, delete and search of $O(\log(n))$ [13]. This is very good when we want a predicable and fast search.

Red-black trees are complex and can be difficult to implement. Therefor, we chose to use the red-black framework from the Linux kernel. The framework, however, requires that a large part of the code has to be implemented. Such as, search, insertion, deletion, the reason for this is that the sorting criteria have to be based on what the tree contains [12].

In our case there are two different trees sorted with different criteria. The ready queue is sorted by their deadline and process id (PID), and the waiting queue is sorted based only on PID. The queues can then be searched based on deadline, or PID depending on the queue. The kernel framework provides functions for inserting and deleting nodes, and keeps the tree balanced (for more details on the framework and how to use it please see [12]).

Summary

In this chapter the design of our run-time system was presented. We built two multiprocessor EDF scheduler based on the master/worker principle, one pure and one utilizing the data from the dependency analysis done during compilation.

How stacks are copied and merged was presented. We also explained how channel communication uses scheduling mechanisms to notify waiting processes. And in the last section we briefly described that data structures used to store processes and channels in the run-time system.

Chapter 6

Examples and Performance

In this section we will present an example where process dependencies affect the scheduling, and explain how it reduces deadline misses. At the end we will present the results of performance tests done on, Erlang, GO, occam and our system.

6.1 Concurrent Hello World

In this section we will present a modified Hello World program, which shows the main features of our experimental language. Readers that feel comfortable with the language grammar can jump a head to the next section.

```
CHAN chan[10];

PROC Hello : 1S : {
    VAR a;
    PAR FOR i = 0 TO 10 : {
        chan[i] ? a;
        Print "Hello World" a;
    }
}

PROC World : 1S : {
    PAR FOR i = 0 TO 10 : {
        chan[i] ! i;
    }
```

```
}

PROC Main : 500 Ms : {
    PAR {
        Hello;
        World;
    }
}
```

Listing 6.1: Concurrent Hello World example

The program consists of three procedures `Hello`, `World` and `Main`. The `Main` procedure is a required procedure that always has to be declared, it is also the first procedure to be run. It has a deadline of 500 Ms. It consists of one `PAR` block, where the two produces `Hello` and `World` are called. Because the procedures are called inside a `PAR` block the procedures are run concurrently as separate processes, both with a deadline of 1 second.

`World` is the simplest procedure. It is made up of one `PAR FOR` loop, that goes from `0` to `10`. Each iteration in the `PAR FOR` is run as a separate process concurrently with the others. The body of the `PAR FOR` consist of one channel communication statement, sending the loop counter `i` over a channel `chan[i]`.

The procedure `Hello` starts with declaring the variable `a`. All variables which is going to be used in a procedure has to be declared at the beginning of the procedure, such as `a`. The statement block in the procedure is the same `PAR FOR` as in `World`. But instead of sending over a channel in the channel array, it receives a message. The message is read into the variable `a`, and is then printed together with the string "Hello World".

The channel array is declared globally, and each element can only be used by two procedures at the time. This is due to the dependency analysis performed by the compiler.

The output from this program is ten "Hello World" messages with the received number at the end:

```
..
Hello World5,
Hello World6,
Hello World7,
Hello World8,
...
```

Listing 6.2: Concurrent Hello World output

6.2 Pure EDF Multiprocessor Scheduling

In listing 6.3 the example code is presented. It consists of five procedures and two channels.

The `Main` procedure is the first to be called, and it is the only procedure that uses a `PAR` block. It calls the procedure `Proc1`, `Proc2`, `Proc0` and runs a `PAR FOR` block in parallel. For each statement in the `PAR` block a new process is generated. Of the `PAR` block in the `Main` procedure four processes is generated, all of them inheriting the parents deadline. Tree of the process runs the procedures calls, which are run as independent processes with their own deadlines.

The last process runs the `PAR FOR`, which runs ten `Proc3` calls in parallel, as with the other procedure calls they are run as independent processes with their specified deadline.

The `Main` creates in all twenty six processes, but only only half of them are independent processes running source code procedures, the other half is compiler generated processes that enable the first half to be created concurrently. These processes only execute one statement and are blocked until the procedures are complete. We therefor disregard them from now on.

The processes created operates with three different deadlines, the process running `Proc0` has the shortest deadline of `1Ms`, the processes running procedure `Proc3` has a deadline of `500Ms`, and process running `Proc1` and `Proc2` has a deadline of `2S`. Only based on their deadline the process running `Proc0` should finish first, then the processes running `Proc3` and last `Proc1` and `Proc2`.

However, in figure 6.1 we can see that this is not how the program is actually is run. This is due to the synchronous communication between `Proc0`, `Proc1` and `Proc2`. `Proc0` communicates with `Proc1` through channel `chan_b`, and

with `Proc2` through channel `chan_a`. `Proc0` is run concurrently with one of the processes running `Proc3`, but is blocked when sending on `chan_b`. It has to wait until `Proc1` is run and receives on `chan_b`. Due to the late deadline of `Proc1`, it run last and `Proc0` misses its deadline.

```
CHAN chan_a;
CHAN chan_b;

PROC Proc0 : 1Ms : {
    VAR a;
    SEQ {
        chan_b ! a;
        chan_a ? a;
    }
}

PROC Proc1 : 2S : {
    VAR a;
    SEQ {
        chan_b ? a;
    }
}

PROC Proc2 : 2S : {
    SEQ {
        chan_a ! 33;
    }
}

PROC Proc3 : 500Ms : {
    VAR a;
    SEQ {
        SEQ FOR i = 0 TO 10 : {
          Work;
        }
    }
}

PROC Main : 10S : {
    PAR {
```

```
36              PAR FOR i = 0 TO 10 : {
                    Proc3;
38              }
                Proc2;
40              Proc1;
                Proc0;
42          }
    }
```

