Roko: Balancing Performance and Usability in Coarse-Grain Parallelization

by

Cedomir Segulja

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science Graduate Department of Electrical and Computer Engineering University of Toronto

Copyright © 2010 by Cedomir Segulja
Abstract

Roko: Balancing Performance and Usability in Coarse-Grain Parallelization

Cedomir Segulja
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2010

We present Roko, a system that allows parallelization of sequential C codes with a modest user intervention. The user exposes parallelism at the function level by annotating the code with pragmas. Roko defines only two pragmas: the parallel pragma is used to denote function calls that will be executed asynchronously, and the exposed pragma is used to describe data usage of the marked function calls.

Architecturally, Roko consists of three components: a compiler that analyzes pragmas, a software environment that spreads the execution over multiple processors, and a hardware support that implements a novel synchronization scheme, versioning. We have designed, implemented and evaluated an FPGA-based prototype of Roko. Our experimental evaluation shows: (i) that few simple pragmas are all that is needed to expose parallelism in benchmark applications and (ii) that Roko can deliver good performance in terms of application speedup.
Acknowledgements

I am grateful to everyone who has made this thesis possible. Firstly, I owe my gratitude to my supervisor, Prof. Tarek S. Abdelrahman, for giving me the opportunity to study at the University of Toronto and for his continuous guidance throughout the course of this work. Numerous discussions with him helped me to think about the encountered problems in a different way and gave me the motivation to “push it further”.

The classes that I have attended during my master studies were an invaluable asset for this work. Hence, I would like to thank to professors Christiana Amza, Paul Chow, Ashvin Goel, Greg Steffan, Andreas Moshovos, and Jianwen Zhu for passing some of their extensive knowledge on to me. I am indebted to many of my colleagues for helping me during the course of this work. Utku Aydonat, Borys Bradel, Mihai Burcea, Davor Capalija, Adam Czajkowski, David Han and Ivan Matosevic have always find time to answer my, often silly, questions. Special thanks to Martin Labrecque and Ivan Matosevic who proof read this thesis and their comments greatly improved this work.

I thank Prof. Sinisa Srbljic for his support and encouragement to pursue my graduate studies at the University of Toronto. After moving a half way across the globe, one finds himself a bit lost. Davor Capalija, Ivan Matosevic and Franjo Plavec made me feel less lost, and for that I am truly grateful.

My deepest gratitude goes to my family for their unflagging love and support throughout my life. I simply cannot ask for more than what they have given me.
## Contents

1 Introduction
   1.1 Thesis Contributions ............................................. 4
   1.2 Thesis Organization .............................................. 4

2 An Overview of Roko
   2.1 Asynchronous Execution ........................................... 5
   2.2 Programming Paradigm ............................................. 6
      2.2.1 Merge Sort Example ........................................ 6
   2.3 Execution Model .................................................. 10
   2.4 System Architecture ............................................. 12

3 The Roko Pragmas
   3.1 Exposed Accesses of a Function ................................. 14
   3.2 The parallel pragma ............................................ 16
   3.3 The exposed pragma .............................................. 17
      3.3.1 Describing Simple Accesses ............................... 18
      3.3.2 Describing Recursive Accesses ............................ 20
      3.3.3 Describing Accesses to Array Regions .................. 22
      3.3.4 Indirect Accesses .......................................... 23
   3.4 Sufficiency of Exposed Accesses .............................. 24
## 4 Versioning

4.1 An Overview of Versioning ........................................ 26

4.2 Fixed-granularity Versioning ...................................... 31

4.2.1 Structures .................................................. 32

4.2.2 Algorithms .................................................. 34

4.2.3 Objects with Non-Static Storage Duration .................... 40

4.3 Variable-granularity Versioning .................................. 42

4.3.1 Structures .................................................. 47

4.3.2 Algorithms .................................................. 49

4.3.3 Concurrent Readers ......................................... 57

4.4 Guarantees Offered by Versioning ................................ 58

## 5 Hardware Support for Versioning

5.1 Overview of Hardware Support .................................... 59

5.1.1 The Main Mechanisms ....................................... 61

5.1.2 The Representation of Version Numbers ...................... 64

5.1.3 Processor Core Modifications ............................... 65

5.2 Filtering Memory Accesses ....................................... 66

5.2.1 LVT Design Considerations ................................ 67

5.2.2 Monitored Set .............................................. 68

5.2.3 Maintaining the Monitored Set .............................. 70

5.2.4 CBF Granularity ............................................ 71

5.3 Hardware support for the Generate Algorithm .................. 72

5.3.1 Operation .................................................. 72

5.3.2 An Example ............................................... 74

5.4 Instruction Set Architecture Extensions ......................... 78

5.4.1 Versioning Instruction Latencies ............................ 79
6 The Roko Kernel and Compiler

6.1 The Roko Kernel .................................................. 82
   6.1.1 Task Representation ........................................ 83
   6.1.2 Scheduling Policy ........................................... 84
   6.1.3 Task Creation and Termination ............................... 87

6.2 The Roko Compiler ................................................ 88
   6.2.1 Analysis Phase ............................................... 88
   6.2.2 Code Transformation Phase ................................. 90

7 Experimental Evaluation .......................................... 95

7.1 Platform .......................................................... 95
   7.1.1 Hardware Platform .......................................... 95
   7.1.2 Software Platform .......................................... 101

7.2 Evaluation Methodology ......................................... 101

7.3 Benchmarks ...................................................... 102
   7.3.1 Matrix Multiplication ...................................... 103
   7.3.2 Search ....................................................... 104
   7.3.3 Water ....................................................... 105

7.4 Results .......................................................... 107
   7.4.1 Roko Kernel Overheads .................................... 107
   7.4.2 Versioning Instructions Latencies .......................... 109
   7.4.3 Applications ................................................. 111
   7.4.4 Filtering Memory Accesses ................................ 114

8 Related Work ...................................................... 118

8.1 Task-level parallelism ............................................ 118
   8.1.1 Jade, pTask, zJava ......................................... 118
   8.1.2 Other Related Systems ....................................... 121
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>Memory Monitoring</td>
<td>123</td>
</tr>
<tr>
<td>8.3</td>
<td>Numbering Schemes</td>
<td>124</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions and Future Work</td>
<td>126</td>
</tr>
<tr>
<td>9.1</td>
<td>Future Work</td>
<td>127</td>
</tr>
<tr>
<td>A</td>
<td>Correctness of Versioning</td>
<td>129</td>
</tr>
<tr>
<td>A.1</td>
<td>Task Graph and Chained Tasks</td>
<td>129</td>
</tr>
<tr>
<td>A.2</td>
<td>Proof of Theorem A.1.1</td>
<td>130</td>
</tr>
<tr>
<td>A.3</td>
<td>Proof of Theorem A.1.2</td>
<td>137</td>
</tr>
<tr>
<td>B</td>
<td>Bloom filters</td>
<td>141</td>
</tr>
<tr>
<td>B.1</td>
<td>Hardware implementation</td>
<td>142</td>
</tr>
<tr>
<td>C</td>
<td>Hardware implementation of the LVT</td>
<td>144</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>146</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Elements of an LVT entry in fixed-granularity versioning</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Elements of an LVT entry in variable-granularity versioning</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>ISA extensions</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Expected latencies of the versioning instructions</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>FPGA device and board characteristics</td>
<td>96</td>
</tr>
<tr>
<td>7.2</td>
<td>Design tools</td>
<td>96</td>
</tr>
<tr>
<td>7.3</td>
<td>SMP system characteristics</td>
<td>97</td>
</tr>
<tr>
<td>7.4</td>
<td>Processor characteristics</td>
<td>98</td>
</tr>
<tr>
<td>7.5</td>
<td>Roko parameters</td>
<td>100</td>
</tr>
<tr>
<td>7.6</td>
<td>Roko memory requirements</td>
<td>100</td>
</tr>
<tr>
<td>7.7</td>
<td>Application characteristics</td>
<td>111</td>
</tr>
<tr>
<td>7.8</td>
<td>Characteristics of the parallel execution</td>
<td>111</td>
</tr>
<tr>
<td>7.9</td>
<td>Overhead analysis</td>
<td>113</td>
</tr>
<tr>
<td>7.10</td>
<td>Average LVT and GVT access times</td>
<td>116</td>
</tr>
<tr>
<td>C.1</td>
<td>Operations of the LVT</td>
<td>145</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Merge sort ................................................................. 7
2.2 Merge sort annotated with Roko’s pragmas ................................. 9
2.3 The task-phase graph for the merge sort example ......................... 11
2.4 Roko architecture and compilation flow .................................. 12

3.1 The grammar of the exposed pragma ..................................... 18
3.2 Example usage of the exposed pragma ..................................... 19
3.3 An example illustrating recursive pointer accesses and the usage of repeat operators ...................................................... 20
3.4 An example illustrating the usage of repeat operators and alternatives for a tree traversal ...................................................... 21
3.5 Matrix multiplication ......................................................... 22
3.6 An example of indirect accesses ............................................. 23

4.1 A simple program used to illustrate versioning ............................... 28
4.2 The structures used in fixed-granularity versioning .......................... 32
4.3 The acquire algorithm in fixed-granularity versioning ...................... 35
4.4 The release algorithm in fixed-granularity versioning ...................... 36
4.5 The generate algorithm in fixed-granularity versioning ...................... 37
4.6 Versioning in the presence of non-static objects ............................. 40
4.7 A simple program used to illustrate the impact of versioning granularity . 43
4.8 Comparison of the parallel executions of the example program from Figure 4.7 under word and double-word granularity versioning ........................................... 44
4.9 The parallel executions of the example program from Figure 4.7 under variable-granularity versioning ................................................................. 46
4.10 The acquire algorithm in variable-granularity versioning ........................... 49
4.11 The release algorithm in variable-granularity versioning ............................ 50
4.12 The generate algorithm in variable-granularity versioning .......................... 51
4.13 The generate algorithm and fragmentation .................................................. 53
4.14 The generate algorithm when fragmentation does not occur ....................... 55
4.15 The role of the usable element of an LVT entry ........................................ 57
4.16 Multiple concurrent readers ..................................................................... 58

5.1 The main components of the hardware design ............................................ 60
5.2 The execution of a memory access ............................................................... 62
5.3 Phases of the generate algorithm implemented in hardware ....................... 72
5.4 The states of structures and registers during the execution of the matching operation ........................................................................................................... 75
5.5 The final state of the LVT of the parent and child task after executing the matching operation ................................................................. 77

6.1 Memory layout of the metadata of a task .................................................... 84
6.2 State transition diagram of a task’s lifetime ................................................ 86
6.3 A tree built by the Roko Compiler used to extract indirect accesses ............ 88
6.4 The generated callback function and the input structure for sort function of the merge sort example ................................................................. 90
6.5 The generated taskify function for sort function of the merge sort example 92
6.6 The function sort of the merge sort example after transforming functions call sites .................................................................................................. 93
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>Code transformation for a task-function that returns a value</td>
<td>94</td>
</tr>
<tr>
<td>7.1</td>
<td>The diagram of the implemented SMP system</td>
<td>97</td>
</tr>
<tr>
<td>7.2</td>
<td>Matrix multiplication code fragment</td>
<td>103</td>
</tr>
<tr>
<td>7.3</td>
<td>Search code fragment</td>
<td>104</td>
</tr>
<tr>
<td>7.4</td>
<td>Water code fragment</td>
<td>106</td>
</tr>
<tr>
<td>7.5</td>
<td>Varying task granularity for a fixed number of tasks</td>
<td>107</td>
</tr>
<tr>
<td>7.6</td>
<td>The per-task breakdown of overheads</td>
<td>108</td>
</tr>
<tr>
<td>7.7</td>
<td>Varying number of tasks for a fixed task granularity</td>
<td>109</td>
</tr>
<tr>
<td>7.8</td>
<td>The measured latencies of the versioning instructions for a fixed region size</td>
<td>110</td>
</tr>
<tr>
<td>7.9</td>
<td>The measured latencies of the versioning instructions for a fixed number of regions</td>
<td>110</td>
</tr>
<tr>
<td>7.10</td>
<td>The speedup of the benchmarks</td>
<td>112</td>
</tr>
<tr>
<td>7.11</td>
<td>The breakdown of memory accesses</td>
<td>113</td>
</tr>
<tr>
<td>7.12</td>
<td>The breakdown of LVT accesses</td>
<td>115</td>
</tr>
<tr>
<td>7.13</td>
<td>The speedup of the matrix multiplication benchmark on 4 processors for different matrix sizes</td>
<td>117</td>
</tr>
<tr>
<td>7.14</td>
<td>The speedup of the benchmarks run on the hardware with no CBF</td>
<td>117</td>
</tr>
<tr>
<td>B.1</td>
<td>The architecture of a CBF Block</td>
<td>143</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

For the past decades, the exponential growth of the number of transistors on a chip (as predicted by Moore [51]), resulted in an exponential growth of microprocessor performance [56]. This was made possible by increasing the clock frequency and exploiting instruction-level parallelism (ILP) via instruction pipelining, multiple function units and out-of-order execution [40]. Unfortunately, this trend has slowed during the last few years, mainly due to hitting the “power wall” [32]. Consequently, the leading processor manufactures announced in 2005 that their high performance microprocessors would henceforth rely on multiple cores [6]. Today, chip-multiprocessors (CMP) have become the industry standard.

The shift towards this level of parallelism is posing a big challenge to the computer research community, which needs to explore how to harness the power of parallel systems without significantly changing programming approaches. Ideally, programmers would continue writing their codes as they have been doing for many years, using traditional languages such as C/C++. Unfortunately, automatic parallelization has had a limited success and has been applicable only to scientific, data-parallel codes, while the parallelization of general programs remains an unsolved problem up to now. Hence, many researchers have proposed various programming models for parallel systems.
Parallel programming languages (e.g., Linda \cite{18} and Strand \cite{33}) frequently permit elegant formulation of certain classes of parallel algorithms \cite{9}. However, writing even a simple program requires learning a completely new language, with semantic rules very different than the ones that most programmers are used to. Additionally, parallel languages make porting of existing sequential codes difficult, since the parallel versions have to be written from scratch. Proposals for using existing, but not widely used paradigms, such as functional (e.g., Haskell \cite{41}) or logical (e.g., Prolog \cite{37}) languages, that are more suitable for parallelization suffer from the similar limitations.

On the other hand, library-based approaches (e.g., Pthreads \cite{53} and MPI \cite{57}), offer well known environments (e.g., C/C++ and Fortran) and can be used for both writing new codes and porting existing applications to CMPs. However, the problem with this style of programming is that it puts a lot of, often unnecessary, burden on the programmer. For instance, when using Pthreads, functions need to have a specific signature in order to be suitable for execution by threads. Clearly, compiler assistance could be used to alleviate these kinds of issues.

Language extensions, either through compiler directives (e.g., OpenMP \cite{23}) or by introducing new constructs (e.g., Cilk \cite{35} and Compositional C++ \cite{22}), work in the direction of easing the use of parallel systems. Nevertheless, they are still a long way from making parallel programming easy. With the current frameworks, after exposing parallelism present in some part of the program, the programmer must explicitly ensure that memory accesses are properly synchronized using locks, barriers, or more recently transactions \cite{75}. Although this requires inserting just a few lines of code, it is still a daunting task since the programmer needs to reason about all the other parts of the program that may be accessing the same or related shared data and potentially insert synchronization there as well \cite{19}. Furthermore, the currently employed non-deterministic execution model makes bugs harder to identify and reproduce \cite{28, 55, 76}. This issue is exacerbated by the fact that parallel codes have more bugs (race conditions, livelocks, deadlocks) than equivalent sequential programs \cite{34}.
In this thesis, we advocate a different approach. We propose a model that is based on the language extensions, but that requires lesser user intervention. Our approach makes it possible to expose parallelism of a part of the code without having a global knowledge of the entire program or modifying other parts of the code. The system requires that only the information about the data usage of relevant code parts be provided. Moreover, a deterministic execution model is supported, based on sequential execution semantics.

More specifically, in this thesis, we present Roko, a system which allows a parallelization of sequential C programs with a modest user intervention. In Roko, the user writes sequential code (or uses an existing one) and then exposes parallelism at the function level by annotating the code with *pragmas*. Roko defines only two pragmas, the *parallel* pragma and the *exposed* pragma. The *parallel* pragma is used to describe the parallel units of work (*tasks*), while the *exposed* pragma is used to describe data usage of tasks. These annotations are used during run-time in order to detect dependences and enforce synchronization. Roko accomplishes this by using a custom hardware support, which implements a novel synchronization mechanism that we refer to as *versioning*.

To explore the feasibility of the proposed approach, we have assembled a symmetric multiprocessor (SMP) system for an FPGA, based on a SPARC V8 compliant processor [65], and modified it in order to support versioning. On top of this hardware, we have built a software platform (run-time system and compiler) which allows execution of annotated C code. Our experimental evaluation of the systems shows: (i) that few simple pragmas are all that is needed to expose parallelism in benchmark applications and (ii) that Roko can deliver good performance in terms of application speedup. Therefore, we believe that Roko represents a *sweet spot*, providing good results with little user effort.
1.1 Thesis Contributions

In this thesis we make the following contributions:

1. A novel versioning-based synchronization scheme. We present two instances of versioning: the fixed-granularity versioning and the variable-granularity versioning.

2. The hardware design of the variable-granularity versioning. We show that the proposed synchronization scheme can be implemented in hardware and that it requires only modest modification of the processor core by building an FPGA prototype and validating its functionality.

3. The implementation of Roko, a system that harnesses hardware support for versioning and allows a parallelization of sequential C programs with a modest user intervention. We show that Roko obtains good application speedup, while being easy to use.

1.2 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of Roko, followed by the description of the Roko pragmas and their usage in Chapter 3. Chapter 4 presents versioning, the scheme used by Roko in order to preserve sequential correctness during parallel execution. The hardware support that implements versioning is described in Chapter 5. Chapter 6 describes software components of Roko: the Roko kernel and the Roko compiler. The experimental evaluation of Roko is given in Chapter 7. Chapter 8 surveys related work. Finally, Chapter 9 presents concluding remarks and directions for future work.
Chapter 2

An Overview of Roko

In this chapter, we give an overview of Roko. We start by introducing the \textit{asynchronous} model of execution that Roko employs and the challenges associated with its use. Next, Roko is described from the programmer’s perspective, and the modifications that have to be made to the sequential code in order to execute it under the Roko framework are explained. Then follows an illustration of the parallel execution of an example application. The chapter concludes with a brief description of the architecture of Roko.

2.1 Asynchronous Execution

In the traditional sequential execution model a function call is executed in a \textit{synchronous} manner, i.e., the execution of the caller is paused until the callee finishes. In a multiprocessor environment, it is desirable to employ an \textit{asynchronous} model of execution. In such a model, the caller and callee proceed simultaneously on different processors, shortening execution time. However, care must be taken so that this overlapping does not break the semantics of the original program. For example, when a callee produces a value which is used by the caller, there exists a \textit{data dependence} between the caller and callee. In the asynchronous execution, when a caller reaches the execution point where the produced value is used, it must be ensured that the value has indeed been
produced by the callee. Otherwise, the caller must be delayed until the callee produces the expected value.