Listing 6.3: Procedures depending on each other through synchronous channel communication

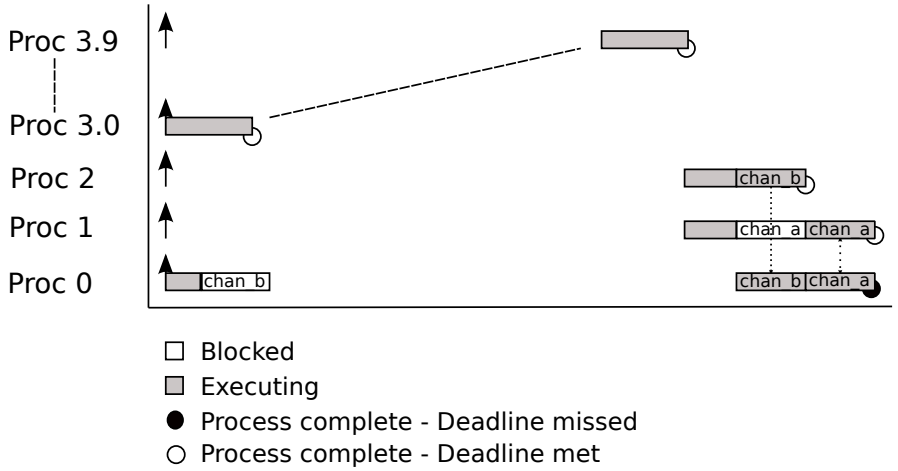

Figure 6.1: GANTT chart for channel dependency example without dependency analysis. Run on a system with two processors

6.3 EDF Multiprocessor Scheduling With Dependency Analysis

With dependency analysis enabled the dependencies between `Proc0`, `Proc1` and `Proc2` is mapped by the compiler. When the scheduler retrieves the process with lowest deadline from the ready queue a run-queue is filled with its *directly* and *indirectly* dependent processes. The worker threads than runs all the processes in this run-queue before retrieving the process with second shortest deadline. Let us take a look at how:

`Proc0` is the process with the lowest deadline in listing 6.3, and is the first process to be run after the main process. The procedure does two channel operations in sequence; it sends the variable `a` on channel `chan_b` and receives a message on channel `chan_a`.

The compiler maps that `chan_a` is used by `Proc0` and `Proc1`, and `chan_b` by `Proc2` and `Proc0`. By combining the information about the channel usage and the process that uses the channels a run queue for `Proc0` in created. In this case the run queue holds first `Proc1`, as `Proc0` first communicates over `chan_b`, and last `Proc2`.

Figure 6.2 shows how the processes are scheduled in a dual processor system. `Proc2` can then be run concurrently with `Proc0` and can communicate with it without having to wait. The end result is that every one of the processes reaches their deadline, unlike without dependency analysis where `Proc0` misses its deadline (figure 6.1).

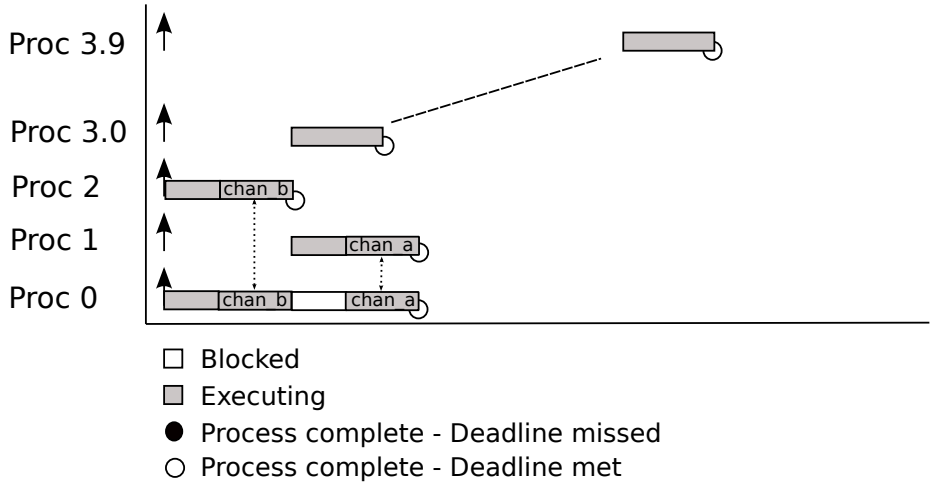

Figure 6.2: GANTT chart for channel dependency example with dependency analysis, run on as system with 2 CPUs

Example Evaluation

In chapter 2.5 two techniques to improve the schedulers efficiency was presented, interlocking protocols and batch scheduling. Our dependency based scheduling is a combination of these two techniques. Dependent processes

are grouped together as in batch scheduling, not to improve cache affinity, but to enable distribution among the workers to make sure they are working together to reach the earliest deadline. This is similar to the interlocking protocols, which tries to schedule tasks together to reach deadlines.

Interlocking protocols are there to prevent processes being blocked on shared resources. They operate solely at run-time. Usually they are based on processes locking a resource, and the other processes that needs the locked resource queues up. There are several advanced techniques to prevent "important" processes from waiting to long, such as PIP (Priority Inheritance Protocol) **??**. However, none of the common interlocking protocols uses compile time knowledge, which is also rare for batch scheduling (CCSP uses compile time knowledge). This is due to the fact that dependency analysis in "ordinary" languages is very complex, such as in C where pointers and pointers manipulation make it impossible for the compiler, at compile time map how a variable are shared.

Due to our language strict grammar, we can utilize compile time analysis to minimize deadline misses. However, it increases the complexity of the compiler, and the time used at compiling. For instance, for a for loop using an array of channels the compiler has to iterate over the whole loop to map all the channels used at compile time. If the number of iteration gets large enough, compilation can be slow.