Roko uses the asynchronous execution model to speed up the execution of sequential programs. In order to identify the dependences and preserve them during the execution, Roko relies on *pragmas* inserted by the programmer. These pragmas are described next.

### 2.2 Programming Paradigm

There are two steps involved in preparing a sequential code for execution on Roko. First, the user exposes parallelism by identifying function calls that are meant to be executed asynchronously. This is done by inserting the `parallel` pragma before the corresponding function call. Second, the user, symbolically and using exposed pragmas, describes the *exposed accesses* of each function for which there is at least one call site marked by the `parallel` pragma. Informally, the exposed accesses of a function are accesses made by the function to variables that can also be accessed “outside” the function. For instance, an access to a global variable is an exposed access, while an access to a local variable is not. The formal definition of exposed accesses and how they are expressed by the `exposed` pragma is given in detail in Chapter 3.

With the above two modifications to the code, Roko extracts the dependences between asynchronous function calls, and ensures that they are enforced during execution. The usage of Roko’s pragmas is best illustrated by an example, which we present next.

#### 2.2.1 Merge Sort Example

Figure 2.1 shows an implementation of a merge sort algorithm. The function `merge` takes two sorted arrays `in1` and `in2` and merges them into the array `out`. The function `sort` takes as an input the array to be sorted (d), a temporary array (t), and the sizes of these arrays (n). If the size of the array is below some predefined value (`cutoff`) the
array is sorted using insertion sort (lines 22-23). Otherwise, \texttt{sort} divides the input array into four sections and recursively calls itself on these sections (lines 25-28). Once the sections are sorted, the function \texttt{merge} is invoked three times. The invocation of \texttt{merge} at line 30 merges the first two quarters into the first half of a temporary array \texttt{t}, while

![Figure 2.1: Merge sort.](image)
merge at line 31 merges the last two quarters into the last half of t. Finally, merge at line 33 merges two sorted half and produces the final result.

We would like to execute the function calls at lines 25-31 asynchronously. This is accomplished by inserting parallel pragma before each of the corresponding function invocation, as shown in Figure 2.2. Next, the exposed accesses of functions sort and merge must be described. The function sort accesses arrays t and d. These arrays can be accessed outside the function and thus, accesses to them are exposed. Consequently, exposed pragmas are inserted before the declaration of sort (lines 24-25). The expression d[0:n-1] denotes that the access is made to elements of the array d with indices 0 to n-1, i.e. the complete array. An analogous expression is used for array t. Similarly, exposed pragmas are inserted for the function merge (lines 1-3). Since arrays in1 and in2 are only read, we add (read) to the pragma. While denoting that a variable is only read in a function is optional, it may allow Roko to increase the degree of overlapping the execution of asynchronous function calls.

With these simple steps, the user’s work is done. Roko takes the annotated code and executes it while preserving correctness. Note that the annotated code can still be compiled with standard C compilers and executed sequentially; the pragmas are simply ignored. This avoids having a separate sequential and parallel source code versions of the same program.
# pragma exposed in1[0 : n1-1] (read)
# pragma exposed in2[0 : n2-1] (read)
# pragma exposed out[0 : n1+n2-1]
void merge(int *in1, int *in2, int *out, int n1, int n2) {
    while (n1 > 0 && n2 > 0) {
        if (*in1 <= *in2) {*out++ = *in1++; n1--;}
        else {*out++ = *in2++; n2--;}
    }
    while (n1 > 0) {*out++ = *in1++; n1--;}
    while (n2 > 0) {*out++ = *in2++; n2--;}
}

void insertionsort(int *d, int n) {
    int p, q, *k;
    for (p = 1; p < n; p++) {
        k = d[p];
        for (q = p - 1; 0 <= q && k < d[q]; q--) {
            d[q+1] = d[q];
        }
        d[q + 1] = k;
    }
}

# pragma exposed d[0 : n-1]
# pragma exposed t[0 : n-1]
void sort(int *d, int *t, int n) {
    if (n < CUTOFF) {
        insertionsort(d, d + n);
    } else {
        #pragma parallel
        sort(d, t, n/4);
        #pragma parallel
        sort(d + n/4, t + n/4, n/4);
        #pragma parallel
        sort(d + 2*(n/4), t + 2*(n/4), n/4);
        #pragma parallel
        sort(d + 3*(n/4), t + 3*(n/4), n-3*(n/4));
        #pragma parallel
        merge(d, d + n/4, t, n/4);
        #pragma parallel
        merge(d + 2*(n/4), d + 3*(n/4), t + 2*(n/4), n, n - 3*(n/4));
        merge(t, t + 2*(n/4), d, 2*(n/4), n - 2*(n/4));
    }
}

int main() {
    ...
    sort(data, temp, n);
    ...
}
2.3 Execution Model

For each function invocation that is annotated with a \texttt{parallel} pragma, a \textit{task} that will execute that function call is created. A task is a run-time entity created during a parallel execution, similar to threads in operating systems. However, due to one-to-one correspondence between a task and the function call executed by a task, we will often use the term task to refer to the corresponding function call, even in the context of the sequential execution. When the differentiation is necessary, the terms \textit{task-function call} and \textit{task-function} are used to refer to the corresponding function call and function, respectively.

At the beginning of program execution, only one task, the \textit{main task}, exists. The main task starts executing the main function of the program. Other tasks are created dynamically; a task is created when the execution reaches call sites marked with the \texttt{parallel} pragma. Points in the program at which a new task is created are called \textit{task creation points}. A task completes when its corresponding task-function call completes. The program completes when all tasks complete.

The decomposition into tasks and asynchronous execution can be visualized using the \textit{task-phase graph}. The 2-level task-phase graph of the merge sort example for the array size of 64 and cutoff parameter of 8 is shown in Figure 2.3. In this graph, the rectangles represent tasks, while circles represent \textit{task phases}. A task phase is a maximal sequence of instructions that ends with the creation of a new task or the completion of the current one. A horizontal arrow represents a transition between two phases of a task. Finally, a non-horizontal line denotes the creation of a task.

The execution of the example starts with the task that corresponds to the invocation of the \texttt{main} routine. The first phase of this task, denoted by A in Figure 2.3, ends by reaching the first task creation point (which corresponds to the line 31 in Figure 2.2). At that point, a new task that will execute the function call \texttt{sort(0, 15)} is created.
Both tasks continue their execution simultaneously\(^1\). Consequently, the main task will continue to create new tasks, and these in turn create new tasks. For the size of the input array of 64 and cutoff value set to 8, the entire program execution is decomposed into 31 tasks.

![Task-Phase Graph](image)

Figure 2.3: The task-phase graph for the merge sort example.

It is important to note that some of these tasks are not independent. Consider for example the task which executes \texttt{merge(0, 31)}. This task uses the result of tasks \texttt{sort(0, 15)} and \texttt{sort(16, 31)}. Consequently, during the execution it must be ensured that this dependence is preserved. Roko accomplishes this as follows. When a task \texttt{merge(0, 31)} is being created, Roko examines the corresponding \texttt{exposed} pragmas and determines that actual accesses will be made to arrays \texttt{data} and \texttt{temp}, and that indices are in the interval from 0 to 31. During the task's execution, Roko monitors all memory accesses made by the task. When the task tries to access arrays \texttt{data} and \texttt{temp}, Roko checks if these arrays have previously been accessed by tasks executing \texttt{sort(0, 15)} and \texttt{sort(16, 31)}, as they would be in the sequential execution. If they have not, \texttt{merge(0, 31)} task must wait. Once \texttt{sort(0, 15)} and \texttt{sort(16, 31)} complete, \texttt{merge(0, 31)} will be able to continue. In order to achieve the proper ordering of accesses Roko uses a novel

\(^1\)Assuming that there are at least two processing units in the multiprocessor system on which the application is being executed.
synchronization scheme, *versioning*, which is explained in Chapter 4.

### 2.4 System Architecture

Roko consists of three components: the *Roko compiler*, the *Roko kernel*, and *hardware support for versioning*, as depicted in Figure 2.4.

The Roko compiler is a source-to-source compiler that takes a C program annotated with Roko’s pragmas and produces C code which will be executed on the Roko kernel. Each function marked by the *parallel* pragma is transformed to a function of the same functionality but with the standardized interface so it can be managed by the Roko kernel.

The Roko kernel is a software environment which manages the execution of tasks on a shared memory multiprocessor system. The kernel controls the creation, scheduling, execution and termination of tasks and cooperates with the hardware support to preserve dependences during parallel execution.
The hardware support implements versioning, which ensures that sequential execution semantics is preserved. Memory accesses which could break inter-task dependences are intercepted and reported to the Roko kernel.
Chapter 3

The Roko Pragmas

Roko defines two pragmas, the parallel pragma and the exposed pragma. The parallel pragmas are used to mark function calls for asynchronous execution. The exposed pragmas are used to signal to Roko the data usage of functions whose function calls are targeted for asynchronous execution. Only certain accesses, exposed accesses need to be reported.

The chapter starts by defining the exposed accesses of a function, followed by the description of the syntax and semantics of the two pragmas. The chapter is concluded by showing that the information about the exposed accesses suffices to preserve dependences during a parallel execution.

3.1 Exposed Accesses of a Function

For functions whose function calls are marked with the parallel pragma, Roko requires users to indicate the objects that they access. An object is a conceptual, programming-level entity that during program execution occupies a piece of a computer memory\(^1\). For instance, in the C programming language, an object can be either a variable or it can

\(^1\)Not to be confused with the concept of an object in the object-oriented paradigm.
correspond to a dynamically allocated memory referenced through a pointer.

Not all accessed objects need to be specified; rather, the user must list only those to which an exposed access is made. An exposed access is a memory access made inside the function being considered to an object created before the function is called. Furthermore, a deallocation of a memory used to hold some object created before the function is called is also considered to be an exposed access.

In the context of the C programming language, the above mentioned term “object created before the function is called” denotes either an object with static storage duration (i.e., global variables and static local variables), or an object with automatic or dynamic storage created before the function is invoked (i.e., local variables of the caller functions or the memory dynamically allocated before the function call). While it is rather straightforward for the user to recognize global variables and static local variables (and accesses made to them), identifying pre-allocated objects with non-static storage duration during compile time could be more challenging. However, the user can identify the exposed access by a process of elimination. Consider all the objects accessed inside the function. Then, just by analyzing the function, local variables, parameters and dynamic memory allocated inside the function can be identified. The accesses to remaining objects should be specified as exposed.

Two rules need to be respected when specifying objects accessed by a function in an exposed manner. First, the user must take into consideration accesses made by all functions called within the inspected function. For example, if function \( f_1 \) calls function \( f_2 \) and \( f_2 \) accesses some global variable \( x \), then, when reporting objects accessed by \( f_1 \), \( x \) should be included. This requirement was previously identified by Chan and Abdelrahman and named inclusion property, a term that we use as well.

Second, when analyzing the exposed accesses of a function, all possible accesses have to be conservatively examined, even if some of them may not take place during every invocation of the function. For example, depending on the function parameters, some
function’s control paths may not be exercised, and consequently, the accesses belonging
to those control paths will not take place. Nevertheless, objects accessed on all control
paths need to be included.

3.2 The parallel pragma

The parallel pragma signals to Roko that the associated function call should be exe-
cuted asynchronously. The line following this pragma is called a target line and it must
be a function call in one of the two following forms:

- \texttt{foo(args)};
- \texttt{var = foo(args)};

Here, \texttt{foo} is the function targeted for asynchronously execution, \texttt{args} is the argument
list, and \texttt{var} is a variable used as a storage for the return value. If the argument list
contains a function call, that function call will not be executed asynchronously, i.e., only
the invocation of the function \texttt{foo} is executed asynchronously. The rationale of this
restriction, and why other, more complex C expressions are not allowed in a target line is
explained as follows. Assume that a function call that is executed asynchronously is an
element of more complex C expression, e.g., an element of the argument list of another
function call or an arithmetic expression. In that case, the return value of that function
call will be used immediately (or just a few instructions) after the asynchronous call is
executed. This prohibits any overlapping during execution, and no performance benefits
can be gained. Indeed, some performance degradation may occur due to the overhead
of asynchronous execution. Having this in mind, we decided not to allow function calls
which are elements of complex C expressions to be marked for asynchronous execution.

The case when a function has a return value differs from the above mentioned case
when no parallelism exists. Here, albeit the return value of the function is immediately
saved to some variable, it is not actually immediately \textit{used}. It will be used when the
variable holding the return value is subsequently read.
3.3 The exposed pragma

The exposed pragma is used to describe the exposed accesses of a function. The function to which this pragma refers to is called its target function. The line following the exposed pragma is either a start of the target function declaration, target function definition or another exposed pragma (in the case multiple pragmas are used to describe the exposed accesses of the target function).

The syntax of the directive is:

```
#pragma exposed expression (read)
```

where expression is a C-like expression identifying an object to which an exposed access will be made. The optional argument read is used to denote that the accesses are reads. The grammar rules for constructing expression are shown in Figure 3.1.

The displayed grammar is based on the symbolic access paths notation [21], but adapted to the C programming language. Note that, without the production rules P1-P5, the grammar generates a subset of C expressions; this subset is used to specify simple accesses. Production rules P1-P4 enable describing accesses to recursive structures, while the production rule P5 makes possible to specify accesses to array regions. The semantics that must be respected while describing each type of accesses mentioned above are explained in the next three subsections.

It should be noted that the above grammar, and the imposed semantic rules, limits the expressibility of the exposed pragma. Consequently, for some C functions it is not possible to describe their exposed accesses. For example, the expression *(x & y) represents a valid C statement which will result in a memory access to an object stored at the address x & y. However, this access cannot be expressed with the presented grammar. Similarly, no function calls or static local variables are currently allowed in the exposed pragma. In such cases, in the current implementation of Roko, a function cannot be executed asynchronously. These constraints can be easily relaxed in future implementations of Roko.
3.3.1 Describing Simple Accesses

A simple exposed access of a function is specified in an intuitive manner using a C expression consisting of the names of variables, integer constants, arithmetic operators
Chapter 3. The Roko Pragmas

(\(+\), \(-\), \(*\), \(/\)), brackets (\(\{\), \(\}\)), member access operators (\(\rightarrow\), \(.\)), and pointer dereference operator (\(\ast\)).

An exposed pragma expression will expose accesses to the object identified by the expression. If the expression identifies a variable of a basic type, accesses to that variable are exposed. If the expression is of a pointer type, accesses to that pointer will be exposed (and not the accesses to the object that a pointer is pointing to)\(^2\). In the case of an expression of an array type, accesses to the entire array are exposed. The same is true in the case of structures and unions; an expression of a structure or union type captures accesses to all members of the identified structure or union, respectively. Figure 3.2 shows a couple of examples of describing simple accesses.

```
int x;
#pragma exposed x  // Access to variable x

int *p;
#pragma exposed p   // Access to pointer p
#pragma exposed *p  // Access to the object pointed by p
#pragma exposed *(p+3)  // Access to the object stored
                       // at the address (p+3)

int a[10];
int l;
#pragma exposed a    // Access to the entire array a
#pragma exposed a[1+3] // Access to a single element

struct node{
  int data;
  struct node *next;
};
struct node *head;
#pragma exposed *head // Access to the entire structure
#pragma exposed head->next // Access to field next
#pragma exposed *(head->next) // Access to the entire structure
                // pointed by head->next
```

Figure 3.2: Example usage of the exposed pragma. The comments describe the accesses captured with the corresponding pragmas. Declarations of variables are shown preceding their usage in pragmas only for the purpose of presentation.

An expression used to describe a simple access is legal, if and only if it refers to an

---

\(^2\)To expose accesses to the pointed object, the pointer should be dereferenced in the expression.
object and, when inserted at the beginning of the target function definition body and ended with a semicolon, conforms to the semantics of the C programming language. An expression that is not legal (e.g., an expression consisting only of a constant) will cause a compilation error. One direct implication of the above stated definition of legality is that identifiers used in the expression must be either names of global variables, formal parameters of the target function or members of structures (i.e., local variables cannot be used).

The definition of legality stems from the fact that the expression of the exposed pragma is evaluated before the body of the target function is executed; consequently, local variables would be out of their scope.

### 3.3.2 Describing Recursive Accesses

Recursive pointer accesses can be found in many programs. For example, the code given in Figure 3.3 traverses a linked list and updates the values of data fields of all nodes. During the execution, accesses will be made to head->data, head->next->data, head->next->next->data, etc. To describe exposed accesses for such types of code, where it is not possible to enumerate all possible accesses, the asterisk (*) and plus (+) repeat operators (productions P1-P2 in Figure 3.1) are used.

```c
struct node{
  int data;
  struct node *next;
};

#pragma exposed head->next->next->next->data

void traverse_list(struct node *head){
  while (head != NULL){
    head->data++;
    head = head->next;
  }
}
```

Figure 3.3: An example illustrating recursive pointer accesses and the usage of repeat operators.
Chapter 3. The Roko Pragmas

The repeat operators, which are similar to ones in regular expressions, indicate that the preceding element in the exposed pragma expression should be repeated zero or more (*) times or one or more (+) times. The legality of an expression used to describe recursive accesses is defined with the following constraints. First, at most one repeat operator can be used in an expression. Second, if the exposed pragma expression is represented with a regular expression \( r_1 \rightarrow \text{identifier}(+|*) \rightarrow r_2 \), then \( r_1 \rightarrow \text{identifier} \rightarrow r_2 \) must be a legal expression describing a simple access. Finally, \( r_1 \) and \( r_1 \rightarrow \text{identifier} \) must be of the same pointer type.

Additionally, repeat operators can be used to describe access to tree structures, by using alternatives. Figure 3.4 shows an example code which traverses a binary tree.

```c
struct node{
  int data;
  struct node *left;
  struct node *right;
};

#pragma exposed root->{left, right}*->data
void traverse_tree(struct node *root){
  if (root != NULL){
    root->data++;  // (i)
    traverse_tree(root->left);
    traverse_tree(root->right);
  }
}
```

Figure 3.4: An example illustrating the usage of repeat operators and alternatives for a tree traversal.

During execution, accesses will be made to root->left->data, root->right->data, root->left->left->data, etc. Productions P3-P4 of the grammar shown in Figure 3.1 enable describing this type of accesses. For this example, the expression root->{left, right}+->data captures all the exposed accesses. The identifiers inside the curly brackets represent alternatives. The legality of an expression including a repeat operator and alternatives is defined similarly as for an expression including only a repeat operator: (i) at most one repeat operator can be used in an expression, (ii) if we represent the expression with a regular expression \( r_1 \rightarrow \{\text{identifier}_1, ..., \text{identifier}_n\}(+,*) \rightarrow r_2 \), then
$r_1 \rightarrow \text{identifier}_1 \rightarrow r_2, \ldots, r_1 \rightarrow \text{identifier}_n \rightarrow r_2$ must be legal expressions describing a simple access, and (iii) $r_1, r_1 \rightarrow \text{identifier}_1, \ldots, r_1 \rightarrow \text{identifier}_n$ must be of the same pointer type.

### 3.3.3 Describing Accesses to Array Regions

Arrays are often accessed in loops, whose trip counts may not be known at compile-time. Consider the matrix multiplication example given in Figure 3.5. The number of iterations of each loop depends on the parameter $n$, which is the size of arrays. Moreover, even if the array size were fixed, it would be exhausting to write the exposed pragma for every accessed array element. For these reasons, the production $P_5$ (Figure 3.1) is introduced, which enables describing accesses to a region of an array with a single exposed pragma.

```c
#pragma exposed a [0:n*n-1] (read)
#pragma exposed b [0:n*n-1] (read)
#pragma exposed c [0:n*n-1]
void matrix_mul(int *a, int *b, int *c, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++){
        for (j = 0; j < n; j++){
            for (k = 0; k < n; k++){
                c[i*n + j] += a[i*n + k] * b[k*n + j];
            }
        }
    }
}
```

**Figure 3.5: Matrix multiplication**

To expose accesses to an array region, the user writes an expression of the form $\text{array}[\text{lower} : \text{upper}]$, where $\text{array}$ represents the array being accessed, and $\text{lower}$ and $\text{upper}$ denote the lower and the upper bound of the accessed region. For the matrix multiply example, the expression $a[0:n*n-1]$ describes accesses to all elements of the array $a$.

The legality of an expression used to describe accesses to an array region is defined as follows. If we represent the exposed pragma expression with a regular expression
array[lower:upper], then: (i) lower and upper must, when inserted at the beginning of the target function definition body and ended with a semicolon, conform to the semantics of the C programming language and (ii) array[lower] and array[upper] must be legal expressions describing simple or recursive accesses.

3.3.4 Indirect Accesses

Some C expressions involve indirect memory accesses. For example, the expression `head->next->data = 1` besides writing 1 to `head->next->data` also implies read accesses to `head` and `head->next`. For the exposed pragma expressions, Roko will implicitly add indirect memory accesses to the exposed accesses of a function. This is a conservative approach, as indirect accesses may not be necessarily exposed accesses. However, we believe the analysis of indirect accesses is a daunting task for the user.

Indirect accesses are included by Roko as read accesses. Therefore, if inside the function a write access is made to a region to which an indirect access exists, it is necessary to expose the write access using the exposed pragma directive. Consider the example given in Figure 3.6. The expression at line 6 gives rise to an indirect read access to `p` and a write access to `*p`. These accesses are described by the exposed pragma at line 3. The write to `p` at line 7, however, is not captured by this pragma, and it is necessary to include an additional pragma (line 4).

```c
int *p;

#pragma exposed *p
#pragma exposed p

void foo () {
    *p = ...;
    p = ...;
}
```

Figure 3.6: An example of indirect accesses.
3.4 Sufficiency of Exposed Accesses

In general, a parallel execution of a program preserves the sequential execution semantics (i.e., it is sequentially correct) if the results are identical to those of the sequential execution. A parallel execution is sequentially correct only if dependences are preserved.

In our case, this corresponds to satisfying the dependences between accesses made by different tasks, since accesses in a single task are always executed sequentially and therefore dependences between them are respected. Accesses made by different tasks (i.e., their task-function calls) during the sequential execution imply precedences between tasks that must be satisfied during a parallel run in order to maintain the sequential execution semantics.

Definition 3.4.1. Let \( t \) denotes some task, \( p \) a program point inside that task, and \( o \) some object. A task \( t' \) is an (exposed) preceding task of \( t \) on \( o \) at \( p \) if and only if in the sequential execution \( t' \) makes an (exposed) access to \( o \) and finishes before \( p \).

Defining precedence relative to some program point is due to the fact that tasks create other tasks during the execution, and the newly created tasks can become their (exposed) preceding tasks. For example, in the merge sort from Chapter 2 there are no tasks that precede \texttt{main} task on any object when the program starts. But after the task \texttt{sort(0,15)} is created, it precedes \texttt{main} on, for instance, object \texttt{data[0]}.

In order to preserve dependences between tasks during a parallel run, a task \( t \) can write (read) some object \( o \) at some program point \( p \) only if all the preceding tasks of \( t \) on \( o \) at \( p \) have finished accessing (writing) \( o \). The claim that we make is that dependences can be preserved relying only on the information about the exposed accesses. More specifically, if during a parallel run, a scheme that guarantees that a task can write (read) an object \( o \) at program point \( p \) only if all the exposed preceding tasks (writers) on \( o \) at \( p \) have finished, is employed, then dependences will be preserved.

Assume that, during a parallel run of some program, the above mentioned scheme
is used, and that a task $t$ writes (reads) some object $o$ at program point $p$ while there exists a preceding task (writer) $t'$ of $t$ on $o$ at $p$ that has not finished. By the assumption, $t'$ is not an exposed preceding task (writer) of $t$ on $o$ at $p$, since otherwise $t$ could have not written (read) $o$ at $p$. This implies, by the definition of exposed accesses, that $o$ is created inside $t'$. Since $t$ accesses $o$, the address of $o$ must somehow be passed from $t'$ to $t$.

In the simplest case, there exists an object $o'$ that is (i) created before $t'$, (ii) $t'$ writes the address of $o$ to $o'$, and (iii) $t$ reads $o'$ before $o$, at some program point $p'$. Hence, $t'$ is an exposed preceding writer of $t$ on $o'$ at $p'$. Consequently, $t'$ must have finished before $p$ was reached, which leads to contradiction. In a more complicated case, there exists a sequence of objects $o_1, ..., o_n$ such that the address of $o$ is written to some object $o_1$, the address of $o_1$ is then written to some object $o_2$, ..., the address of $o_{n-1}$ is written to some object $o_n$, and then finally $o_n$ is read by task $t$ before $p$. However, by using mathematical induction on $n$, it is easy to see that the claim holds in this case as well.

So far, the fact that functions can return values has been ignored, which introduces additional dependences. However, a function that returns a value can be easily transformed into a function that does not, as done by Roko compiler. These transformations are described in Section 6.2.2.
Chapter 4

Versioning

Roko uses a scheme we refer to as versioning to ensure the proper ordering of memory accesses during parallel execution. In this chapter, we first give an overview of versioning. Next, two instances of versioning are presented. The chapter concludes with the guarantees that versioning provides.

4.1 An Overview of Versioning

In versioning, every memory location accessed by a running program is assigned a version number. Each running task maintains a pair of version numbers \((a, r)\) for each memory location it accesses: an acquire version number and a release version number. At the start of execution, each memory location is assigned a version number of 0. The task that executes the main function of the program has \((0, W)\) as its pair of version numbers for each location, where \(W\) is a large integer. A task is allowed to access a memory location if and only if its acquire version number matches that of the memory location; otherwise, the task waits. When a task finishes execution, it changes the version numbers of all its memory locations to its release version number.

The acquire and release version numbers of tasks are assigned dynamically as the tasks are created at run time. They are assigned in such a way that tasks that access a
certain memory location “earlier” in sequential execution receive smaller acquire numbers for that memory location than tasks that access the location “later”. Furthermore, the release version number of a task for a memory location is always larger than its own acquire version number for the same location and it matches the acquire version number of the task that access the location immediately later in sequential execution. This ensures that tasks access each memory location in sequential execution order and that deadlock can not occur.

The versioning scheme is illustrated using the example shown in Figure 4.1a. The figure shows a simple program that consists of a main function that writes to the global variable $x$, then invokes the function childA, and eventually returns the value of $x$. The function childA increases the value of $x$, invokes the function childB, increases $x$ again, and then returns. Similarly, the function childB increases $x$ and returns. Since $x$ is read and written by these three functions, any parallel execution must preserve the sequential execution order with respect to the variable $x$.

The parallel execution of the above example is depicted in Figure 4.1b. When the task that executes main starts, it is assigned the acquire and release numbers of $(0, W)$ for the memory location used to store $x$. When main attempts to first access $x$, its acquire version number is compared to the version number of $x$. Since the numbers match, main proceeds to access $x$. When childA is invoked, a new task is created to execute it. The new task receives the version numbers of $(0, \lfloor \frac{W}{2} \rfloor)$ while the version numbers of main are updated to $(\lfloor \frac{W}{2} \rfloor, W]$. This signals that childA must access $x$ before main does. Thus, when main attempts to access $x$ again while childA is still executing, as shown in the figure, it must wait since its acquire version number no longer matches that of $x$.

In contrast, childA is able to access $x$ and then creates a task that executes childB. The acquire and release numbers of both childA and childB are updated as shown in Figure 4.1b. Thus, when childA attempts its second update of $x$ while childB is still

---

1The floor and ceiling functions are used to ensure that all numbers used in versioning are integers.
Chapter 4. Versioning

(a) Program code.

(b) The time-line of the parallel execution.

Figure 4.1: A simple program used to illustrate versioning.
running, it must wait because of the mismatch between its acquire version number and that of \( x \).

However, when \texttt{childB} completes execution, it updates the version number of \( x \) to its release version number, or \( \lfloor \frac{W_4}{2} \rfloor \). This matches the acquire version number of \texttt{childA} at this point, allowing it to continue and to increase \( x \). When \texttt{childA} in turn completes execution, it updates the version number of \( x \) to its release version number, \( \lfloor \frac{W_2}{2} \rfloor \), which allows \texttt{main} to continue. Thus, the three concurrently executing tasks, \texttt{main}, \texttt{childA} and \texttt{childB} access the variable \( x \) according to sequential execution order.

The above versioning scheme maintains sequential execution semantics during a parallel execution. However, its implementation in a system has several challenges that must be addressed for it to become practical. The remainder of this section describes these challenges and how Roko addresses them.

First, versioning requires that all memory locations that will be accessed by a task are known at compile time. This is necessary to be able to compare acquire version numbers with those of the memory locations. Obtaining this information with static analysis is undecidable \cite{47}. To mitigate this issue, Roko requires users to indicate the exposed accesses of each function, using pragmas, as explained in Chapter 3. The information about these accesses, sufficient to preserve dependences, is stored in the exposed table (ET) that is maintained for each task and created before a task is forked.

Versioning also requires that version numbers of all memory locations are maintained. Roko stores all version numbers in the system in shared memory in a global version table (GVT). Obviously, if version numbers of all memory locations were maintained, the size of this table would be proportional to the total size of the memory. To reduce this space requirement, Roko creates version numbers dynamically during the program execution and only for memory locations to which exposed accesses are made. For example, in the parallel execution of program from Figure 4.1a, neither the version number nor the acquire and release numbers for \( x \) are not created until \texttt{childA} is forked. Still, the
size of the GVT can be considerable depending on the size of the data accessed by a program. Therefore, rather than assigning version numbers to memory locations, Roko assigns version numbers to memory regions, i.e., to a set of consecutive memory locations. For instance, version numbers may be assigned at the granularity of a page. Clearly, a coarser granularity may limit parallelism due to false sharing [12]. In order to tackle this problem Roko uses variable granularity regions, whose size is adjusted during the program execution. Version numbers are first assigned at a coarser granularity and are made finer during execution to prevent false sharing. However, this comes at the expense of a more complex versioning scheme.

Furthermore, the acquire and release version numbers of each running task and for each memory region accessed by a task in an exposed manner need to be maintained. In Roko, this information is stored in a local version table (LVT) that is maintained by each task. This table must be accessed upon every memory access in order to compare the acquire version number of the task for the accessed region with the version number of that region in the GVT. This implies that not only must the LVT of a task be efficiently searched to lookup the acquire version number, but also the corresponding version number in the GVT be located. Performing these operations in software would be detrimental to performance. Therefore, Roko uses hardware structures to both store the LVT and to speedup the lookup process. In addition to storing task version numbers, the LVT stores an index to the corresponding version number in the GVT. Special purpose hardware is used to lookup the version number in the GVT and to perform version number comparisons. The special purpose hardware is also used to generate the acquire and release numbers of parent and child tasks at points of task creation.

Despite the use of hardware support, the execution of an additional memory access (to look up the GVT) on every memory access would also be detrimental to performance. Therefore, Roko uses a filtering mechanism that is based on a Bloom filter [10] to ensure that only a small fraction of memory accesses requires a lookup into the LVT and GVT.
Since a lookup into the Bloom filter is performed during cache access, most memory accesses are executed without any delay.

Finally, versioning must support multiple concurrent readers to the same memory region. Thus, Roko modifies the scheme described above by dividing each version number into two components: a write version number and a read version number. A task is allowed to read from a memory region if and only if its acquire version number matches the write version number of the memory region. Hence, assigning the same acquire version number to multiple readers allows concurrency of their accesses. A read version number, in conjunction with an additional number maintained by tasks, called delta number, is used to enforce the proper ordering of accesses between concurrent readers and the subsequent writer. In addition to requiring a match between the acquire number and the write version number, a task can perform a write only if the sum of its delta number and the read version number of accessed region is equal to $R$, where $R$ is a large integer.

\section{Fixed-granularity Versioning}

In fixed-granularity versioning, version numbers are assigned to memory regions of a fixed size, and that size is maintained for all memory regions during the entire program execution. Although Roko does not use this scheme, it is a good introduction to the more complex variable-granularity scheme. We first present the data structures used in fixed-granularity versioning, followed by the description of the algorithms operating on these structures, explaining how they interact in order to preserve dependences between tasks. For the clarity of presentation, it is assumed that all objects have a static storage duration. Versioning, however, operates correctly in the presence of objects with non-static storage duration as described in Section 4.2.3.
4.2.1 Structures

The fixed-granularity scheme uses three types of structures, illustrated in Figure 4.2. The global version table (GVT) is a structure shared by all tasks and contains all the version numbers of memory regions. There is only one GVT in the system. Each entry of this table corresponds to one version number, and \( GVT[i] \) is used to denote the \( i \)th entry of the GVT. As mentioned before, in order to support multiple consecutive readers, a version number is a pair \((wv, rv)\) that consists of a write version number and a read version number\(^2\) respectively. Version numbers are created dynamically during the program execution, and the global counter, called the version counter \((vc)\), maintains the current count of versioning numbers in the system. Initially, the GVT is empty and the version counter is set to 0. When a new version number is added to the GVT, both of its components are set to 0, and the version counter is increased.

![Figure 4.2: The structures used in fixed-granularity versioning.](image)

A local version table (LVT) contains versioning information needed only by a single task. There exists one LVT for each task in the system; we write \( LVT_t \) to denote the LVT of a task \( t \). The LVT of a task contains one entry for each memory region that

\(^2\)The terms the write component and the read component are also used.
is used as a storage for an object accessed in an exposed manner by the task or by the
children of the task. Entries are added to the LVT when a task is created and when the
task forks child tasks. An entry in LTV is a 6-tuple denoted as $l = (addr, read, i, a, r, \Delta)$.
The elements of an entry are described in Table 4.1. Since there can be at most one
entry for any memory region\(^3\) and regions are of a fixed size (REGION\_SIZE), an LVT
entry is uniquely identified with the $addr$ element. $LVT[addr]$ is used to denote the
entry corresponding to a region that starts with the address $addr$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$addr$</td>
<td>start address</td>
<td>start address of the memory region</td>
</tr>
<tr>
<td>$read$</td>
<td>read</td>
<td>1 if the region is only read by the task, otherwise 0</td>
</tr>
<tr>
<td>$i$</td>
<td>index</td>
<td>index to the GVT used to access the version number of the memory region</td>
</tr>
<tr>
<td>$a$</td>
<td>acquire version number</td>
<td>the value that the write version number pointed by $i$ must be equal to in order for the task to access the memory region</td>
</tr>
<tr>
<td>$r$</td>
<td>release version number</td>
<td>used to update the write version number pointed by $i$</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>delta number</td>
<td>if $read == 0$, used to calculate the value to which the read version number pointed by $i$ must be equal in order for the task to write to the memory region if $read == 1$, used to update the read version number pointed by $i$</td>
</tr>
</tbody>
</table>

Table 4.1: Elements of an LVT entry in fixed-granularity versioning.

The LVT of a task is initialized when the task is created using the exposed table
(ET) of a task. The exposed table of a task contains an entry for each memory region
that is used as a storage for an object that will be accessed in an exposed manner by
the task. An ET entry is a pair $e = (addr, read)$ where the elements $addr$ and $read$

\(^3\)Due to this fact, the terms LVT entry and region are used interchangeably in the context of an LVT. For example, we state the LVT contains a region meaning the LVT contains an entry that corresponds to the memory region.
have the same meaning as the corresponding elements of an LVT entry. The exposed table of a task is created by mapping the objects that the task accesses in an exposed manner (as specified by the exposed pragmas) to memory regions that those object occupies, taking into account the granularity of regions. For example, if an exposed access is made to an object that occupies memory area \([0x40001204, 0x40001214]\) and double-word granularity is used, then three entries, corresponding to regions \([0x40001200, 0x40001207]\), \([0x40001208, 0x4000120f]\) and \([0x40001210, 0x40001217]\) will be added to the exposed table of a task.

### 4.2.2 Algorithms

The fixed-granularity versioning employs three algorithms: the acquire algorithm, the release algorithm and the generate algorithm. The combination of these algorithms ensures that dependences are preserved, i.e., a task \(t\) can write (read) some object \(o\) at some program point \(p\) only if all the preceding tasks of \(t\) on \(o\) at \(p\) have finished. Besides preceding tasks, defined in Section 3.4 in the description of algorithms, the notion of writer (reader) predecessor of a task is used as well.

Informally, the writer predecessor of a task \(t\) on object \(o\) is the task that in the sequential execution writes to \(o\) earlier than \(t\) accesses \(o\), and later than all the other preceding task of \(t\) write \(o\). Similarly, a reader predecessor of a task \(t\) on object \(o\) is the task that in the sequential execution reads \(o\) earlier than \(t\) writes to \(o\), and later than all other preceding tasks of \(t\) write \(o\). Notice that there could be at most one writer predecessor, but arbitrary many reader predecessors.

**Definition 4.2.1.** Let \(t\) denotes some task, \(p\) a program point inside that task, and \(o\) some object. A task \(t'\) is a writer (reader) predecessor of \(t\) on \(o\) at \(p\) if and only if \(t'\) is a preceding writer (reader) of \(t\) on \(o\) at \(p\) and there does not exist a preceding writer \(t''\) of \(t\) on \(o\) at \(p\) that in the sequential execution finishes after \(t'\) and before point \(p\).
Acquire Algorithm

The acquire algorithm is used to check if a task is allowed to execute a certain memory access. Its pseudocode is shown in Figure 4.3. If the LVT does not contain the region to which an access is trying to be made, the access can always be executed (line 1). The generate algorithm guarantees that in this case there does not exist a preceding task.