The scheduling techniques also result in higher complexity in the run-time system, such as analysis done when retrieving a new process from the ready queue. In most cases the cost of this operation is minimal compared to the work done in the process.

6.4 Performance

We have performed two different performance tests on our language and three well known concurrent languages, occam(SPOC), GO, and Erlang. We have tested two features; process creation and synchronous communication. All the test programs can be found attached to this report.

The testes where performed on this system:

- OS: Ubuntu 9.10, Kernel Linux 2.6.31
- Hardware: 3.2 GiB memory, Intel®Core™2 Duo CPU 3.00 GHz
- Erlang: Erlang R13B01 (erts-5.7.2)

	Process creation (per process)	Commstime (per communication)
Erlang	3.0 μS	0.62μS
GO	4.5 μS	1 μS
occam (SPOC)	0.00155 μS	0.0039 μS
Our system	2.2 μS	0.125 μS

Table 6.1: Results from test of the different run-time systems

- GO: Commit hash 0be68ce1d89d from [9]
- occam: SPOC Version 1.3

Process Creation The first test was creating 20000 processes. A test inspired by Joe Armstrong benchmarks in [2]. He wanted to prove that an Erlang processes was light and was created fast. We replicated the test from [2] in GO, occam, and our own language. The tests where timed, and the results from the test can be found in table 6.1.

During design and implementation of our system we have not focused on optimizing mechanisms. It was more important to get the right functionality in place. Therefore, the results that was achieved was quite surprising. Our system was faster than both Erlang and GO, only occam was considerably faster (1400 faster) than our system. This is probably due to the work put into optimizing occam's run-time system. The reason that our system out performance GO and Erlang is probably due to that is is very simple and light. There is no complex register and allocation operations which has to be run for each process creation. The scheduler is small compared to GO and Erlang. In our run-time system there is therefor less operations involved in creating a process, than in the Erlang and GO.

Commstime The second test performed, was a performance test commonly used for CSP frameworks [33]. It tests the communication time by sending message in a ring. There are four processes in the ring: `Prefix`, `Delta`, `Succ` and `Consume`. `Prefix` starts communication by sending a value over the channel to delta. The messages is copied, and sent to `Consume` and `Succ`. `Succ` increments the value received by one, and sends it back to `Prefix`. It then passes the message through to `Delta` (The communication ring is illustrated in figure 6.3).

By timing each time `Consume` receives a message, we can determine how long a complete round of the ring takes.

A complete round requires four channel communications, so it also possible to determine how long an individual channel operation takes. By having multiple processes communicating it is easier to distribute the processes in a multiprocessor system, it also invokes context switches, which is then indirectly tested.

The design and implementation is based on the commstime test presented in [33].

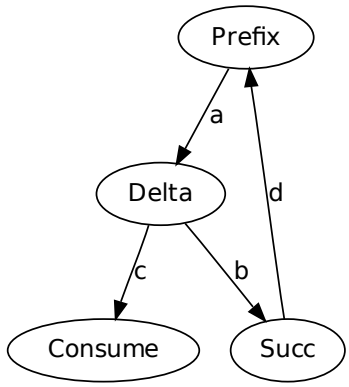

Figure 6.3: Process network for the communication benchmark

We implemented the commstime test in GO, occam(SPOC), Erlang[1] and our language. Listing 6.4 contains the commstime test implemented in our language.

We timed each round of the ring, and then found time pr. channel communication. In table 6.1 the results from the test is presented.

The result from the commstime test was a bit surprising. occam was clearly the fastest as it where in the first test, and again it is likely this is due to the highly optimized run-time system. But our system performed a channel operations faster than GO and Erlang.

[1]The Erlang test is a modified ring test from `http://projects.cs.kent.ac.uk/projects/kroc/svn/kroc/trunk/tests/ccsp-comparisons/`(May, 2010)

	Commstime 20000 rounds
Erlang	0.0136S
GO	0.038 S - 0.050S
occam (SPOC)	0.007S
Our system	0.050 S - 0.070S

Table 6.2: Results from running the complete commstime test

These results where intriguing. Again we think that the good results can be explained by that our system is simple and light.

The commstime benchmark was based on a implementation where 20000 rounds of the ring where made. We wanted to see what the total run-time of the system where, instead of single rounds. The results are presented in table 6.2. Based on the two first tests, we would expect that our systems would be faster also when the complete program was run. It has faster channel communication, and creates processes faster, but it did not. It is likely that our small and simple mechanisms perform better individually, then when the system is tested as a whole and all the unoptimized mechanisms makes its impact on performance.

```
CHAN chan_a;
CHAN chan_b;
CHAN chan_c;
CHAN chan_d;


PROC Prefix : 4S : {
    VAR a;
    SEQ {
        a = 1;
        SEQ FOR i = 0 TO 20000 : {
            chan_a ! a;
            chan_d ? a;
        }
    }
}

PROC Delta : 4S : {
    VAR a;
    SEQ FOR i = 0 TO 20000 : {
        chan_a ? a;
        chan_b ! a;
        chan_c ! a;
    }
}

PROC Succ : 4S : {
    VAR a;
    SEQ FOR i = 0 TO 20000 : {
        chan_b ? a;
        a = a + 1;
        chan_d ! a;
    }
}

PROC Consume : 4S : {
    VAR a;
    SEQ FOR i = 0 TO 20000 : {
        chan_c ? a;
```

```
    }
}

PROC Main : 2S : {
    PAR {
        Prefix;
        Delta;
        Succ;
        Consume;
    }
}
```

Listing 6.4: The process network test (commstime) implemented in our language

Summary

In this chapter we have demonstrated how dependency analysis affects the EDF scheduler. Results from two performance tests were then presented, and in both tests our system did surprisingly well.