**Acquire(t, addr, read)**

1: if $\exists l \in LVT_i$ such that $l.addr \leq addr \leq l.addr + \text{REGION\_SIZE}$ then
2: if $read == 1$ then
3: while $l.a \neq GVT[l.i].wv$ do
4: wait
5: else
6: while $(l.a \neq GVT[l.i].wv) || (l.\Delta + GVT[l.i].rv \neq R)$ do
7: wait

Figure 4.3: The acquire algorithm: task $t$ is trying to access memory location $addr$. The $read$ parameter has a value of 1 in a case of a read access, and 0 otherwise.

In a case of a read access, the acquire version must match the write component of the version number associated with the memory region being accessed ($GVT[l.i].wv$) (line 3). The generate and release algorithms ensure that the write version number is equal to the acquire version number only after the writer predecessor finishes. Further, the release algorithm also ensures that the writer predecessor has finished only after all its preceding writers have finished. Therefore, a task is allowed to execute a read access only after all the preceding writers have finished.

For a write access, a stronger condition needs to be fulfilled. Here, in addition to requiring the match between the acquire number and the write version number, the sum of the delta number and the read version number must be equal to integer $R$ (line 6). The generate and release algorithms ensure that the read component of the version number associated with the memory region being accessed ($GVT[l.i].rv$) is increased only when the reader predecessors on objects stored in the memory region finish and that the total
amount of the increase is exactly $R - l.\Delta$. Consequently, a task can execute a write only after the writer predecessor and all read predecessors have finished. Similarly as in the case of read accesses, it follows that a task can execute a write only after all preceding tasks have finished.

**Release Algorithm**

Each task, right before it finishes, updates the version numbers of all the memory regions that exist in its LVT. This action is defined by the release algorithm, whose role is to signal tasks that all their preceding tasks (or writers, depending on the type of accesses to the region being considered) have finished. Its pseudocode is shown in Figure 4.4.

```
Release(t)
1: for all $l \in LVT_t$ do
2: Acquire(t, l.addr, l.read);
3: if $l.read == 1$ then
4:   $GVT[l.i].rv = GVT[l.i].rv + l.\Delta$;
5:   else
6:   $GVT[l.i].wv = l.r$;
7:   $GVT[l.i].rv = 0$;
```

Figure 4.4: The release algorithm.

The act of updating is performed differently for regions that were accessed in read-only manner than for regions that were also written to. In both cases, it is ensured that the writer predecessor (and all the reader predecessors if a region was written to) has finished before the version numbers can be modified, which implies that all the preceding writers (tasks) have finished as well\(^4\). This is enforced by invoking the acquire algorithm (line 2). If a task only reads a region then the read component is increased by $l.\Delta$ (line 4). The addition operation has to be done atomically, since the GVT is a shared structure.

\(^4\)Because an predecessor will also execute the release algorithm ensuring that *its* predecessor have finished, and so on.
On the other hand, if a region is written to, the write component is set to \( l.r \), and the read component is reset to zero (lines 6-7).

**Generate Algorithm**

The generate algorithm defines how acquire, release and delta numbers are created. The algorithm is run at task creation points: it initializes the LVT of the child task and updates the LVT of the parent task, using the exposed table of the child task as input. Its pseudocode is shown in Figure 4.5. The algorithm iterates over all regions in the exposed table of the child task and adds entries that correspond to those regions to the LVT of the child task. Additionally, new entries to the parent LVT are added and/or the existing ones are updated as follows.

---

**Generate**\( (t_p, t_c) \)

1. for all \( e \in ET_{t_c} \) do
2.   if \( \exists l' \in LVT_t \) such that \( l'.addr = e.addr \) then
3.     \( l = l' \); 
4.     else
5.     \( l = (e.addr, 0, vc, 0, W, R); \)
6.     \( GVT[vc].wv = 0; \)
7.     \( GVT[vc].rv = 0; \)
8.     \( vc = vc + 1; \)
9.   if \( e.read == 0 \) then
10. \( LVT_{t_c}[e.addr] = (e.addr, 0, l.i, l.a, \lfloor \frac{l.a + l.r}{2} \rfloor, l.\Delta); \)
11. \( LVT_{t_p}[e.addr] = (e.addr, l.read, l.i, \lfloor \frac{l.a + l.r}{2} \rfloor, l.r, R); \)
12. else
13. \( LVT_{t_c}[e.addr] = (e.addr, 1, l.i, l.a, N/A, \lfloor \frac{l.\Delta}{2} \rfloor); \)
14. \( LVT_{t_p}[e.addr] = (e.addr, l.read, l.i, l.a, l.r, \lceil \frac{l.\Delta}{2} \rceil); \)

---

**Figure 4.5**: The generate algorithm: task \( t_p \) is creating task \( t_c \). \( l \) is a temporary variable used to denote either the entry in parent’s LVT before modifications or a newly initialized entry.

The inclusion property of exposed accesses guarantees that a region from the exposed
table does not exist in the LVT of the parent task only if there are no preceding tasks on the object stored in the memory region being considered. Hence, if a region from the exposed table cannot be found in the LVT of the parent task, a new entry will be added to both LVTs, and a new version number that is used to synchronize accesses among the parent and child task will be added to the system (lines 5-8). Since the version counter is shared by all tasks, the addition at line 8 must be done atomically. For regions that already exist in the LVT of the parent task, or that have just been inserted, the algorithm proceeds as follows. For regions that are written to, the release version number of the child task and the acquire version number of the parent task are set to \(\lfloor \frac{l.a + l.r}{2} \rfloor\) (lines 10-11). Under the condition\(^5\) \(l.a > l.r + 1\), this ensures that the release version number of the child task and the acquire version number of the parent task are the same and different than all previously generated release version numbers for version number \(GVT[i]\). In effect, the parent task will be able to access this memory region only after the child task finishes, as it would happen in the sequential execution. By setting the acquire version number of the child task to the previous value of the acquire version number of the parent task, and not modifying the release version number of the parent task, the proper ordering of accesses in relation to other tasks is maintained.

It is easy to check that any number from interval \((l.a, l.r)\) can be chosen instead of \(\lfloor \frac{l.a + l.r}{2} \rfloor\). However, assuming that both the parent and the child task will subsequently create an equal number of tasks that will access the region being considered, and these in turn again create an equal number of tasks that access the same region, and so on, choosing \(\lfloor \frac{l.a + l.r}{2} \rfloor\) maximizes the number of tasks that can be created until the situation in which the condition \(l.a > l.r + 1\) no longer holds occurs. If it is, however, known in advance that the parent and child task will not create an equal number of tasks that will access the region being considered, choosing a different number from interval \((l.a, l.r)\)

\(^5\)The condition is fulfilled if a sufficiently large \(W\) is chosen. However, if the condition does not hold, a child task cannot be created, since sequential execution semantics can no longer be guaranteed.
could be beneficial.

For read-only regions, the algorithm ensures that the sum of the $\Delta$ elements of the parent and child task (that is now a reader predecessor of the parent) remains the same as the original $\Delta$ element of the parent task (lines 13-14). Consequently, for every task $t$ which writes to the object $o$ stored in region $m$ the total sum of the $\Delta$ elements of reader predecessors on $o$ is $R - l.\Delta$, where $l$ denotes the entry in $LVT_t$ corresponding to the region $m$. Hence, if the reader predecessors increase some variable by their delta number as they finish (as it is done by release algorithm where the read component is increased) the writer can easily check if all the reader predecessors have finished, by examining the value of that variable (as it is done by the acquire algorithm where the read component is examined). Next, by setting the acquire version number of the child task to the value of the acquire version number of the parent task, the proper ordering of accesses of the child task in relation to preceding tasks is maintained. Further, not modifying the acquire version number of the parent task results with the following facts. First, reads done by the parent task (those that precede eventual writes done by the task), can be done concurrently with the preceding readers. Second, subsequently forked readers will be allowed to execute accesses concurrently with the previously created preceding readers. Consequently, this scheme supports multiple concurrent readers.

One would expect that all tasks use the same version number to synchronize accesses to a certain object. In the case of a statically allocated memory, due to the one-to-one correspondence between regions and static objects, that is true. However, for dynamically or automatically allocated objects, it could happen that different tasks use different version numbers to synchronize accesses to the same object. Similarly, the case when different tasks use the same version number to synchronize accesses to different objects is also possible. In fact, versioning exploits this scenario to maintain the proper ordering of accesses to stack-allocated objects, as explained next.

\footnote{Similarly as for $W$, $R$ needs to be sufficiently large so that it always holds $l.\Delta > 1$.}
4.2.3 Objects with Non-Static Storage Duration

Additional care is needed to preserve the sequential execution semantics in the presence of objects with non-static (dynamic and automatic) duration. When tasks access heap or stack-allocated objects, preserving dependences between tasks does not suffice to preserve sequential execution semantics. Consider the example given in Figure 4.6. The main function invokes two functions (f2 and f3) that invoke function f4. One invocation of f4 accesses variable a, while the other accesses variable b. The important thing to observe is that although these variables represent different objects, they share the same storage.

```c
void f2 () {  
   int a;  
   f4(&a); // accesses only a  
}

void f3 () {  
   int b;  
   f4(&b); // accesses only b  
}

int main() {  
   f2();  
   f3();  
}
```

Figure 4.6: An example showing that, in the presence of non-static memory, preserving precedences between tasks does not suffice to preserve sequential execution semantics.

Now consider the parallel execution in which only the two function calls to f4 are executed asynchronously. The variables a and b still share the same storage, the stack of the main task. There are no dependences between the two tasks executing the calls to f4. However, due to the fact that the same memory region is used to store a and b, accesses that the tasks make to a and b need to be executed in a serializable manner, i.e., results of these accesses need to be identical to the results of an execution in which all accesses to one object preceded accesses to the other object.

---

7The order between these two group of accesses is not important. Indeed, a parallel execution in
Since different memory areas are used for automatic and dynamic allocation, two objects can share the same storage only if they are both automatically (stored on the stack) or both dynamically (stored in the heap) allocated. Further, because versioning maintains the proper ordering of accesses on a single object\(^8\), it suffices to ensure that the first access on a certain object happens after the last access on some other object that uses the same storage. Versioning handles heap and stack-allocated objects differently, as explained below.

In order that two dynamically allocated objects share storage, one object has to be explicitly deallocated (e.g., by calling \texttt{free} routine) before the other one is allocated. Hence, it suffices to ensure that before a task deallocates an object, all preceding tasks (if any) have finished accessing it. By the definition of exposed accesses, a deallocation of dynamically allocated object that is potentially accessed by multiple tasks is considered an exposed write access. Therefore, if the acquire algorithm is run for addresses that correspond to the object being deallocated, before the deallocation is executed, it will be ensured that all preceding tasks have finished. Roko accomplishes this by using the memory allocation hooks mechanism \[^{[49]}\]. Specifically, a call to \texttt{free} routine is redirected to a custom routine in which the acquire algorithm is invoked before the actual deallocation occurs.

For automatically allocated objects that share storage, versioning ensures serializability of accesses by exploiting the earlier mentioned fact that the same version number can be used to synchronize accesses to different objects. Consider the example given previously in Figure 4.6. Due to the sequential execution order within a single task, the \textit{main} task will first create the task that accesses \texttt{a} (we refer to it as \textit{taskA}). Since the access to \texttt{a} done by \textit{taskA} is exposed, the exposed table of \textit{taskA} contains a region corresponding to variable \texttt{a}. Consequently, the generate algorithm will insert an entry to LVTs of the

---

\(^8\)This was previously discussed for static objects, however, the claim holds for non-static objects as well, as shown in Appendix A
main task and taskA for that region. Specifically, the acquire version numbers of taskA and the main task for the memory region being considered will be set to 0 and $\lfloor \frac{W}{2} \rfloor$, respectively. Next, when the task that accesses b (taskB) is created, since the access to b is exposed and a and b share the same storage, the exposed table of taskB will contain an entry for the same memory region used to store a. Since the LVT of the main task already contains an entry for that region, taskB inherits the acquire version number of the main task for that region ($\lfloor \frac{W}{2} \rfloor$). Hence, accesses made by taskB will be executed only after taskA finishes. Unlike in earlier work on task-level parallelism [1], no additional synchronization is required at exit points of a program block to maintain the sequential execution semantics, and thus a greater degree of parallelism can be achieved.

### 4.3 Variable-granularity Versioning

An important property of the scheme presented in the previous section is the granularity at which version numbers are assigned. In general, choosing a coarse granularity results in a smaller number of version numbers than for the finer granularity case, and consequently, reduces the space needed to hold the GVT and LVTs. On the other hand, a coarse granularity can limit parallelism due to false sharing. The example in Figure 4.7 is used to illustrate this tradeoff. The figure shows a program that consists of a main function that invokes the function childA and eventually reads both members of the global structure s. The function childA invokes the function childB, increases both members of s and then returns. The function childB increases only s.b and returns.

The parallel execution in the case of word-granularity versioning is shown Figure 4.8a. When main task creates task childA, two version numbers are created, which will be used to synchronize accesses to variables s.a and s.b. Also, entries corresponding to these variables are added to the LVTs of main and childA in order to ensure that main can

---

9 Assuming that an integer is of a word size.
access \texttt{s.a} and \texttt{s.b} only after \texttt{childA} finish. When \texttt{childA} creates task \texttt{childB}, only the entry for \texttt{s.b} in the LVT of \texttt{childA} is updated and a new entry, for the same variable, is added to the LVT of \texttt{childB}. Consequently, when \texttt{childA} tries to access \texttt{s.a} it can proceed. However, when \texttt{childA} attempts to access \texttt{s.b} while \texttt{childB} is still executing, as shown in the figure, it must wait. Once that \texttt{childB} completes execution, it updates the version number of \texttt{s.b}, allowing \texttt{childA} to continue and increase \texttt{s.b}. Subsequently, when \texttt{childA} completes execution, it updates version numbers of \texttt{s.a} and \texttt{s.b}, allowing \texttt{main} to continue.

```
struct{
  int  a;
  int  b;
} s;

int  main(){
  ...
  #pragma parallel
  childA();
  ...
  return (s.a+s.b);
}

#pragma exposed  s
void  childA(){
  ...
  #pragma parallel
  childB();
  ...
  s.a++;
  ...
  s.b++;
  ...
}

#pragma exposed  s.b
void  childB(){
  ...
  s.b++;
  ...
}
```

Figure 4.7: A simple program used to illustrate the impact of versioning granularity.

Figure 4.8b shows the parallel execution of the same program when the double-word-granularity versioning is used. Here, the same version number is used to synchronize
accesses to both members of structure s\textsuperscript{10}. Consequently, in order to access s.a, childA will have to wait for childB to finish, although performing this access before childB completes execution would not endanger sequential execution semantics. In effect, the part of the execution between accesses to s.a and s.b done by childA will be unnecessarily serialized with the execution of childB, leading to a longer execution time than in the word-granularity case. However, the execution under the double-word-granularity versioning requires only 1 GVT entry and 3 LVT entries, as opposed to 2 GVT entries and 5 LVT entries in the word-granularity case.

\textsuperscript{10}Assuming that structure s is aligned on a double-word boundary.
The variable-granularity versioning tackles the issues described above by adjusting the granularity at which version numbers are assigned during the program execution. Versioning number are first assigned at a coarser granularity and are made finer during the execution to prevent unnecessary serialization. In fixed-granularity versioning, a single task, during its entire execution, uses a single version number to synchronize accesses to a single memory region, i.e. within one task there is a one-to-one correspondence between memory regions and the version numbers. The variable-granularity versioning breaks this logical association. Here, the same task can use, over time, different version numbers to synchronize accesses to the same memory region. Moreover, for the same memory region, a task may examine one version number upon accessing a memory region and then, once it completes execution, either update a different version number or not update any version numbers at all.

Figure 4.9 depicts the parallel execution of the program from Figure 4.7 under variable-granularity versioning. Since the one-to-one correspondence between version numbers and memory regions no longer exists, the version numbers are marked with numbers. Also, since a task can use different version numbers for the same memory region during the execution of acquire and release algorithms, the acquire and release version numbers are written in a form \((a_{i_A}, r_{i_R})\) to denote that the acquire version number \(a\) is compared against the version number \(i_A\) and that the release number \(r\) is used to update the version number \(i_R\). If a version number is not to be updated at all, the symbol '-' is used for \(i_R\).

When \texttt{main} task creates task \texttt{childA}, the version number 1 is created, which is (at this phase) used to synchronize accesses to \(s\). Additionally, the entries corresponding to structure \(s\) are added to the LVTs of \texttt{main} and \texttt{childA}, in a similar manner as in the case of the double-word granularity scheme. The only difference is that, once it finishes, \texttt{main} will not update version number 1. Since \texttt{main} cannot have a succeeding task, signaling that \texttt{main} has finished is unnecessary. Other than that, until \texttt{childB} is created, variable versioning operates in the same way as the double-word-granularity versioning.
Once that \texttt{childA} creates task \texttt{childB}, due to the fact that \texttt{childB} accesses only \texttt{s.b} and not the whole structure \texttt{s}, a new version number, as shown in the figure, is created. Version number 2 will be used to ensure that \texttt{childA} accesses \texttt{s.b} only after \texttt{childB} finished, as it would happen in the sequential execution. Further, the entry in the LVT of \texttt{childA} corresponding to structure \texttt{s} is now \textit{fragmented} into two entries that correspond to variables \texttt{s.a} and \texttt{s.b}, and a new entry, for variable \texttt{s.b}, is inserted into the LVT of \texttt{childB}. In order to access \texttt{s.a}, \texttt{childA} must compare the same version number and the acquire version number used previously to access the complete structure \texttt{s}, which signals that \texttt{childB} does not access \texttt{s.b} so \texttt{childA} remains the “earliest” task to access this variable. On the other hand, for \texttt{childA} to access \texttt{s.b}, the version number 2 has to match \( \lceil \frac{W}{2} \rceil \), which happens only after \texttt{childB} finishes. Consequently, as depicted in Figure 4.9 \texttt{childA} is allowed to access \texttt{s.a}, but has to wait for \texttt{childB} to finish in order to access \texttt{s.b}. Meanwhile, \texttt{main} cannot access neither \texttt{s.a} nor \texttt{s.b}, since its acquire version number for structure \texttt{s} does not match the current value of version number 1.
When childB completes execution, it updates version number 2 allowing childA to increase s.b. Finally, when childA has finished, it sets version number 1 to \( \lceil \frac{W}{2} \rceil \), permitting main to continue. In effect, variable-granularity versioning resulted in identical execution as when word-granularity versioning is used. However, an additional version number was created only when it was needed, in order to avoid false sharing. Moreover, variable-granularity required 2 GVT entries and 4 LVT entries, which is less than when the word-granularity versioning was used.

We proceed by describing the changes that variable-granularity versioning system introduces to the structures and algorithms used by fixed-granularity versioning. Since variable-granularity versioning handles heap and stack-allocated objects in the same way as fixed-granularity versioning, issues that arise in the presence of non-static memory allocation are not discussed. Finally, in some cases, variable-granularity versioning hinders the available parallelism among multiple concurrent readers. This undesirable side-effect is discussed in Section 4.3.3.

### 4.3.1 Structures

The variable-granularity scheme uses the same types of structures as fixed-granularity versioning. No changes to the GVT are required, while new elements must be added to all LVT and ET entries, as explained.

An entry in an LVT is a 10-tuple denoted as \( l = (baddr, eaddr, read, i_A, a, i_R, r, \Delta, rs, b) \). The elements of an entry are described in Table 4.2. Similarly as in fixed-granularity versioning, the LVT of a task still contains one entry for each memory region that is accessed in an exposed manner by the task or by the children of the task. Here, however, since a region can be of arbitrary size, it is necessary to maintain both the start (\( baddr \)) and the end (\( eaddr \)) address of the region, and \( LVT[baddr, eaddr] \) is used to identify the LVT entry corresponding to the region \( [baddr, eaddr] \). Next, as described before, a task may update a different version number than the one used during execution of the acquire
algorithm, or it may not update any version numbers at all. Elements \( i_A, i_R \) and \( rs \) are introduced to handle this situation. Finally, the element \( b \) is introduced in order to allow generation of LVT entries for the parent and child task in cases when fragmentation occurs, as explained later.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( baddr )</td>
<td>start address</td>
<td>start address of the memory region</td>
</tr>
<tr>
<td>( eaddr )</td>
<td>end address</td>
<td>end address of the memory region</td>
</tr>
<tr>
<td>( read )</td>
<td>read</td>
<td>1 if the region is only read by the task, otherwise 0</td>
</tr>
<tr>
<td>( i_A )</td>
<td>acquire index</td>
<td>index to the GVT used to access the version number that is checked when the task access the memory region</td>
</tr>
<tr>
<td>( a )</td>
<td>acquire version number</td>
<td>the value that the write version number pointed by ( i_A ) must be equal to in order for the task to access the memory region</td>
</tr>
<tr>
<td>( i_R )</td>
<td>release index</td>
<td>index to the GVT used to access the version number that will be updated</td>
</tr>
<tr>
<td>( r )</td>
<td>release version number</td>
<td>used to update the write version number pointed by ( i_R )</td>
</tr>
</tbody>
</table>
| \( \Delta \) | delta number | if \( read == 0 \), used to calculate the value to which the read version number pointed by \( i_A \) must be equal in order for the task to write to the memory region  
if \( read == 1 \), used to update the read version number pointed by \( i_R \) |
| \( rs \) | release | 1 if the entry causes update of the version number, otherwise 0            |
| \( u \) | usable | used by the generate algorithm to handle fragmentation of LVT entries. It has one of the four possible values: BOTH, ACQUIRE, RELEASE, NONE |

Table 4.2: Elements of an LVT entry in variable-granularity versioning.

An entry in an exposed table is a triple \( e = (baddr, eaddr, read) \), where the \( baddr \) and \( eaddr \) elements are introduced for the same reasons as in LVT entries. Similarly as in fixed-granularity case, the exposed table of a task is created by mapping the objects
that the task accesses in an exposed manner to memory regions that those object occupy. However, in variable-granularity versioning, the sizes of these regions are determined by the user. For example, if a user specifies that a task makes exposed accesses to the first 10 elements of some array \(a\) (by using \#pragma exposed \(a[0-9]\)), a single entry corresponding to the memory region of size 10 integers will be inserted into the exposed table of a task.

### 4.3.2 Algorithms

Similar to fixed-granularity versioning, variable-granularity scheme uses the acquire, release and generate algorithms. While changes to the acquire and release algorithm are minimal, the generate algorithm is more complex than in fixed-granularity case.

#### Acquire Algorithm

The pseudocode of the acquire algorithm is shown in Figure 4.10. The semantics of the acquire algorithm remain the same: the algorithm is used to check if a task is allowed to execute a memory access. The only change is the use of the \(i_A\) field to locate the version number in the GVT needed for comparison.

**Acquire**(*t*, *addr*, *read*)

1: if \(\exists l \in LVT_t\) such that \(l.baddr \leq addr \leq l.eaddr\) then
2: if \(read == 1\) then
3: while \(l.a \neq GVT[l.i_A].wv\) do
4: wait
5: else
6: while \((l.a \neq GVT[l.i_A].wv) \lor (l.\Delta + GVT[l.i_A].r \neq R)\) do
7: wait

Figure 4.10: The acquire algorithm: task *t* is trying to access memory location *addr*. The *read* parameter has a value of 1 in a case of a read access, and 0 otherwise.
Chapter 4. Versioning

Release Algorithm

The pseudocode of the release algorithm is shown in Figure 4.11. Similarly as in fixed-granularity scheme, the algorithm is used to signal tasks that all their preceding tasks (or writers, depending on the type of accesses to the region being considered) have finished. However, there are three differences. First, only entries with the release field set to 1 will cause updates (line 4). Second, the version number to be updated is pointed by the $i_R$ field. Finally, for an entry $l$ with $l.i_A \neq l.i_R$, even if $l.read$ is 1, the release procedure is done in the same way as for writers in fixed-granularity scheme. The rationale of this will become clear once the generate algorithm is explained.

$$\text{Release}(t)$$

1: for all $l \in \text{LVT}_t$ do
2: $read = l.read \&\& (l.i_A == l.i_p);$ 
3: $\text{Acquire}(t, l.addr, read);$ 
4: if $l.rs == 1$ then 
5: if $read == 1$ then 
6: $GV T[l.i_R].r = GVT[l.i_R].r + l.\Delta;$ 
7: else 
8: $GV T[l.i_R].wv = l.r;$ 
9: $GV T[l.i_R].rv = 0;$

Figure 4.11: The release algorithm.

Generate Algorithm

The pseudocode of the generate algorithm is displayed in Figure 4.12. Similarly to fixed-granularity case, the algorithm iterates over all regions in the exposed table of the child task. Here, however, every entry of the exposed table needs to be partitioned before LVTs can be updated (line 2). This is necessary since, as mentioned earlier, the sizes of regions in the exposed table are set implicitly by the user, while the sizes of the regions in LVTs are adjusted dynamically, during the execution. Therefore, every
region in the exposed table of the child task is first represented as an union of minimal number of regions such that each region of the union is either an existing region in the parent LVT, a subset of an existing region or a new region. For example, if an entry \( e' \) of the exposed table corresponds to the region \([0x4000A100, 0x4000A17F]\) and the LVT of the parent has only entries for regions \([0x4000A100, 0x4000A13F]\) and \([0x4000A160, 0x4000A18F]\) then \( P_{e'} = \{\{0x4000A100, 0x4000A13F\}, \{0x4000A140, 0x4000A15F\}, \{0x4000A160, 0x4000A17F\}\} \). Once the sizes of the regions from the exposed table are adjusted, the algorithm proceeds for every region \( e \) of the created partition.

---

**Figure 4.12:** The generate algorithm: task \( t_p \) is creating task \( t_c \). \( l \) is a temporary variable used to denote either the entry in parent’s LVT before modifications or a newly initialized entry.

---

The condition at line 4 tests if there exists a region in the LVT of the parent task that is a superset of \( e \). If this is not the case, using the same reasoning as in fixed-granularity
case, a new version number is created (lines 11-15) and new entries are generated and added to LVTs of the parent and child task in the same manner as in fixed-granularity scheme, as it will be explained later. On the other hand, if there exists a region in the LVT of the parent task that encloses \( e \), there could be two cases. First, these two regions could be identical, and in this case no fragmentation needs to occurs, since the child task does not accesses a finer grained region then its parent and hence no false sharing can occur (line 9). In the second case, when the region in the LVT is a strict superset of \( e \), a false sharing will occur if the granularity is not properly adjusted. However, fragmentation is not necessary when the parent task only reads the region from LVT and the child task\(^{11}\) also only reads the region corresponding to \( e \). Since both the parent and child tasks only read these regions, no parallelism is lost if the generation is done as in the case when \( e \) is equal to the larger region corresponding to \( l \). Consider the example where the parent only reads region \([0x4000A160, 0x4000A18F]\), and the child task only reads region \([0x4000A160, 0x4000A17F]\). Since versioning supports multiple consecutive readers, even if the generation is done as in the case where the child task reads the entire region \([0x4000A160, 0x4000A18F]\), both task will be able to simultaneously access this (larger) region.

The part of the algorithm that updates the LVT of the parent task and adds a new entry to the LVT of the child part in the case that fragmentation is necessary is shown in Figure 4.13. The entry in the LVT of the parent task is fragmented into two or three fragments, depending on the relations between the borders of \( e \) and \( l \). Lines 2-6 insert new entries to the LVT of the parent task that correspond to parts of \( l \) not accessed by the child task. This is how the LVT entry for \( s.a \) was generated once that \texttt{childA} created \texttt{childB} in the example from Figure 4.8. For these entries, the algorithm adjusts their borders, release and usable elements. The rationale of modifying the release field is

\(^{11}\)It suffices to check that the LVT entry is only read by the parent, since, by the definition of exposed accesses, this implies that the child task also only reads the region corresponding to \( e \).
explained next, while the meaning of usable element is described later.

**Generate_fragment**($t_p$, $t_c$, $e$, $l$)

1: $rs = l.r$;
2: **if** $l.baddr < e.baddr$ **then**
3: $LVT_t_p[l.baddr, e.baddr - 1] = (l.baddr, e.baddr - 1, l.read, l.i_A, l.a, l.i_R, l.r, l.\Delta, 0, \text{NONE});$
4: **if** $l.eaddr > e.eaddr$ **then**
5: $rs = 0$
6: $LVT_t_p[e.eaddr + 1, l.eaddr] = (e.eaddr + 1, l.eaddr, l.read, l.i_A, l.a, l.i_R, l.r, l.\Delta, l.rs, \text{NONE});$
7: $LVT_t_c[l.baddr, l.eaddr] = (l.baddr, l.eaddr, l.read, l.i_A, l.a, vc, \lceil W/2 \rceil, l.\Delta, 1, \text{RELEASE});$
8: $LVT_t_p[l.baddr, l.eaddr] = (l.baddr, l.eaddr, l.read, vc, \lceil W/2 \rceil, l.i_R, l.r, R, rs, \text{ACQUIRE});$
9: $GVT[vc].wv = 0;$
10: $GVT[vc].rv = 0;$
11: $vc = vc + 1;$

Figure 4.13: Generation of LVT entries for the parent and child task in the case that fragmentation occurs. $t_p$ and $t_c$ represent the parent and child task, respectively. $l$ represents an entry from the LVT of the parent task, while $e$ represents a part of the region corresponding to $l$ that is accessed by the child task. $rs$ is a temporary variable.

Consider the case where the LVT of the parent task contains an entry for region $m = [0x4000A100, 0x4000A17F]$ to which the parent task writes, and that the parent is supposed to signal that all its preceding tasks and itself, have finished accessing region $m$ by updating some version number, i.e., the release element of the LVT entry corresponding to $m$ is 1. Now, assume that the child tasks writes to the part $[0x4000A120, 0x4000A14f]$ of $m$. Hence, $m$ will be fragmented into three regions ([0x4000A100, 0x4000A11f], [0x4000A120, 0x4000A14f], [0x4000A150, 0x4000A17F]). To allow further progress, the parent still needs to signal the succeeding tasks about its completion. The question is for which of the three newly created entries should this signaling be done, i.e., which entry should have its release element set to 1. Assume that the entry corresponding to the fragment with the lowest staring address ([0x4000A100, 0x4000A11f]) has the release
element set to 1, and that the parent task also created a second child which accesses region [0x4000A120, 0x4000A14f]. If the release algorithm is executed in such order that regions are processed in the ascending order of their starting address\(^\text{12}\), i.e., the region [0x4000A100, 0x4000A11f] is processed first, the succeeding tasks would be signaled that the parent task and all its preceding tasks on the *entire* region \(m\), have completed execution. However, the second created child could in fact still be accessing a part of that region. This situation is avoided by ensuring that, when fragmentation occurs, only the fragment with the highest starting address inherits the value of the release element from the entry that has been fragmented, while others fragment have the release element set to 0.

Lines 7-11 in Figure 4.13 modify entry \(l\) of the LVT of the parent task, setting its border so it represents the fragment accessed by the child task. For this fragment, a new version number is created which is used to ensure that the parent task access the region only after the child task finishes. This is enforced by setting the acquire version number and acquire index of the entry in the LVT of the parent task equal to the release version number and release index of the entry added to the LVT of the child task, respectively. The generation of the LVT entries corresponding to \(s.b\) in Figure 4.8 is done using this part of the algorithm. Notice that this step is done equally for child tasks that read and write to the region being considered. Hence, even if the child task is a reader, when performing release algorithm it should set the write component of the version number to its release version number, i.e. it behaves like a writer in this situation as dictated by the release algorithm.

The part of the algorithm that handles generation of LVT entries when fragmentation does not occur is shown in Figure 4.14. We discuss only the case when the child is writing to a region, since implications of read-only regions are described in Section 4.3.3. For regions in the LVT of the parent task that have not been created as the result of previous

\(^\text{12}\)This is the case in our implementation of versioning, as described in Chapter 5
Generate\_no\_fragment(t_p, t_c, e, l)

1: if l.u = BOTH then
2: if e.read == 1 then
3: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, 1, l.i_A, l.a, l.i_R, N/A, \lfloor \frac{l.A}{2} \rfloor, 1, BOTH);\)
4: \(LVT_{t_p}[e.baddr, e.eaddr] = (e.baddr, e.eaddr, l.read, l.i_A, l.a, l.i_R, l.r, \lfloor \frac{l.A}{2} \rfloor, l.rs, BOTH);\)
5: else
6: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, 0, l.i_A, l.a, l.i_R, \lfloor \frac{l.A + l.r}{2} \rfloor, l.A, 1, BOTH);\)
7: \(LVT_{t_p}[e.baddr, e.eaddr] = (e.baddr, e.eaddr, l.read, l.i_A, l.a, l.i_R,\)
\[\lfloor \frac{l.A + l.r}{2} \rfloor, l.r, R, l.rs, BOTH);\)
8: else if l.u == ACQUIRE then
9: if e.read == 1 then
10: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, 1, l.i_A, l.a, l.i_A, N/A, \lfloor \frac{l.A}{2} \rfloor, 1, BOTH);\)
11: \(LVT_{t_p}[e.baddr, e.eaddr] = (e.baddr, e.eaddr, l.read, l.i_A, l.a, l.i_R, l.r,\)
\[\lfloor \frac{l.A}{2} \rfloor, l.rs, ACQUIRE);\)
12: else
13: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, 0, l.i_A, l.a, l.i_A, \lfloor \frac{l.A + W}{2} \rfloor, l.A, 1, BOTH);\)
14: \(LVT_{t_p}[l.baddr, l.eaddr] = (e.baddr, e.eaddr, l.read, l.i_A, \lfloor \frac{l.A + W}{2} \rfloor, l.i_R, l.r, R, l.rs, ACQUIRE);\)
15: else if l.u == RELEASE then
16: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, l.read, l.i_A, l.a, l.i_R, \lfloor \frac{l.r}{2} \rfloor, l.A, 1, RELEASE);\)
17: \(LVT_{t_p}[e.baddr, e.eaddr] = (e.baddr, e.eaddr, l.read, l.i_A, \lfloor \frac{l.r}{2} \rfloor, l.i_R, l.r, R, l.rs, BOTH);\)
18: else
19: \(LVT_{t_c}[l.baddr, l.eaddr] = (l.baddr, l.eaddr, l.read, l.i_A, l.a, vc, \lfloor \frac{W}{2} \rfloor, l.A, 1, RELEASE);\)
20: \(LVT_{t_p}[e.baddr, e.eaddr] = (e.baddr, e.eaddr, l.read, vc, \lfloor \frac{W}{2} \rfloor, l.i_R, l.r, R, l.rs, ACQUIRE);\)
21: \(GVT[vc].wv = 0\)
22: \(GVT[vc].rv = 0\)
23: \(vc = vc + 1;\)

Figure 4.14: Generation of LVT entries for the parent and child task in the case that fragmentation does not occur. \(t_p,\) and \(t_c\) represent the parent and child task, respectively. \(l\) represents an entry from the LVT of the parent task, while \(e\) represents a part of the region corresponding to \(l\) that is accessed by the child task.

fragmentation, the generation of acquire and release indices and version numbers for the parent and child task is straightforward, and done in the similar way as in fixed-
granularity case (lines 6-7). This is how the LVT entries were created once that main
created childA in the example from Figure 4.8. However, handling regions that have
been created as a result of previous fragmentation (i.e., LVT entries generated by the
procedure from Figure 4.13) is more involved.

Consider the case where, in the example from Figure 4.8 childB would fork task
childC that makes an exposed access to s.b. It is necessary to ensure that childB can
access s.b only after childC finished. However, this cannot be done in the way specified
by lines 6 and 7, simply because different version numbers are used for acquire and release
procedures. One option would be to use the same approach used for fragmenting regions
and create a new version number (i.e, generating entries \((0, \lceil W/2 \rceil)_{s.b}\) and \((\lceil W/2 \rceil, \lfloor W/2 \rfloor)_{s.b}\)
for childC and childB respectively, using the same notation as in Figure 4.8). However,
this could lead to a large number of version numbers and would hinder parallelism in
the case of concurrent readers as explained in Section 4.16. A similar issue arises when
childA task would, after creating childB, try to create task childD that makes an
exposed access to s.b. Variable-granularity versioning employs the usable element of an
LVT entry to tackle this issue. Figure 4.15 depicts how LVT entries are generated for
the above mentioned situation.

When the fragmentation of the entry for s happens, the usable element in the cor-
responding entries in the LVT of childA and childB is set to acquire and release,
respectively. When childA creates subsequent children that access s.b, it uses only its
acquire\(^{13}\) version number to generate the release version number of child tasks (lines 13-
14 in Figure 4.14). On the other hand, when childB creates subsequent children that
access s.b, it uses only its release version number to generate the release version number
of child tasks (lines 16-17). This procedure ensures that the release version numbers
generated for the ancestors of childA that access s.b will be in the interval \((0, \lceil W/2 \rceil)\)
while the release version numbers generated for the ancestors of childB will be from

\(^{13}\)Hence acquire as the value of the usable field
Chapter 4. Versioning

\([\lfloor \frac{W}{2} \rfloor, W]\), i.e., all tasks will have unique values of the release version numbers for \(s.b\).

\[
childA
\]

\[
childB
\]

\[
childC
\]

\[
childD
\]

Figure 4.15: The example showing the role of the usable element of an LVT entry in the generate algorithm.