Chapter 7

Missing Features in Language and Run-Time System

As mentioned earlier, in chapter 3, we have restricted the number of statements in our language severely. There has been a combination of reasons to disregard them; most of all that we did not see them necessary to demonstrate the real-time and concurrency principles in our language. However, we will in this chapter discuss which features necessary to add to make it a "usable" language, and possible designs for their implementation.

7.1 Missing Language Features

We will start with whats missing from the language, and then discuss some of the features missing from the run-time system.

Types

It is quite clear that it in necessary to add type declaration, so a variable can be something else than an `int`. The type declaration is put in front of the variable being declared, to specify the type of the variable. Some of the most common types in other languages are `int`, `double`, `char`, `string` and `bool`. Supporting types would require some changes in the compiler, and some changes in the run-time system. The grammar would have to be

extended to accept types, and require them to proceed variable and array declarations. The type would have to be mapped to the variable during compilation, which would be used in type checking and size calculation.

The run-time system would then check the type size, and allocate the correct size to the variable or array. Implementation should be relatively easy as the run-time system and compiler already uses a compiler generated size measurement. It would require that the variables no longer was stored in one array, but in separate pieces of memory. And the array array in the stack held pointers to the memory.

Channel declaration would then also have to require a type being specified for the data that could be sent over the channel. The type could also be used to allocate the size of the channel.

Procedure Arguments

There is currently no support for procedure arguments. This is something that would be natural to add. A procedure declaration would have to accept a list of variables it would take as arguments. The variables could then be used in the procedure body, as in a "normal" programming language such as C.

Because our compiler generates a C function from a procedure. Supporting arguments for procedures would not require a lot of work. When a procedure is called with arguments the data has to be copied on to the process stack, and later used in the C function.

If channels also where to be supported as arguments as well. It would be necessary to map who "owns" the channel to be able to map process dependencies. It would make the compile time analysis more complex, as it might be necessary to run parts of the procedure to determine ownership of a channel.

Conditional Statements

Our language does not support any conditional statements. This is something critical to add support for to make it usable language.

First support for boolean expression has to be added, which is used by as the condition in the statement. A typical example of such an expression is equal `==` or unequal `!=`, which can compare two variables. Then add support for the statements `IF/ELSE`, `WHILE` and `ALT`.

Adding support for boolean expression would require that we extend the number of expression supported in the grammar. The compiler can for most of the expressions directly translated to C expressions.

ALT The alternation(`ALT`) statement is a special occam statement (see [21] for details). It consists of several guards that all are evaluated concurrently. The first one to be true is run. A typical usage of `ALT` is when waiting on messages on multiple channels. When a message comes through, one of the guards becomes true and the belonging statements are executed. Such as in listing 7.1, where the process can receive on two channel, right and left and in both cases the message is to be sent out on the channel `stream`.

A possible run-time design for `ALT` statement could be to have one independent process waiting on the each branch. As soon as one becomes true and continues to execution, the rest of the waiting `ALT` processes are killed.

```
ALT
    left_chan ? packet {
        stream ! packet;
    }
    right_chan ? packet {
        stream ! packet;
    }
```

Listing 7.1: Waiting on channels and sends the incoming packet to the output packet (found in [21])

WHILE `WHILE` statements also needs support for boolean expressions. However, there are no branches in `WHILE` statements, so branch handling such as with `ALT` is not necessary. Our scheduler is is non-preemptive. This means that an eternal `WHILE` can occupy a single worker thread on its own. The `DeSchedule` keyword is therefor important to use in a `WHILE`, to not block other processes. Especially if the `WHILE` does not do any blocking channel communication where it can be descheduled in its own.

The potential of eternal `WHILE`s would require a new way for specifying deadlines, for a procedure with an eternal `WHILE` it is impossible to give a deadline. Hence, a `TIME` keyword can be a solution, such as in Martins Korsgaards Time/occam [19]. Or it could no longer be required to specify a

procedures deadline. Deadlines where voluntary and could for instance be specified for a `WHILE` iteration instead.

Except from this `WIHLE` loops are similar to the ones used in C, and the compiler can simply generate a C `while`.

```
PROC Test : {
    ...
    WHILE : 3S {
    ....
    }
}
```

Listing 7.2: Deadline specified for one iteration in an eternal WHILE loop

IF `IF` statements could be implemented quite similar to how they are in C. All the guards in a `IF` statement is checked sequentially, and only one of them can be true, if none of them is, the process stops. A `IF` could then generate a `if else` block, which deschedules on the last `else`.

One of the "problems" with conditional statements is that they make the program less deterministic. It can be difficult to determine which branch that will be run at compile time. Techniques such a branch prediction "choose"the branch the program is likely to take. Most branch prediction techniques is based on run-time statistics, the branch that has run the most earlier, is the one "chosen" (see [1] for more details).

```
chan_a ? ready;
IF :
    ready {
        ...
        chan_b ! b;
        ...
        }
   !ready  {
        ...
        chan_c ? c;
        ...
    }
```

Listing 7.3: IF/ELSE statement where the branches communicate over different channels

Due to the nondeterministic behavior of conditional statements the dependency analysis gets difficult. For instance, in listing 7.3 there is a `IF` statement with two branches: both guards checks the variable `ready`, which is assigned earlier by a incoming message on the channel `chan_a`. In the first branch a message is sent over channel `chan_b`, and in the second a message is received on channel `chan_c`. At compile time it is almost impossible to know which branch will be taken, hence impossible to know if it depends on the procedure communicate over `chan_b`, or on the procedure communicating over `chan_c`.

From a real-time perspective it is not efficiency that is the most critical criteria, but reaching deadlines. A solution to the get around using branch prediction in the example in 7.3 could be forking the process on the conditional statement and run both branches. Then there are two processes ready for communication on either `chan_b` or `chan_c`. The processes running the "wrong" branch will be terminated when the process running the other branch continues beyond the guard. Dependency wise the process running the code in listing 7.3 is dependent on both `chan_b` and `chan_c` (and off course `chan_a`).

Branch prediction on conditional statements is an interesting problem. It could be interesting to implement a solution in our run-time system. Although it is not essential to prove the real-time and concurrency principles in our language, it is rather a project on its own.

Functions

occam additionally supports function, which is a named value statement. A function can take an input argument and give an output. Functions are restricted compared to procedures, as they can not have any side effects. Therefor, the statements allowed in a function is restricted; no parallel statements, no input output (channel communication), and no alternations. occam functions are much like ordinary functions is other programming languages, so adding support for function declaration should be fairly simple. The compiler can simply translate a function to a C function, and a function call can be a ordinary C function call. Due to the strict grammatical rules, dependency is not a problem, as a function can not have any side effects.

7.2 Stronger Type Checking

To easier find errors in a program written in our language implementing type checking would be a good idea. However, type checking can be complex and time consuming task. As mentioned in the section (2.1.4) on semantic analysis types has to be checked with the operation they are used in. Luckily the BNF converter creates a syntactical and semantic check. However, it does not check grammatical rules such as; variables can only be written to by one of the statements in `PAR` block. Implementing stronger type checking will require some work, because we have had no focus on it so far.

But our compiler already has error handling, it will make implementation a bit easier. Because the compiler is written in Haskell, it is easy to throw error messages during compilation. This is made possible through Haskell monads (see [29] for more details on monads). The compilers error handling is for instance used to check that no more than two processes uses a channel. The monad "lies" above the parse tree, in abstraction level. So when an error occurs, the error is lifted up to the monad, the compiling stops, and returns an error message.

7.3 Asynchronous Communication

Support for buffered channel and asynchronous communication is something that also would be natural to add. As both the language and the run-time system already supports synchronous communication, only minor changes would be necessary to support asynchronous communication. The grammar has to be extended to define a buffer size on asynchronous channels. The run-time system each channel can have a buffer assigned to it. The size the buffer would then also depend on the type of data sent over the channel, as we discussed earlier.

Channel communication on buffered channels would be none blocking. On send, data is read into the buffer, and on receive it is read from the buffer. The channel could also have a counter to determine if the buffer is full or not, if it is send would be blocking. This design is similar to how buffered channels are handled in GO (see /src/pkg/runtime/chan.c:275 chanrecv and 231:chansend in the GO source [9]).

7.4 Garbage Collection

Currently variables can only be declared once in a process, which makes it impossible for the process to accumulate more memory. However, if `WHILE` and the other statements mentioned earlier where to be supported it would be natural to let the programmer create variables inside different scopes. To avoid programs to run out of memory. Such as in an eternal `WHILE` that for each iteration declares one variable. The garbage collector would analyze the stack in iterations, removing/"freeing" unused variables. There is several different garbage collection algorithms, GO uses a mark and sweep collector. It stops the program in specified intervals, marks the memory on the stack, each time a memory is accessed it is unmarked. Last the garbage collector removes/"frees" the cells which have not been unmarked since last iteration (src/pkg/runtime/mgc0.c gc:209 in [9]);

7.5 Missed Deadline Handling

Currently there is no handling for a process that misses its deadline, the scheduler blindly choses the process with the lowest specified deadline. It disregards what the time is when the process is retrieved, and if it possible to reach the deadline at all. Implementing a behavior that disregards all processes that has missed their deadline already is easy; it would just require that the deadline is compared to the current time. However, implementing an intelligent scheduler that sees that the process will not be able to reach is deadline, and handle it in a good way is not that easy.

Our proposal is that each procedure also has to specify a slack time, which is the amount of time it can miss its deadline with. This would improve the scheduler ability to decide if it should let the process run, even though it will not be able to reach its deadline.

The scheduling would also be based on a WCET (worst case execution time) analysis, which can be both unreliable and complex. If the scheduler where to interrupt a process that will not reach its deadline, we would also have to make the scheduler preemptive. This would require that a lot of work and most of the scheduler has to be rewritten.

Our suggestion is to simplify the problem, and say there are two types of process: a process where meeting the deadline is not critical and the process can complete, or reaching their deadline is critical and measurement have to be taken. When the scheduler is about to run a process it checks what

kind of process it is noncritical or critical. If it is a critical process it starts a timer concurrently with the process, which timeout at the process deadline. The timer holds a pointer to the process, and changes a flag in the pointer which triggers the "deadline not met" part of the process.

Because our system runs on top of an OS, it can never guarantee hard real-time requirements. A systems running on our system has to be of the type where a deadline miss is not very critical, but hopefully can the solution such as the one discussed above improve usability for semi-critical process a bit.

Summary

In this section what we consider as missing features in the language was presented. One of the major missing language features is conditional statement. We presented a possible design for implementation for it and other missing features.

Chapter 8

Discussion

In this section we will discuss the major design choices met during construction of the experimental language and the run-time system.

8.1 The Language Grammar

We chose to base our language on occam, which is a concurrency oriented language. As we have presented earlier there are several concurrent languages around. We could have based our language on any of these, or a C like grammar with some special keywords.

What makes occam so special; is its focus on fine grain concurrency. With occam the developer is able to explicitly express how statements are to be run. None of the other languages presented has this ability. The grammar is also very strict due to this fine grained concurrency. With so many processes running concurrently it would be impossible to write an efficient program with shared data. Imagine a `PAR FOR` loop running 20000 iterations in parallel all sharing sharing one variable and all of them has to wait on its turn to read/write. It would just not work.