In certain cases, the creation of new version numbers cannot be avoided (lines 19-23). This occurs when the value of the usable element is \texttt{NONE}.

\[\text{4.3.3 Concurrent Readers}\]

In the cases when fragmentation occurs, or when the usable element of an LVT entry of the parent LVT that corresponds to the region accessed by the child task is set to \texttt{RELEASE} or \texttt{NONE}, the generate algorithm proceeds in the same way no matter if the children task is a reader or not (as it can be seen from Figures 4.14 and 4.13). In these cases, variable-granularity versioning hinders available parallelism among multiple concurrent readers, as demonstrated in Figure 4.16. The figures shows a program where \texttt{taskA} makes exposed write and read accesses to first 10 elements of array \(a\) and forks 4 tasks that only read \(a[0]\).

Since the part of the algorithm that handles fragmentation does not differentiate between readers and writers, when the first child task is created, the LVT entries are generated in a such way that \texttt{taskA} will be able to read \(a[0]\) (line 8 in the figure), only after the child task finishes. Consequently, this leads to an unnecessary serialization if
taskA tries to read a[0] before the child task finishes, as it is assumed in the figure. However, subsequently created tasks will be able to proceed in parallel, both among each others and to the respect with taskA.

```c
# pragma exposed a[0-9]

void taskA(int *a) {
    ...
    int i, x;
    for (i = 0; i < 4; i++) {
        #pragma parallel
        taskB(a);
        ...
        x = a[0];
        ...
    }
    ...
}

# pragma exposed a[0] (read)

void taskB(int *a) {
    ...
    int x = a[0];
    ...
}
```

Figure 4.16: A simple program used to illustrate multiple concurrent readers in the presence of fragmentation.

### 4.4 Guarantees Offered by Versioning

Versioning provides guarantees expressed by the following theorem, the proof of which can be found in Appendix A.

**Theorem 4.4.1.** Versioning provides the following guarantees during a parallel run:

(i) Dependences between tasks are preserved

(ii) Accesses to different objects that share the same storage are serializable

(iii) No deadlocks can occur
Chapter 5

Hardware Support for Versioning

As described in the previous chapter, versioning maintains sequential execution semantics during parallel execution. This chapter describes our architectural support for variable-granularity versioning. The overview of the proposed support is given in the first section. Versioning requires that a comparison of version numbers is executed on every memory access. Section 5.2 describes a filtering mechanism that avoids the overhead of this comparison for most memory accesses. Follows the description of a hardware implementation of the most complex versioning algorithm, the generate algorithm. The chapter concludes with the description of instruction set architecture (ISA) extensions.

5.1 Overview of Hardware Support

Figure 5.1 depicts the placement of the main components of the hardware support for versioning in a typical microprocessor architecture. Our design requires two dedicated on-chip memories: one memory block holds the LVT of the currently running task, while the other one is used as a storage for a Counting Bloom Filter (CBF)\(^1\). The CBF is used to reduce the number of accesses to the LVT, as explained in Section 5.2.

\(^1\)A description of Bloom filters is given in Appendix B.
The Roko Coprocessor (RCP) implements the versioning algorithms and interfaces the CPU core in two ways: as the coprocessor to execute instructions added to the ISA (Section 5.4), and as a proxy to the cache controller to monitor memory accesses. In addition, RCP includes the virtualization logic for transferring the LVT during context switching from dedicated on-chip storage to the memory and vice versa. RCP also contains a register (not shown in the figure) that holds the base address of the GVT. There is no on-chip resources dedicated to the GVT; the GVT is a shared structure and as such is placed in the shared memory. The version counter is also stored in the shared memory, at the location that immediately precedes the GVT. In this way, the address of the version counter can be derived using the GVT base address register.

The rest of this section is organized as follows. The main mechanisms of the added hardware support are described first, followed by the discussion on how versioning numbers are represented and what are the implication of the used representation. The section concludes with a description of the modifications that need to be done to the processor core in order to integrate the proposed hardware support for versioning.

Figure 5.1: The organization of the main components (depicted with the shaded blocks) of hardware support for versioning in a typical microprocessor architecture.
5.1.1 The Main Mechanisms

The Execution of Memory Accesses

The execution of a memory access is displayed in Figure 5.2. On a memory accesses, in parallel with cache lookup, the CBF is accessed. The CBF holds the set of memory locations (the monitored set, Section 5.2.2) with the following characteristic: if a memory location is not in the monitored set it can be safely accessed without performing the acquire algorithm. Therefore, a CBF miss implies that the memory access can be executed without accessing the LVT and GVT. On the other hand, in the case of a CBF hit, the LVT is searched and the entry corresponding to the memory location being considered is identified, if such exists. If the entry is not found, the memory access can be executed, as dictated by the acquire algorithm. Otherwise, the RCP, using the GVT base address register and the acquire index element of the LVT entry, calculates the address of the corresponding version number and reads its value from the shared memory. Next, the comparison of version numbers is performed. If it is determined that the task can access the requested memory location, the original memory access is resumed and the LVT and the CBF are updated, as explained in Section 5.2.3. Otherwise, a trap is raised and the program control is transferred to a trap handler. The terms version miss-match and version miss-match handler are used to refer to this event and the corresponding trap handler, respectively. The version miss-match handler can, for instance, continue the execution of the same task by re-executing the faulting instruction or it can take the current task off the processor and replace it with a different one.

Transferring the LVT to the Memory

When a task is taken off the processor, its LVT must be stored to memory, in addition to saving processor registers. The RCP contains simple logic for transferring the contents.

\[^{2}The\ role\ of\ the\ acquired\ bit\ mentioned\ in\ the\ figure\ can\ be\ ignored\ for\ now.\ Its\ rationale\ is\ explained\ in\ Section\ 5.2.2.\]
of the on-chip LVT to the memory and vice versa. The contents of the CBF do not need to be saved, because they can be derived from the LVT, examining the acquired bit of LVT entries, as explained in Section 5.2.3. Similarly, the RCP contains the logic for transferring the previously saved LVT from the memory to dedicated on-chip storage. Also, during this action the content of the CBF is restored.
Chapter 5. Hardware Support for Versioning

Updating the GVT

The release algorithm is implemented as follows. The RCP walks through all the entries of the on-chip LVT, examines the release element of each entry and updates the GVT correspondingly. The addresses of version numbers that need to be updated are derived by adding the GVT base address register and the release index element of the LVT entry being considered.

However, before updating a certain version number, a test if the corresponding region can be accessed by the currently running task must be made, as dictated by the release algorithm. If a task cannot access the region being considered, a trap to version-miss match handler will be made, similarly as in the earlier described case of a memory access execution. Once the task continues execution, it should not execute the release algorithm for the LVT entries for which it has previously updated version numbers, since this can result with an erroneous execution. For example, increasing the same read version number twice may signal to some task that all its exposed predecessors have finished, even if this is not the case. To avoid this situation, the LVT entries are deleted as they are processed.

Support for the Generate Algorithm

An attractive attribute of versioning is that, on a task creation, only the content of the parent and child LVT needs to be modified, while other LVTs are unchanged. Therefore, when a currently running task creates a child task, all that is needed to do is to updated the on-chip LVT (which holds the LVT of currently running task, i.e., the parent task) and save the newly created LVT of the child task to the memory. Once the software chooses that the child task should be started, its LVT is transferred from the memory to dedicate on-chip storage.

The RCP contains hardware support for the generate algorithm. Custom hardware reads entries of the exposed table of the child task (stored in the memory), updates the LVT of the parent task and saves the newly created entries of the LVT of the child task.
to memory. Also, the content of the CBF is updated so that it reflects modifications done to the parent LVT.

However, implementing the generate algorithm in hardware is not as straightforward, mainly due to the reason that it may not always be possible to execute the generate algorithm. For instance, due to the limited on-chip resources devoted to the LVT, it could happen that there is no space left for additional entries that must be inserted. An additional reason stems from the finite number of bits used to represent version numbers, as explained below. The description of the hardware support for the generate algorithm and how it addresses failures is given in Section 5.3.

5.1.2 The Representation of Version Numbers

The write and read version component are represented as unsigned integers. The numbers of bits used to represent these components are parameters\(^3\) and do not influence the design decisions. However, the number of tasks that can be created depend on the values chosen for these parameters, for the following reason.

As mentioned in Section 4.2.2, in order for versioning to operate properly, when generating the acquire version number of a child task for some region, it must hold \(l.a > l.r + 1\) and \(l.\Delta > 1\), where \(l\) represents the LVT entry in the LVT of the parent task for the region being considered. Otherwise, since versioning cannot guarantee sequential execution semantics in this situation, the child task cannot be created. In theory, this situation can be avoided by setting \(R\) and \(W\) sufficiently large. However, in the implementation, due to the finite numbers of bits available to store version numbers, there exist an upper bound on the number of tasks that can be created.

To make the situation in which a child task cannot be created less frequent, the generate algorithm can be modified in the following way. If a writer child task will not create any tasks, the release version number of the child task and the acquire version

---

\(^3\)The widths of the write and read version components do not have to match.
number of the parent task are set to to \( l.a + 1 \) instead of \( \lfloor \frac{l.a + l.r}{2} \rfloor \), where \( l \) represents the LVT entry in the LVT of the parent task for the region being considered. In the similar manner, for the reader child task, \( 1 \) and \( (l.\Delta - 1) \) are used instead of \( \lfloor \frac{l.\Delta}{2} \rfloor \) and \( \lceil \frac{l.\Delta}{2} \rceil \) as values for the child and parent delta numbers, respectively. This maximize the number of tasks that can be created, in the situation when it is known that a child task will not create any tasks. This information can be easily retrieved by employing standard call-graph analysis. Hence, the generate algorithm has an additional input parameter which determines if a child task creates any tasks\(^4\).

### 5.1.3 Processor Core Modifications

Since our hardware support is encapsulated as a coprocessor, the modifications done to the processor core are limited in scope. Here, we address only the modifications specific to the proposed hardware support. Other modifications, such as passing values of general purpose registers to the coprocessor, writing the result of the coprocessor instructions to the register file, or stalling the processor until the coprocessor finishes execution, are not specific to our design and would be needed in any system that uses a coprocessor.

**Processor Pipeline**

In the proposed designed, the CBF is accessed in parallel with the processor’s cache. In order to hide these accesses from the processor pipeline, it is crucial that the result of the CBF access is known no later than the cache tag comparison is done. Since the CBF is a much smaller structure than the cache, the processor pipeline should indeed be oblivious of CBF accesses. Memory accesses that, besides accessing the CBF, require lookups into the LVT and GVT are presented to the processor core as cache misses, stalling the pipeline until the comparison of version numbers is done.

A version miss-match is treated in the same way as any other instruction-induced

---

\(^4\)Roko performs this optimization.
exception. Therefore, it suffices to ensure that the signal coming from the RCP that
denotes this event is recognized by the processor, and that, when this signal is activated,
program control is transferred to the corresponding entry of the trap table. The re-use of
the already present exception handling mechanism for version miss-match events allows
graceful handling of cases when, due to the delayed branching mechanism, after re-
executing the faulting instruction, the next instruction to be executed is not the one that
follows the faulting instruction in the address space.

Cache Protocol

In order to read and modify the contents of the GVT, and transfer the on-chip LVT to and
from memory, the RCP must be able to execute memory accesses. This is accomplished
by multiplexing the input signals of the cache controller. When the RCP is not executing
one of the versioning instructions (Section 5.4) nor it is checking a memory access, the
cache input signals are driven by the processor core. Otherwise, the data cache signals
are driven by the RCP.

5.2 Filtering Memory Accesses

This section describes a filtering mechanism based on a Counting Bloom Filter (CBF).
The mechanism avoids the overhead associated with the LVT and GVT accesses. The
technique is similar to those that have been used to filter coherence traffic and reduce
energy consumption in lower-level caches in SMPs, and to decrease the cache and load-store queue (LSQ) pressure.

The section starts with the discussion on the necessity of filtering accesses to the
LVT. Next, the definition of the set that is held by the CBF in our proposed hardware is
given, followed by the description of how this set is constructed and maintained during
execution. Finally, we discuss the implications of the granularity at which CBF operates.
5.2.1 LVT Design Considerations

In variable-granularity versioning, an entry in LVT can represent a region of any size. Consequently, unlike a direct-mapped or an n-way set associative cache, where it can be identified if a certain memory location is in the cache by looking only at a few cache entries (that hold the regions of a fixed size: cache lines), the entire LVT has to be searched in order to find an entry (region) that contains a certain address. In other words, the LVT is inherently a fully associative structure.

Versioning dictates that LVT should be searched on every memory access. If the LVT would be accessed in parallel with the cache, and the result of an LVT search known before the cache tag comparison finished, no delay would occur unless an access to GVT is needed\(^5\). However, this procedure requires expensive hardware support, and it would potentially limit the maximum system frequency and increase the processor power dissipation.

To make the LVT search practical, all entries should be examined in parallel. FPGAs, which represent our target platform, do not provide a structure with these characteristics as an embedded block. Given that the maximum number of ports of a memory block that the FPGA devices currently support is two, implementing a parallel search of all entries using generic resources would be expensive in terms of both memory and logic blocks.

Secondly, even on platforms that provide structures whose entries can be accessed in parallel (such as content addressable memories, CAM, present in ASIC designs), their usage is limited by the access time and power consumption\(^{[24, 40]}\). The access time increases significantly with the associativity of a structure (e.g., for ASICs, 8-way associative caches have on average 1.43 times longer access time relative to the direct-mapped

\(^{5}\)The frequency of GVT accesses can be decreased by adding the *acquired* bit to each LVT entry, as it will be explained later.
caches for \[40\). Hence, requiring that the result of the search is known in one cycle\(^6\), would limit the maximum system frequency. Further, CAM structures dissipate much higher energy than SRAM because of their internal comparison logic and additional match lines \[24\].

Therefore, we resort to a sequential search of the LVT and use the CBF to eliminate unnecessary LVT searches for memory accesses that can be safely executed. The LVT is implemented as a singly-linked list, where each node contains one LVT entry and a pointer to its successive node. The nodes are ordered by the start address (\textit{baddr} element) of the region they correspond to. The details of our design of the LVT can be found in Appendix C.

### 5.2.2 Monitored Set

The monitored set is a set of memory locations with the characteristic that if a location is \textit{not} in the monitored set, it can be accessed without performing the acquire algorithm; i.e., accesses to this location will not compromise sequential execution semantics. It is important to identify the smallest set that satisfies the above mentioned condition, for the following reasons. First, due to the construction of the LVT as a linked list, the latency of a search operation can be considerable. Therefore, frequent LVT accesses could add a significant run-time overhead. Secondly, CBF is a probabilistic structure, and its false positive rate\(^7\) increases with the size of the set that it holds.

Ideally, the monitor set would contain only those locations for which a version miss-match would occur. Although, in general this information is not available before actually performing version comparison, the monitored set can be reduced by noticing that during certain execution phases, accesses to some memory location will certainly not result with version miss-matches. First, from the definition of the acquire algorithm, it follows that

---

\(^6\)The access time of the L1 cache is typically one cycle \[40\].

\(^7\)The false positive rate is the proportion of CBF accesses that were erroneously reported as being in the set that CBF is representing.
the monitored set need not hold memory locations for which there does not exist a region in the LVT that contains these memory location. Second, since the LVT of a task is modified only when the task creates a child task, once the task is allowed to access a certain region from the LVT, it will be allowed to access that region at least until the first subsequent child task is forked. When this child task is created, however, the parent acquire numbers for the regions accessed by the child task could be modified, so it cannot be longer guaranteed that a version miss-match will not occur.

Hence, the monitored set should be initialized when a task is created using only the memory locations for which corresponding entries exist in the LVT of the task. Once the task is allowed to access a certain region from the LVT, the corresponding memory locations can be safely deleted from the monitored set. When a child task is created, the locations corresponding to regions accessed by the child task should be added to the monitored set.

However, since the CBF is oblivious to the type of memory access (read or write), additional consideration is needed. Once a memory region has been successfully written to, the corresponding memory locations can be safely deleted from the monitored set, since if a task can write to a region, then it can read from it as well, as defined by the acquire algorithm. On the other hand, if a task have read from a certain region, it still may not be able to write to it. Consequently, it is not safe to remove the region from the monitored set, unless the region is only read by the task.

In order to keep track of regions for which a version comparison is not needed, an additional \textit{acquired} element is added to all LVT entries. The value of 1 denotes that access to the corresponding region will not cause a version miss-match. Hence, the monitored set of a task \( t \) is defined as:

\[
M_t = \{ \text{addr} : \exists l \in LV_T t \text{ such that } l.baddr \leq \text{addr} \leq l.eaddr \wedge l.acquired = 0 \}
\]
5.2.3 Maintaining the Monitored Set

As described above, the monitored set can be updated only during the execution of a memory access or when executing the generate algorithm. Then, it must be ensured that the acquired element of the LVT is set to a proper value and that the CBF is updated if needed. In addition, once the LVT of a task is brought from the memory to the on-chip storage, the CBF needs to be updated as well. These cases are described below.

The execution of a memory access was previously described using Figure 5.2. On a CBF hit, the LVT is searched and the entry that corresponds to the memory location being considered is determined. Although a CBF hit should imply that such entry does exist and that its acquired element is 0, due to the false positives of the CBF this does not have to always hold. In that case (the corresponding entry does not exist or its acquired element is 1), the access can be safely executed without accessing the GVT. If a CBF hit is not a false positive, the GVT is accessed and comparison of version numbers is made. Finally, if version-miss match event does not occur, the acquired bit of the corresponding LVT entry is updated and the memory locations belonging to that LVT entry are deleted from the CBF, as shown in the figure.

During execution of the generate algorithm, the acquired bits of the parent and child task are generated as follows. For regions accessed by the child task that were not in the LVT of the parent task, the acquired bit of the parent and child task is set to 0 and 1, respectively, signaling that the child can access the region without comparing version numbers while the parent cannot, since it has to wait for the child task to finish. If an existing region (or a region derived by fragmenting an existing region) in the parent LVT is accessed by the child task, the child task inherits the acquired bit of the corresponding LVT entry of the parent task. If both the parent and child task only read the region being consider, the acquired bit of the parent task is not modified. Otherwise, it is set to 0. The change of acquired bits in the LVT of the parent task is propagated to the CBF by adding and deleting the locations that corresponds to regions for which the acquired
bit is updated to 0 and 1, respectively.

Upon transferring the LVT from the memory to the dedicated on-chip storage, the content of the CBF is restored as follows. First, the CBF is cleared. Then, for every LVT entry for which the acquired bit is not set the corresponding memory locations are added to the CBF.

### 5.2.4 CBF Granularity

Depending on the CBF implementation, an element of the monitored set can correspond to a memory byte, word, double word, or to a memory area of some other size. However, the CBF granularity has an impact on the maximum system frequency and the hardware complexity as described below.

For example, if a CBF element corresponds to a memory byte, four accesses to the CBF have to be made when a processor issues a load word instruction, one for each byte of the corresponding word\(^8\). In order to hide the latency of these accesses, assuming that the cache lookup is executed in one cycle, it is necessary to execute them in one clock cycle, which may not be always possible, depending on the system frequency. Hence, the system frequency determines the lower bound of the CBF granularity.