As we have seen, having strict rules makes dependency analysis easier, as there is only a number of specific ways processes can depend on each other. One of our initial goals was to utilize compile time knowledge about dependencies in the scheduler. We saw that only trough strictly limiting the language was this a plausible task within the timespan of the project. occam was the only language, which was so limited in its original grammar. We could have chosen a language as C, and put hard strains on how it could

be used, but then it would no longer look like C. A C programmer would not be able to program naturally. Although the grammar we have specified is much more limited than occam's, it still keeps the essence of occam. By having fine grained parallelism, channel communication and independent procedures.

8.2 How to Distribute Processes Among Processors

There where many ways processes could be implemented and run in our run-time system. As we have seen in the chapter on parallelism (see section 2.3) there are several parallel design patterns that could be utilized.

The main purpose of the run-time was to schedule processes and run them as efficient as possible on the processors. The easiest solution would be to run each processor on a separate OS-thread, and control scheduling through OS-signals. However, we knew that OS-thread was slow and OS-thread operations came with a large amount of overhead (see [17]). It would also be difficult to control scheduling exactly like we wanted. When we examined the run-time system of the other concurrent oriented languages, we saw that they all tried to minimize the amount of OS-thread used. There where different approaches and different designs, but they all clearly tried to avoid having to many OS-threads.

Most of the run-time system used the master/worker pattern, where there was a set of worker threads executing the processes and doing other run-time tasks. The master/worker pattern was implemented using mainly two designs: one thread pr. processor that did all the work in the run-time system or a pool of worker threads, where workers were "called" to do different work. In the pool solution the number of worker was not fixed, and more worker threads could be created dynamically. This is to reduce startup/shutdown cost of threads.

With one worker thread pr. processor the number of OS-threads was kept at a minimum, it also requires less mechanisms to handle the worker threads. We therefor chose to fix a thread to a specific processor through Linux mechanisms. Keeping the number of context switches down.

We can see from the results in chapter 6 that our design was comparable with other run-time systems. Clearly showing that with a simple design has its benefits.

8.3 Global or Partitioned Run Queue

When choosing a multiprocessor scheduler design, there two different approaches, global or partitioned run queue. As presented earlier global vs partitioned scheduler is one of the major debates in the scheduler community. Both of them have advantages over each other (see section 2.5).

Partitioned scheduler is scalable; it focuses on minimal shared data between the processors. Because partitioned scheduler has one run queue pr. processor, none of them has to wait to retrieve a task from a shared run queue. And in the global scheduler the run queue is shared between all the processes. However, a real-time system is rarely run on systems on with a huge number of processors. Usually they will be run on standard or specialized platforms with something like one to sixteen processors, and not on server farms with 1024 nodes.

We ended up choosing a global scheme mainly due to dependency analysis. Defining dependency among distributed processes would be complex. It would require that dependencies was mapped across different run-queues. The processes the process with the earliest deadline is dependent has to be run first of the processes in the local run queue, even if there is longer time to their deadline then the "local" tasks.

As presented, the global scheduler design in our run-time system uses an separate run queue for the dependent processes. This run queue is then used as the "primary" run queue as long as there is processes in it, and when it its empty it uses the normal ready run queue. This is a very simple solution to the problem that works very well with a low number of workers, such as in our system.

8.4 Preemptive or Non-Preemptive

Another design choice was if we where to create a preemptive or a non-preemptive scheduler. The choice was based on that the scheduler was a pure EDF scheduler. Where there are really one scenario where a task can be preempted, and that is when a new task arrives with an even shorter deadline than the currently running. However, in our system where there are no conditional statements it in unlikely that such a process is created. A preemptive scheduler is also much complex than a non-preemptive scheduler, as a process can be interrupted at any time. It requires that the process is able to save its state before it is descheduled. This means that we would no

longer be able to use Duff's device, and it would be necessary to implement a traditional stack. Most of the concurrent run-time systems that we have presented have also a non-preemptive scheduler.

We chose implement a non-preemptive scheduler because it would require less from our run-time system, and it was not necessary to demonstrate the real-time and concurrency principles of our language.

8.5 Dependency Analysis

One of our initial goals was to utilize dependency analysis to improve scheduling. As we presented earlier in the section on multiprocessor schedulers (see sec 2.5.4) dependent processes can be grouped together and run on the same processor, to improve cache reuse. This approach will increase the efficiency of the scheduler as it can better utilize the hardware.

But for a real-time scheduler increasing efficiency is not the primary goal, it is to reduce deadline misses. So we wanted to see if we could utilize dependency analysis to decrease deadline misses.

We concluded with that the best way to utilize dependency was to combine interlocking with batch scheduling, and group dependent processes together and run them concurrently. This let the worker threads then work together to reach the deadline of the running process. As we demonstrated in chapter 6 this approach help processes reach their deadline, when they depend on other processes.

The cost of this approach is that it increases complexity and overhead for each time a new process is retrieved from the ready queue. But as explained earlier the work at run-time is minimal, most of the work is done at compile time and when a process is created.

Due to the focus on dependency analysis we have restricted our language in many ways. This has been necessary to make the dependency analysis possible. The more the complex the language is the more complex the dependency analysis is. Through limiting the language we were able to demonstrate how dependency analysis could be utilized by a multiprocessor real-time scheduler.

8.6 Pure EDF Scheduler

Another major design choice was what which real-time scheduler to use. Our choice was based on the scheduler survey done in [17]. Most of the scheduler discussed variations of the global EDF-scheduler that tries to avoid Dhall's effect, which limits the performance of multiprocessor EDF. Dhall's effect only occurs when there exists blocking heavy processes. Due to the structure of our language the number of blocking heavy processes is less than in an "ordinary" language. occam is focused on fine-grained concurrency, and hence generates a large number of light processes, instead of a few intensive tasks.

If we where to implement an modified EDF scheduler a WCET analysis has to be performed on all the tasks, to know if they met the additional constraints of the modified EDF schedulers (for more details on modified EDF please see [17]). We chose to only implement a simple pure multiprocessor EDF due to its simple and rigid design. The scheduler can also later be extended to support different modified multi processor EDF schedulers.

Summary

In this section we have discussed some of our design choices, and what was achieved.

Chapter 9

Conclusion

Our language is a concurrency oriented real-time language, built to run on multiprocessor systems. We have shown that the language grammar focuses on statements generating fine-grained parallelism, was designed based on occam. They main features of our language is parallel statement blocks, and synchronous channel communication.

The processes created in the language are run on a run-time system built on the master/worker principle. This is to reduce the amount of OS added overhead, and makes us able to stay in complete control of process scheduler. The scheduler in the run-time system is a multiprocessor EDF scheduler, where there are one OS-thread per. processor executing the processes.

Performance comparison with other concurrent oriented languages, show that our implementation is comparable with them. Although, optimization and run-time costs have been a secondary priority to deadline misses. The major priority was to try to limit deadline misses through dependency analysis.

This was achieved through doing compile time analysis of channel communication, the compiler than generated lists over processors communicating with each other. The information found by the compiler is utilized by the scheduler to let all the workers work towards a common deadline. Each worker executes process that the process with the earliest deadline is dependent on. Minimizing the time the process with earliest deadline has to wait on messages from other processes. We illustrated such a scenario where the earliest deadline process was dependent on two processes. Without dependency analysis it missed its deadline due to the processes running in between it and the dependent processes. However, with dependency anal-

ysis it reached its deadline, due to the dependent processes was running concurrently with it.

Our language and run-time system show how compile time knowledge can be utilized by a multiprocessor real-time scheduler. Although, our language is limited it proves the design principles and can be a foundation for further development.

9.1 Further Work

Earlier we described what we consider as missing features in the language and run-time system. The list is long, and even creating some of the basic parts can be time consuming. The most pressing and interesting aspect would be to explore, how conditional statements can be handled. We already proposed one possible design, but more research has to be done before implementing a solution. Clearly dependency analysis and branch prediction go hand in hand in this case and are closely linked together.

Other aspects we found interesting is Erlang's robustness, and error handling. Such as that a process can register as dependent on other processes. If a process then crashes, all the dependent process gets notified. From a real-time aspect this is interesting, and it should be possible to implement such a solution in our run-time system.

Appendix A

BNF grammar

⟨*ProgramSpec*⟩ ::= ⟨*ListGlobDecl*⟩

⟨*GlobDecl*⟩ ::= `PROC` ⟨*CapIdent*⟩ `:` ⟨*DeadLine*⟩ `:` ⟨*SingleBlock*⟩
| ⟨*Chan*⟩ `;`

⟨*ListGlobDecl*⟩ ::= ϵ
| ⟨*GlobDecl*⟩ ⟨*ListGlobDecl*⟩

⟨*SingleBlock*⟩ ::= { ⟨*ListVar*⟩ ⟨*StmtCompound*⟩ }

⟨*StmtCompound*⟩ ::= ⟨*Order*⟩ `FOR` ⟨*LowIdent*⟩ ⟨*AssignOp*⟩ ⟨*Integer*⟩ `TO` ⟨*Integer*⟩ `:` ⟨*StmtBlock*⟩
| ⟨*Order*⟩ ⟨*StmtBlock*⟩

⟨*ListStmtCompound*⟩ ::= ϵ
| ⟨*StmtCompound*⟩ ⟨*ListStmtCompound*⟩

⟨*StmtBlock*⟩ ::= { ⟨*ListStmt*⟩ }

⟨*Order*⟩ ::= `PAR`
| `SEQ`

⟨*ListOrder*⟩ ::= ϵ
| ⟨*Order*⟩ ⟨*ListOrder*⟩

⟨*DeadLine*⟩ ::= ⟨*Integer*⟩ `S`
| ⟨*Integer*⟩ `Ms`
| ⟨*Integer*⟩ `Us`
| ⟨*Integer*⟩ `Ns`

⟨*ListDeadLine*⟩ ::= ϵ
| ⟨*DeadLine*⟩ ⟨*ListDeadLine*⟩

⟨*Stmt*⟩ ::= ⟨*LowIdent*⟩ ⟨*AssignOp*⟩ ⟨*Expr*⟩ `;`
| ⟨*LowIdent*⟩ ⟨*Array*⟩ ⟨*AssignOp*⟩ ⟨*Expr*⟩ `;`
| `DeSchedule ;`
| ⟨*CapIdent*⟩ `;`
| ⟨*Expr*⟩ `;`
| `Work ;`
| ⟨*StmtCompound*⟩
| `Print` ⟨*String*⟩ ⟨*ListExpr*⟩
| ⟨*ChanCom*⟩ `;`
| ⟨*Stmt*⟩ `;`