Further, setting the CBF granularity larger than the minimum size of a memory region used by the versioning\(^9\), complicates maintaining of the consistency between LVT and CBF structures. For instance, before an element can be deleted from the CBF, it would have to be ensured that all the LVT entries that correspond to the CBF element have their acquired bit set to 1. While this could be done, it comes with the expense of more complicated hardware circuitry. Consequently, this gives an upper bound of the CBF granularity.

Since it is desirable to set the minimum granularity of versioning to lowest possible

\(^8\)Assuming 32-bit architecture.

\(^9\)Although variable-granularity adjusts the size of memory regions, the minimum size of a region could still be set to the value greater than a byte.
value (to avoid false sharing) both granularities should be equal and determined by the maximum number of CBF accesses that can be done during the cache lookup.

5.3 Hardware support for the Generate Algorithm

The generate algorithm is the most complex mechanism of variable-granularity versioning. Hardware limitations further complicate its implementation. In addition to the previously mentioned reasons due to which the generate algorithm may fail (the finite size of the LVT and the representation of version number), a failure can also occur in the case when it is not possible to create a new version number, due to the finite size of the memory area dedicated to the GVT.

A failure caused by any of these factors cannot be foreseen before the generate algorithm is started. Therefore, once the generate algorithm cannot proceed due to the factors described above, a recovery mechanism that will revert the structures to the state in which they were before the generate algorithm is invoked must be provided.

5.3.1 Operation

For the reason mentioned above, in the hardware implementation, the generate algorithm is executed in four phases, as displayed in Figure 5.3. Also, an additional state element is added to every LVT entry, the role of which is explained below.

Figure 5.3: Phases of the generate algorithm implemented in hardware
During the matching phase, ET entries are read from the memory, and for each entry the matching operation is executed. The matching operation creates the partition set of an ET entry, adds new entries to the parent LVT, fragments existing ones if necessary and updates the state element of LVT entries. The state element of an LVT entry contains the information about a type of access (read, write or none) of the child task to the memory region corresponding to the LVT entry, and specifies is the LVT entry created as a result of fragmentation. No other element of the LVT entries of the parent task is modified, nor the entries of the child task are generated. Actual modifications are deferred until the update phase.

In parallel with modifying the state element, a check for a failure due to the representation of version numbers is made. If a failure does occur for this reason, or due to the LVT capacity, the matching operation is stopped and the execution continues with the recovery phase. Otherwise, the version counter handling phase is entered. Finally, during the matching phase the total number of new version numbers that need to be created is calculated.

To simplify the creation of a new version number, it is assumed that the memory area allocated for the GVT is zeroed before execution. Hence, the creation of a version number corresponds to increasing the version counter. This is done during the version counter handling phase, where the version counter is first read from the shared memory, and then increased by the value calculated in the matching phase. These two operations are done atomically to avoid possible data race in the case when multiple tasks concurrently update the version counter. In addition, before performing the actual update, a check is made if the value of the version counter is greater than the maximum number of GVT entries, in which case the execution continues with the recovery phase.

In the update phase, every entry of the LVT of the parent task is inspected. Depending on the value of the state element, other elements of an entry are modified and the corresponding entry for the child LVT is generated and written into the memory. Also,
as explained in Section 5.2.3, the content of the CBF is updated.

During the recovery phase, the following actions are taken. First, the LVT entries added in the matching phase are deleted. Next, for fragmented entries, the value of the usable element is set to \textit{none}, and the release element of all but one fragments derived from the same region is set to zero. Although this does not revert the parent LVT to the state in which it was before the execution of the generate algorithm, it does bring the LVT in a state which does not compromise sequential correctness.

5.3.2 An Example

The execution of the generate algorithm is described on an example. The input structures (the ET entry of the child task and the LVT of the parent task) are shown in Figures 5.4a and 5.4b. As described in the previous section, after the first ET is fetched from the memory, the matching operation is started. The sequential part of the circuit that executes this operation consists of seven registers, and the values of these registers during the execution are depicted in Figure 5.4c. The state machine of the matching operation has three states: \textit{fetch}, \textit{analyze} and \textit{modify}.

The execution of the matching operation starts in the \textit{fetch} stage (column 1 in the figure). Here, the LVT is searched for an entry which contains the begin address of the ET entry, or 0x4000F2C0. Since such entry does not exists, the result of the search is the first entry such that the value of \textit{baddr} element is larger then the begin address of the ET entry, i.e., the second entry from the LVT of the parent task. This entry is stored in \textit{lvt_entry} register, as shown in the figure. Additionally, since this is the first executed cycle, the starting and ending address of the ET entry are stored in \textit{baddr} and \textit{eaddr} register. As the execution progresses, \textit{baddr} is increased, indicating the part of the ET entry that has been processed, while \textit{eaddr} is unchanged. Further, the register \textit{vc_inc}, that holds the number of new version numbers that need to be created for the ET entry being processed, is initialized to 0. The execution continues with the \textit{analyze} stage.
(a) The ET entry of the child task.

<table>
<thead>
<tr>
<th>baddr</th>
<th>eaddr</th>
<th>read</th>
<th>iₐ</th>
<th>a</th>
<th>iₐR</th>
<th>r</th>
<th>Δ</th>
<th>rs</th>
<th>aq</th>
<th>u</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000F2B0</td>
<td>0x4000F2BF</td>
<td>1</td>
<td>1</td>
<td>56</td>
<td>1</td>
<td>64</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>BOTH</td>
<td>OLD</td>
</tr>
<tr>
<td>0x4000F2D0</td>
<td>0x4000F2DF</td>
<td>0</td>
<td>2</td>
<td>48</td>
<td>2</td>
<td>64</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>NONE</td>
<td>OLD</td>
</tr>
<tr>
<td>0x4000F2E0</td>
<td>0x4000F2FF</td>
<td>0</td>
<td>5</td>
<td>32</td>
<td>2</td>
<td>64</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>ACQUIRE</td>
<td>OLD</td>
</tr>
</tbody>
</table>

(b) The LVT entry of the parent task before the generate algorithm is executed.

<table>
<thead>
<tr>
<th>stage</th>
<th>baddr</th>
<th>eaddr</th>
<th>read</th>
<th>iₐ</th>
<th>a</th>
<th>iₐR</th>
<th>r</th>
<th>Δ</th>
<th>rs</th>
<th>aq</th>
<th>u</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0x4000F2C0</td>
<td>0x4000F2C0</td>
<td>0x4000F2D0</td>
<td>0x4000F2D0</td>
<td>0x4000F2EF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>INSERT</td>
<td>INSERT</td>
<td>UPDATE</td>
<td>INSERT</td>
<td>INSERT</td>
<td>INSERT</td>
<td>INSERT</td>
<td>INSERT</td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(c) The contents of the main registers during the execution of the matching operation. Columns represent execution cycles. Shadowed cells represent a change of value. The LVT entries are denoted with an index which correspond to the enumeration used in Figure 5.4b.

<table>
<thead>
<tr>
<th>stage</th>
<th>baddr</th>
<th>eaddr</th>
<th>read</th>
<th>iₐ</th>
<th>a</th>
<th>iₐR</th>
<th>r</th>
<th>Δ</th>
<th>rs</th>
<th>aq</th>
<th>u</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0x4000F2C0</td>
<td>0x4000F2D0</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td>0x4000F2DF</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td>FRAGMENT</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

(d) The LVT entry of the parent task after the matching phase is executed. Shadowed cells represent a change of value.

Figure 5.4: The states of structures and registers during the execution of the matching operation.
In the analyze stage, registers \textit{baddr} and \textit{eaddr} are compared with the corresponding elements of the LVT entry stored in \textit{lvt\_entry}. This yields the first element of the partition set, region [0x4000F2C0, 0x4000F2CF]. The ending address of identified region is stored in the register \textit{t\_eaddr} (column 3 in the figure). Since that region does not exist in the LVT, a new entry needs to be inserted as implied by the \textit{action} register. Since adding a new region will result with the creation of a new version number, \textit{vc\_inc} is increased. The execution continues with the \textbf{modify} stage.

Updates of the LVT of the parent task happens in the \textbf{modify} stage. The current state of registers dictates that a new entry for region [0x4000F2C0, 0x4000F2CF] (the borders of regions are determined by the registers \textit{baddr} and \textit{t\_addr}) will be added to the LVT. The updated state of the parent LVT is shown in Figure 5.4d. The new read value of the state element denotes that this is a new entry, read by the child task. Further, the register \textit{baddr} is updated to reflect that the region [0x4000F2C0, 0x4000F2CF] has been processed (column 4 in Figure 5.4c). The execution continues with the \textbf{analyze} stage.

As explained above, during \textbf{analyze} stage registers \textit{baddr} and \textit{eaddr} are compared with the corresponding elements of the LVT entry stored in \textit{lvt\_entry}. Since begin addresses of the stored LVT entry matches the \textit{baddr} and the end address of the entry is smaller than \textit{eaddr}, the second element of the partition set is region [0x4000F2D0, 0x4000F2DF]. The corresponding entry will be updated during the next modify stage (column 5). The update read value of the state element of updated LVT entry (entry 3 in Figure 5.4d) denotes that this entry is not fragmented and that it is read by the child task.

Next, in the \textbf{analyze} stage (column 6), it is determined that the \textit{baddr} is greater than the end address of the LVT entry, signaling that the current LVT entry has been processed and that the subsequent one needs to be obtained. This is done during the \textbf{fetch} stage (column 7). During the comparison of \textit{baddr} and \textit{eaddr} register with the borders of the obtained LVT entry in the \textbf{analyze} stage (column 8), it is decided that
the next element of the partition set is [0x4000F2E0, 0x4000F2EF] and that the current LVT entry should be fragmented. This is done in the next stage (column 9), and the result of fragmentation is depicted in Figure [5.4d] (entries 4 and 5). Finally, in the next stage (column 10) since baddr is greater than eaddr, i.e., the entire region of the ET entry has been processed, the matching operation finishes.

The execution of the generate algorithm continues with the version counter handling phase, where the current value of the version counter is stored, and the version counter is increased by the value stored in $vc_{inc}$ register. In our example, we assume that the value of the version counter before the increase was 9. Hence, the version counter will be set to 11.

The final phase of the generate algorithm is the update phase. The state element of every entry in the parent LVT is inspected. Based on this information, the parent LVT is modified and the LVT of the child task is created. The final state of the LVT of the parent and child task, assuming $W = 128$ and $R = 64$, is shown in Figures [5.5a] and [5.5b].

<table>
<thead>
<tr>
<th>baddr</th>
<th>eaddr</th>
<th>recad</th>
<th>$i_A$</th>
<th>$i_R$</th>
<th>$r$</th>
<th>$\Delta$</th>
<th>rs</th>
<th>$aq$</th>
<th>$u$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4000F2B0</td>
<td>0x4000F2BF</td>
<td>1</td>
<td>1</td>
<td>56</td>
<td>1</td>
<td>64</td>
<td>11</td>
<td>1</td>
<td>BOTH</td>
</tr>
<tr>
<td>2</td>
<td>0x4000F2C0</td>
<td>0x4000F2CF</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>128</td>
<td>32</td>
<td>1</td>
<td>BOTH</td>
</tr>
<tr>
<td>3</td>
<td>0x4000F2D0</td>
<td>0x4000F2DF</td>
<td>0</td>
<td>10</td>
<td>64</td>
<td>2</td>
<td>64</td>
<td>24</td>
<td>0</td>
<td>ACQUIRE</td>
</tr>
<tr>
<td>4</td>
<td>0x4000F2E0</td>
<td>0x4000F2EF</td>
<td>0</td>
<td>5</td>
<td>32</td>
<td>2</td>
<td>64</td>
<td>16</td>
<td>0</td>
<td>ACQUIRE</td>
</tr>
<tr>
<td>5</td>
<td>0x4000F2F0</td>
<td>0x4000F2FF</td>
<td>0</td>
<td>5</td>
<td>32</td>
<td>2</td>
<td>64</td>
<td>32</td>
<td>1</td>
<td>NONE</td>
</tr>
</tbody>
</table>

(a) The LVT entry of the parent task after the generate algorithm is executed.

<table>
<thead>
<tr>
<th>baddr</th>
<th>eaddr</th>
<th>recad</th>
<th>$i_A$</th>
<th>$i_R$</th>
<th>$r$</th>
<th>$\Delta$</th>
<th>rs</th>
<th>$aq$</th>
<th>$u$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4000F2C0</td>
<td>0x4000F2CF</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>-</td>
<td>32</td>
<td>1</td>
<td>BOTH</td>
</tr>
<tr>
<td>2</td>
<td>0x4000F2D0</td>
<td>0x4000F2DF</td>
<td>1</td>
<td>2</td>
<td>48</td>
<td>10</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>RELEASE</td>
</tr>
<tr>
<td>3</td>
<td>0x4000F2E0</td>
<td>0x4000F2EF</td>
<td>1</td>
<td>5</td>
<td>32</td>
<td>5</td>
<td>-</td>
<td>16</td>
<td>1</td>
<td>BOTH</td>
</tr>
</tbody>
</table>

(b) The LVT entry of the child task after the generate algorithm is executed.

Figure 5.5: The final state of the LVT of the parent and child task after executing the matching operation.

For our example, the first entry of the parent LVT does not require any update since
its value is set to \texttt{OLD}, signaling that the corresponding region is not accessed by the child task. The second entry is updated as dictated by the generation algorithm (line 11 from Figure 4.12 and lines 3-4 from Figure 4.14). Similarly, the third entry is updated respecting the lines 3-4 from Figure 4.14. Finally, the last two entries are the result of fragmentation and are therefore updated as dictated by lines 7-8 and 6 from Figure 4.13 respectively.

5.4 Instruction Set Architecture Extensions

In order to expose the proposed hardware support to software, several extensions to the instruction set architecture are required. Table 5.1 displays the versioning instructions and their suggested assembly language syntax.

\begin{verbatim}
  v_init   reg_{rs1}, reg_{rs2}
  v_monitor_enable
  v_monitor_disable
  v_acquire  reg_{rs1}, reg_{rs2}
  v_release
  v_generate  reg_{rs1}, reg_{rs2}, reg_{rd}
  v_store_lvt  reg_{rs}
  v_restore_lvt  reg_{rs}
\end{verbatim}

Table 5.1: ISA extensions. \texttt{reg} is an register name. Subscripts identify the source (\texttt{rs}) and destination (\texttt{rd}) operand.

The instruction \texttt{v_init} is used to communicate the address and the size of the GVT to the hardware. This instruction needs to be called only once, when the system is being initialized. Next, the instructions \texttt{v_monitor_enable} and \texttt{v_monitor_disable} turn the memory monitoring mechanism on and off, respectively. Using \texttt{v_acquire} instruction, a check if the currently running task can access the memory region whose borders are determined by the values stored in \texttt{reg}_{rs1} and \texttt{reg}_{rs2} can be made without actually accessing that region. This is needed in order to maintain the sequential execution semantics in the presence of dynamically allocated objects, as discussed in Section 4.2.3.
algorithm is started using the \texttt{v_release} instruction.

The instruction \texttt{v_generate} is used to start the generate algorithm. This instruction takes two input arguments. One source register contains the address of the memory area where the LVT of the child task should be stored. The lower-most bit of the other register determines if the task being created creates any child tasks or not, while the upper bits of the same register hold the word-aligned address of the exposed table of the child task. The value of the destination register reveals if the generate algorithm executed successfully or not.

Instructions \texttt{v_store_lvt} and \texttt{v_restore_lvt} are used to transfer the on-chip LVT to and from memory, respectively. The address of the memory area used as a storage for the on-chip LVT is passed through the source register.

### 5.4.1 Versioning Instruction Latencies

The latencies of the instructions described above vary depending on the state of the CBF, LVT, GVT and the input parameters of the instructions. However, under certain assumptions, for the majority of the instruction, the latencies can be predicted. Approximate analytical expressions quantifying the latencies of the versioning instructions are given in Table 5.2.

The latencies of the instruction \texttt{v_init}, \texttt{v_monitor_enable}, \texttt{v_monitor_disable} are 1 cycle, since these instruction perform only a write to RCP registers. The latency of the instruction \texttt{v_acquire} heavily depends on the state of the CBF, LVT and the GVT. Therefore, for the same input parameters, its latency can vary significantly, from 1 cycle (in the case of only a single CBF accesses) to thousands of cycles (when several LVT, CBF and GVT accesses are needed).

The execution of \texttt{v_release} consist of a walk through the LVT and updating the corresponding GVT entries. Therefore, under the assumption that version miss-match does not occur, the latency of this instruction scales with the number of LVT entries.
Chapter 5. Hardware Support for Versioning

The cost of updating an GVT entry depends on the fact if the corresponding regions is read-only or not. In the current implementation, an GVT entry has size of a memory word, and consequently, for read-only region, an atomic load-modify-store word sequence must be executed. Otherwise, only a simple write to memory is needed.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_init</td>
<td>1</td>
</tr>
<tr>
<td>v_monitor_enable</td>
<td>1</td>
</tr>
<tr>
<td>v_monitor_disable</td>
<td>1</td>
</tr>
<tr>
<td>v_acquire</td>
<td>−</td>
</tr>
<tr>
<td>v_release</td>
<td>$n_{LVT} \cdot \begin{cases} t_{store\text{-}word} &amp; \text{if regions are read-only} \ t_{atomic\text{-}load\text{-}store\text{-}word} &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>v_generate</td>
<td>$(n_{ET} + 1)(t_{load\text{-}double}) + \frac{n_{ET} \cdot n_{LVT}}{2} + 2(n_{ET} + 1)t_{store\text{-}double}$</td>
</tr>
<tr>
<td>v_store_lvt</td>
<td>$2(n_{LVT} + 1)t_{store\text{-}double}$</td>
</tr>
<tr>
<td>v_restore_lvt</td>
<td>$t_{clear\text{-}bf} + n \cdot \begin{cases} 2t_{load\text{-}double} &amp; \text{if } t_{load\text{-}double} &gt; r \ 2r &amp; \text{otherwise} \end{cases}$</td>
</tr>
</tbody>
</table>

Table 5.2: Approximate analytical expressions quantifying the latencies of the versioning instructions. The parameters $n_{LVT}$, $n_{ET}$, and $r$ represent the number of memory regions in the LVT and the ET and the size of a single region, respectively.