⟨*ListStmt*⟩ ::= ϵ
| ⟨*Stmt*⟩ ⟨*ListStmt*⟩

⟨*Chan*⟩ ::= `CHAN` ⟨*LowIdent*⟩
| `CHAN` ⟨*LowIdent*⟩ ⟨*Array*⟩

⟨*ListChan*⟩ ::= ϵ
| ⟨*Chan*⟩ `;` ⟨*ListChan*⟩

⟨*Var*⟩ ::= `VAR` ⟨*LowIdent*⟩
| `VAR` ⟨*LowIdent*⟩ ⟨*Array*⟩

⟨*ListVar*⟩ ::= ϵ
| ⟨*Var*⟩ `;` ⟨*ListVar*⟩

⟨*Array*⟩ ::= `[` ⟨*Expr*⟩ `]`

⟨*ListArray*⟩ ::= ϵ
| ⟨*Array*⟩ ⟨*ListArray*⟩

⟨*AssignOp*⟩ ::= `=`

⟨*ChanCom*⟩ ::= ⟨*ChanExpr*⟩ `!` ⟨*Expr*⟩
| ⟨*ChanExpr*⟩ `?` ⟨*Expr*⟩

⟨*ListChanCom*⟩ ::= ϵ
| ⟨*ChanCom*⟩ `;` ⟨*ListChanCom*⟩

⟨*ChanExpr*⟩ ::= ⟨*LowIdent*⟩
| ⟨*LowIdent*⟩ `[` ⟨*Expr*⟩ `]`

⟨*ListChanExpr*⟩ ::= ϵ
| ⟨*ChanExpr*⟩ `;` ⟨*ListChanExpr*⟩

⟨*Expr1*⟩ ::= ⟨*Expr1*⟩ + ⟨*Expr2*⟩
| ⟨*Expr1*⟩ − ⟨*Expr2*⟩
| ⟨*Expr2*⟩ `[` ⟨*Expr2*⟩ `]`
| ⟨*Expr2*⟩

⟨*Expr4*⟩ ::= ⟨*Integer*⟩
| ⟨*Expr5*⟩

⟨*Expr5*⟩ ::= ⟨*LowIdent*⟩
| `(` ⟨*Expr*⟩ `)`

⟨*ListExpr*⟩ ::= ϵ
| ⟨*Expr*⟩ `;` ⟨*ListExpr*⟩
| ϵ
| ⟨*Expr*⟩
| ⟨*Expr*⟩ `,` ⟨*ListExpr*⟩

⟨*Expr*⟩ ::= ⟨*Expr1*⟩

⟨*Expr2*⟩ ::= ⟨*Expr3*⟩

⟨*Expr3*⟩ ::= ⟨*Expr4*⟩

Bibliography

[1] *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann, September 2006.

[2] *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

[3] Jibu in depth. `http://www.axon7.com/flx/home/jibu_in_depth/`, April 2010.

[4] jibu source. `http://www.axon7.com/flx/downloads/`, April 2010.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2 edition, August 2006.

[6] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[7] J.L Armstrong, S.R Virding B.O Däcke and, and M.C. Williams. Implementing a functional language for highly parallel real time applications. *SETSS*, 1992.

[8] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[9] The Go Authors. Google go compiler. `https://go.googlecode.com/hg`, January 2010.

[10] The Go Authors. Language design faq. `http://golang.org/doc/go_lang_faq.html`, April 2010. Why build concurrency on the ideas of CSP?

[11] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 2001.

[12] corbet. Trees ii: red-black trees. `http://lwn.net/Articles/184495/`, June 2006.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science / Engineering / Math, 2nd edition, December 2003.

[14] Russ Cox. Function generation. `http://groups.google.com/group/golang-nuts/browse_thread/thread/880511ed74498e24/f99f2f29a1166f0a?lnk=gst&q=jon+tore#f99f2f29a1166f0a`, 2010.

[15] S.K Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, pages 127–240, 1978.

[16] Dario Faggioli, Giuseppe Lipari, and Tommaso Cicinotta. An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel. *OSPERT*, 2008.

[17] Jon Tore Hafstad. Real time schedulers for multiprocessor systems survey, and the linux scheduler simulator. 2009.

[18] C. A. R. Hoare. Communicating sequential proocesses. *Commun. ACM*, 21(8):666–677, August 1978.

[19] Martin Korsgaard. Introducing time driven programming using csp/occam and wcet estimates. august 2007.

[20] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992.

[21] SGS-THOMSON Microelectronics Limited. occam 2.1 reference manual, 1989.

[22] C.L Liu and J.W Layland. Scheduling algorithms for multiprogramming in a hard-real-time enviroment. *JACM 20.1*, 1973.

[23] Kenneth Lundin. Inside the erlang vm with focus on smp. *Erlang user conference*, 2008.

[24] Aarne Ranta Markus Forsberg. The labelled bnf grammar formalism. *Chalmers University of Technology and the University of Gothenburg*, 2005.

[25] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[26] C. Meenderinck and B. Juurlink. (When) Will CMPs Hit the Power Wall? In *Euro-Par 2008 Workshops-Parallel Processing*, pages 184–193. Springer, 2009.

[27] James Moores. Csp - a portable csp-based run-time system supporting c and occam. 1999.

[28] Pedro Mejia Alvarez Omar U. Pereia Zapata. Edf and rm multiprocessor scheduling, algorithms: Survey and performance evaluation. 2007.

[29] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1 edition, December 2008.

[30] Rob Pike. The go programming language. `http://www.youtube.com/watch?v=rKnDgT73v8s`, april 2009.

[31] Carl G. Ritson, Adam T. Sampson, and Frederick R. Barnes. Multicore scheduling for lightweight communicating processes. In *COORDINATION '09: Proceedings of the 11th International Conference on Coordination Models and Languages*, pages 163–183, 2009.

[32] Yuan Shi. Reevaluating Amdahl's Law and Gustafson's Law. *Computer Sciences Department, Temple University (MS: 38-24)*, 1996.

[33] Bernhard Sputh, Oliver Faust, and Alastair R. Allen. Portable CSP Based Design for Embedded Multi-Core Systems, sep 2006.

[34] Simon Tatham. Coroutines in c. `http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`, March 2008.

[35] Tim Teitelbaum. Types and type-checking. `http://www.cs.cornell.edu/courses/cs412/2007sp/schedule.html`, 2007.

[36] Unknown. Prime numbers. `http://www.haskell.org/haskellwiki/Prime_numbers#The_Classic_Turner.27s_Sieve`, May 2010.

[37] Ulf Wiger. Erlang scheduler: What does it do? `http://www.erlang.org/pipermail/erlang-questions/2001-April/003131.html`, 2001.