The execution of the instruction $v\_generate$ is divided into several phases. Assuming that the generate algorithm succeeds, the most significant amount of time is spent in the matching and updating phase. In the matching phase, for each ET entry that is loaded from the memory, the LVT is searched and the matching operation is executed. Under the assumption that each ET entry corresponds to exactly one LVT entry, the cost of the matching phase is approximately represented with the first two summands of the sum in Table 5.2. The first term quantifies the time needed to load $(n + 1)$ ET entries from memory\textsuperscript{10}. In the current implementation, an ET entry has size of a double-word. The second term is due to the $n_{ET}$ searches of LVT; the average cost of one search is

\textsuperscript{10}The last, $(n + 1)$th ET entry is a dummy entry denoting the end of ET. Similarly, a dummy entry denotes the end of an LVT stored in memory.
estimated to \( \frac{n}{2} \). Next, during the update phase, the on-chip LVT is updated, the child
LVT is stored to memory, and the CBF is updated if needed. These three operations
are overlapped; hence, the duration of the update phase is determined by the operation
with the longest latency. For simplicity, we ignore the latency of CBF accesses: the
CBF needs to be updated only when acquired bit of a parent LVT entry changes from
1 to 0. Hence, the cost of this operation can be neglected in the case when the parent
creates a number of tasks that all accesses the same memory region, as it is often the
case. Consequently, the cost of update phase is equal to storing \((n + 1)\) newly generated
LVT entries to memory. In the current implementation, the size of an LVT entry is two
double words.

The instruction \texttt{v_store_lvt} transfers the content of the on-chip LVT to the memory.
Consequently, the latency of this instruction is \(2(n_{LVT}+1)t_{\text{store\_double\_word}}\). Upon executing
the \texttt{v_restore_lvt} instruction, both the LVT and CBF need to be updated. First, the
CBF is cleared. The latency of this operation corresponds to the number of counters
in one CBF block, and it is represented with the term \(t_{\text{clear\_cbf}}\). Next, for each entry in
the LVT that is being restored three operations are executed: the LVT entry is loaded
from memory, the on-chip LVT is updated and the corresponding region is added to the
CBF (if the acquired bit is 0). As these operation are overlapped, the execution time of
the instruction is determined by the operation with the longest latency. In the current
implementation, it takes 2 cycles to add a double word to the CBF.
Chapter 6

The Roko Kernel and Compiler

The Roko kernel is a software environment that manages the execution of tasks on a shared memory multiprocessor (SMP) system with hardware support for versioning. The Roko compiler analyzes pragmas added to the original C program and carries out needed code transformation so that the program can be executed on the Roko kernel. This chapter describes these two components of the system.

6.1 The Roko Kernel

The Roko kernel is a program that provides an interface between an application and an SMP with hardware support for versioning and controls the creation, scheduling, execution and termination of tasks. The kernel executes the application code that has been previously modified by the Roko compiler, in a such way that each function call marked with the parallel pragma is replaced with the code that triggers the creation of a task.

The execution starts on one processor. Once the program point at which a new task is created has been reached, one of the other processors in the system starts to execute the newly created task. A processor that is executing a task continues to do so until the entire task (i.e., the task-function of the task) completes or a version miss-match occurs.
If the version miss-match occurs, the task may be suspend, in which case it will be taken off the processor and its execution will be continued at some later time. Meanwhile, the processor on which the version miss-match occurred executes some other task.

The rest of this section is organized as follows. First, the way how tasks are represented in the Roko kernel is presented, followed by the description of the scheduling policy. The section concludes with discussing the steps involved in the procedures of task creation and termination.

### 6.1.1 Task Representation

The kernel maintains certain metadata for each task in order to manage the tasks in the system. A task is represented by the *task descriptor*, similar to how processes are represented in the Linux kernel [13]. The task descriptor, besides the information needed for making scheduling decisions for the task (Section 6.1.2), contains the address of the function that will be executed by the task. Additionally, each task has its private memory area reserved for its stack, LVT, ET and the arguments of the function executed by the task. The memory layout of a task’s metadata, inspired by the Linux kernel, is shown in Figure 6.1.

Since the LVT and the ET of the task never need to simultaneously coexist (the LVT is created based on the ET, and after that the ET can be discarded), the same memory area is used to store these structures. The fixed memory layout enables accessing the arguments, the LVT/ET and the stack of the task by knowing only the address of the task descriptor. During execution, each processor maintains the address of the task descriptor of the currently running task, which allows the kernel to identify it.

---

1. The size of the memory area dedicated to the arguments is set to a sufficiently large value so that functions with a reasonable number of arguments are supported.
2. Some processors have special registers dedicated to the system code, so maintaining this information does not require any hardware modifications. For example, we use SPARC’s g6 register, which by the SPARC ABI [53] is not supposed to be used by applications. If a system register is not available, the address of the task identifier can be stored in the memory.
Figure 6.1: Memory layout of the metadata of a task.

The kernel reserves a memory area that is used to store the metadata of tasks. This memory area is divided into contiguous blocks, and each block is used to store the metadata of a single task. The kernel maintains a linked list where each node represents one block of the reserved memory area. Nodes are removed from the list when tasks are created and returned when tasks terminate.

6.1.2 Scheduling Policy

The scheduling policy decides which processor should execute a certain task. Additionally, when a version miss-match happens, the policy determines if the currently running task should be taken off the processor, in which case it also selects a task that should be executed instead.

There has been a great body of work on the topic of scheduling \[42, 43, 46, 50, 59, 68, 74\]. In general, choosing a scheduling policy in a non-multiprogramming\(^3\) environment is a tradeoff between load balancing (i.e., spreading execution equally over all processors), affinity (i.e., executing a task on a processor that can execute it the fastest) and scheduling overhead (i.e., time spent making scheduling decisions).

\(^3\)In the multiprogramming case the situation is more complex, since additional considerations (fairness, response time, etc.) must be taken into account. The Roko kernel executes only a single program.
In our case, there exists an additional dimension. Based on dependences between tasks, some tasks may be suspended as long as preceding tasks have not finished. While hardware support for versioning will ensure that dependences among tasks are respected (by reporting a version miss-match), performance-wise it would be beneficial to schedule tasks in respect to the precedence relation. Unfortunately, although the exact order of all tasks on all objects that they access is maintained by the hardware, this information is not exposed to the Roko kernel. Our design approach was to implement a low-overhead scheduling policy, that will respect sequential execution order of tasks and achieve a good load balance, while ignoring processor affinity. The implemented policy is presented next.

The kernel maintains a global, double-linked list of all tasks (scheduler list), in which tasks are ordered by their sequential execution order. The proper order is maintained by ensuring that when a task is created, it is (i.e., a pointer to its task descriptor is) inserted in this list immediately before its parent task. When not executing a task, processors execute the Roko kernel code which access the scheduler list, searching for the first task that is not active. If such a task exists, the processors starts executing it.

As shown in Figure 6.2, a task can be in one of three possible states: new, active, or suspended. The information about the state of a task is stored in the task descriptor of the task. When a task is first created its state is new. Once the execution of a task starts, the state of the task becomes active and it remains like that until the task finishes or becomes suspended upon a task switch. The task switch is the process during which the currently executing task is taken off the processor, and replaced by a different task. The task switch can occur upon a version-miss match, as describe next.

When a version miss-match is triggered by a processor, the control is transferred to the Roko kernel. Here, the scheduler list is searched for the first task that is not active, and that is placed before the currently running task in the list. If such a task exists, the task switch between the currently executed task and the selected one is performed. Otherwise, the execution continues with re-executing the faulting instruction of the currently running
task. This scheduling policy tries to avoid unnecessary task switches, speculating that a sequentially “earlier” task should be able to make further progress sooner than the sequentially “later” tasks. This assumption, however, does not hold if there are no dependencies between these tasks, when it could happen that the sequentially “later” task can make progress, while the sequentially “earlier” cannot. The performance impact of un-optimality of the implemented scheduler is discussed in Section 7.4.3.

![State transition diagram of a task’s lifetime.](image)

The following steps are performed on a task switch. First, the context of the currently running task is saved so it execution can be later resumed. The context of a task consists of processor registers and its LVT. Processor registers are saved on the stack of the task, and the current value of the stack pointer is stored in the task descriptor, so that register can be restored later. The LVT is transfered from the on-chip storage to the dedicated memory described earlier, by invoking the instruction `v_store_lvt`. Second, the state of the task is changed from active to suspended. Next, the LVT of the task selected to be executed next is loaded from the memory to the dedicated on-chip storage using the instruction `v_load_lvt`.

Finally, if the task state is suspended, its registers are restored and the execution continues by re-executing the faulting instruction of the selected task. On the other hand, if the state of the task selected to be executed next is new, the function that is supposed to be executed by this task (the address of which is stored in the task descriptor) is invoked.
6.1.3 Task Creation and Termination

The process of task creation consists of the following steps:

1. Allocating the memory for the metadata of the task being created
2. Creating the exposed table of the task
3. Executing the generate algorithm, i.e. invoking the instruction \texttt{v\_generate}
4. Storing the address of the function that will be executed by the task in the task descriptor
5. Copying the arguments of that function to the metadata of the task
6. Inserting the task to the scheduler list

The code that executes these operations is generated by the Roko Compiler, as explained in Section 6.2.2. The termination of a task involves executing the release algorithm (i.e., invoking the instruction \texttt{v\_release}) and returning the memory used to store the metadata of the task to the kernel.

As described in Section 5.3, the generate algorithm may fail. Also, it is possible for the kernel to run out of memory, so the memory for the metadata of the task cannot be provided. While these situation should be rare, the Roko kernel must provide a mechanism to handle them gracefully.

When the creation of a child task cannot be successfully completed, the execution can be either continued with running the function of the child task sequentially, or the parent task can be suspended, postponing the task creation and retrying it at some later point. While serializing the execution of the child task guarantees forward progress, it comes with the price of hindering potential parallelism. On the other hand, blocking the parent task until conditions for successful creation are present can lead to a deadlock. Consider the example where the sequentially youngest task is trying to create a task and this fails due to the fact that there is no memory left to store the metadata of the child task. It could happen that all other tasks are dependent on the task being considered, so they will not be able to make any forward progress as well. The situation is even worse for
failures originating from hardware constraints. For example, since the GVT entries are never deleted, once the GVT gets filled, the creation of any tasks that adds at least one entry to the GVT will fail.

The Roko kernel handles failures that occur during the creation of a task as follows. If the main task tries to create a task and fails, the main task is suspended until all other tasks terminate, returning the memory reserved for the metadata to the kernel. This cannot result in deadlock since the main task is the sequentially oldest task. Once all tasks (except the main task) have finished the LVT of the main task and the GVT are emptied, and the version counter is set to zero. This effectively returns the state of versioning structures to the state in which they were initially, at the beginning of the execution, removing all obstacles for subsequent task creations. Finally, the main task continues its execution from the point it was previously suspended.

In the case that a task creation fails, and the parent task is not the main task, the function of the child task is executed sequentially, which potentially hinders parallelism. However, Rinard reports that a 2-level task hierarchy is the most often case in practice.

6.2 The Roko Compiler

The Roko compiler analyzes pragmas added to the original program and produces transformed C code that will be executed on the Roko kernel. The operation of the Roko Compiler is divided into two phases: analysis and code generation. These are described next.

6.2.1 Analysis Phase

During the analysis phase, functions calls marked with the parallel pragma are identified and the expressions following the exposed pragmas are parsed. A semantic analysis
(type and scope checking) is performed on every such expression in order to determine if it conforms to the rules specified in Section 3.3.

A single expression can encapsulate accesses to multiple objects. The Roko compiler ensures that accesses to these objects are synchronized as well, i.e. it implicitly adds them to the set of exposed accesses of the function being analyzed. While parsing an expression, the Roko compiler builds a tree that eases the extraction of indirectly accessed objects. An example tree corresponding to the expression \texttt{head->next->next*->data[0:n-1]} is given in Figure 6.3.

```
Figure 6.3: The tree built by the Roko Compiler for the expression \texttt{head->next->next*->data[0:n-1]}, assuming that \texttt{data} field is an array (and not a pointer, in which case there would be an additional node representing accesses to the \texttt{data} field).
```

All the memory accesses that stem from the example expression can be easily extracted by performing a modified postorder traversal of this tree. First, the tree is traversed once in a postorder manner. For each visited node, except the node corresponding to the repeat operator (referred to as the repeat node), an expression denoting the accessed object is generated. For the example in the figure, this yields accesses to \texttt{head}, \texttt{head->next}, \texttt{n}, and \texttt{head->next->data[0:n-1]}. If there does not exist a repeat node, all the regions have been identified. Otherwise, traversal is repeated but only from the repeat node upwards and excluding the nodes that are not ancestors of the repeat node. For running example this yields additional accesses to \texttt{head->next->next->data[0:n-1]}, \texttt{head->next->next->next->data[0:n-1]}, etc.
Additionally, a standard call-graph analysis is also applied during the analysis phase. The result of call-graph analysis is the list of all the functions that can create tasks (directly or through their callees).

### 6.2.2 Code Transformation Phase

During the code transformation phase, the code of the `callback` and `taskify` functions is emitted, and the function calls marked with the parallel pragma are replaced with the code that handles the task creation.

**Generating Callback Functions**

A function that will be executed by a task must have a standardized interface. Therefore, the Roko Compiler generates, for each such function, a function that implements the same functionality but it has a standardized interface (callback function). A callback function receives only one argument, a pointer to the input structure. The input structure contains a member for each parameter of the original function. The callback function extracts the parameters from the input structure and invokes the original function. The callback function and the input structure for the function `sort` of the merge sort example introduced in Chapter 2 are shown in Figure 6.4.

```c
struct _ROKO_input_sort {
    int *d;
    int *t;
    int n;
};

void _ROKO_callback_sort (void *args) {
    sort (((struct _ROKO_input_sort *) args)->d, \
         (((struct _ROKO_input_sort *) args)->t, \n         ((struct _ROKO_input_sort *) args)->n);
}
```

Figure 6.4: The generated callback function and the input structure for `sort` function of the merge sort example.
Generating Taskify Functions

For each function that is marked with the pragma parallel, the Roko Compiler generates the \textit{taskify function} that will carry out the creation of the tasks that execute the invocation of the function being considered. The taskify function has the same argument list as the original function and returns a boolean value specifying if the creation of the task succeeded or not. The taskify function for the function \texttt{sort} of the merge sort example is shown in Figure 6.5.

The only involved step is the creation of exposed table. As explained in Section 4.3.1, an entry in the exposed table is a triple \( e = (baddr, eaddr, \text{read}) \). However, in our implementation, hardware actually expects the entry of the form \( e = (baddr, size, \text{read}) \), where \( size = eaddr - baddr + 1 \). This information is stored in two memory words. One word holds the starting address of the region. The other word holds the size and the read bit; the read bit is stored in the lowest bit, while the size is stored in the upper bits. For example, lines 11-12 in Figure 6.5 generate an ET entry that describes write accesses to array region \( d[0:n-1] \). As explained previously, the trees (one for each expression that follows the exposed pragma) constructed during the analysis phases are used to generate this code.

The hardware procedure of the task creation is invoked at line 20 in Figure 6.5. \texttt{ROKO\_asm\_generate} is a macro which invokes the instruction \texttt{v\_generate}. The second argument of the macro denotes that the \texttt{sort} function can create a task.

Transforming Functions Call Sites

Every function call marked with the \texttt{parallel} pragma is enclosed with a \texttt{ROKO\_parallel} macro. Transformations of the \texttt{sort} function of merge sort example are shown in Figure 6.6. This macro invokes the corresponding taskify function, and passes its result to the Roko kernel.
int _ROKO_taskify_sort(int *d, int *t, int n) {
struct _ROKO_task *task;

// 1) Allocating the memory
if ((task = _ROKO_memory_allocate()) == NULL)
    return _ROKO_CREATE_FAIL;

// 2) Creating the exposed table
int _ROKO_et_offset;
if (d == NULL) goto _ROKO_label_1;
*(GET_TASK_ET(task) + _ROKO_et_offset++) = &d[0];
*(GET_TASK_ET(task) + _ROKO_et_offset++) =
    (((n-1-0+1)*sizeof(d[0])) << 1) | _ROKO_WRITE;
    _label_ _ROKO_label_1:
if (t == NULL) goto _ROKO_label_2;
*(GET_TASK_ET(task) + _ROKO_et_offset++) = &t[0];
*(GET_TASK_ET(task) + _ROKO_et_offset++) =
    (((n-1-0+1)*sizeof(t[0])) << 1) | _ROKO_WRITE;
    _label_ _ROKO_label_2:
*(GET_TASK_ET(task) + _ROKO_et_offset) = 0;

// 3) Invoking the generate algorithm
if (_ROKO_asm_generate(task, _ROKO_TASK_CREATOR) ==
    _ROKO_GENERATE_FAIL) {
    _ROKO_memory_free(task);
    return _ROKO_CREATE_FAIL;
}

// 4) Setting the address of the callback function
task->func = _ROKO_callback_sort;

// 5) Copying the arguments of the callback function
((struct _ROKO_input_sort *)_ROKO_GET_ARGS(task))−>d = d;
((struct _ROKO_input_sort *)_ROKO_GET_ARGS(task))−>t = t;
((struct _ROKO_input_sort *)_ROKO_GET_ARGS(task))−>n = n;

// 6) Adding task to the scheduler list
_ROKO_scheduler_add(task);

return _ROKO_CREATE_SUCCESS;
}

Figure 6.5: The generated taskify function for sort function of the merge sort example.

As mentioned in Section 3.2, the parallel pragma can also (beside a simple function call) be followed by an expression of the form var = foo(args). This case is handled by observing that a function returning a value can always be transformed to a semantically equivalent function that does not return a value. The transformed function has an additional parameter which is a pointer to the variable designated to store the return
value of the original function. Also, the return statement is replaced by the expression which stores the return value to the memory location pointed by the added parameter.

```c
void sort(int *d, int *t, int n){
  if (n < CUTOFF) {
    insertionsort(d, d + n);
  } else {
    _ROKO_parallel(sort(d, t, n/4));
    _ROKO_parallel(sort(d + n/4, t + n/4, n/4));
    _ROKO_parallel(sort(d + 2*(n/4), t + 2*(n/4), n/4));
    _ROKO_parallel(sort(d + 3*(n/4), t + 3*(n/4), n-3*(n/4)));
    _ROKO_parallel(merge(d, d + n/4, t, n/4, n/4));
    _ROKO_parallel(merge(d + 2*(n/4), d + 3*(n/4), t + 2*(n/4), n/4, n-3*(n/4)));
    merge(t, t + 2*(n/4), d, 2*(n/4), n - 2*(n/4));
  }
}
```

Figure 6.6: The function sort of the merge sort example after transforming functions call sites.

An example of code transformations in a case of return values is shown in Figure 6.7. _ROKO_parallel_ret_ is a macro similar to the _ROKO_parallel_ macro, but it takes two argument. The first argument is used to call the corresponding taskify function, while the second one is an expression which is executed if Roko chooses to execute task-function sequentially. The input structure has an additional pointer (_ROKO_ret_) which is used in the callback function to save the return value of the task-function. Since the access to the object pointed to by the pointer _ROKO_ret_ inside the callback function is an exposed one, an additional entry is added to the exposed table (lines 14-15).
Figure 6.7: Code transformation for a task-function that returns a value.
Chapter 7

Experimental Evaluation

In this chapter the experimental evaluation of Roko is presented. First, the platform used for the evaluation is described, followed by the discussion on the methodology we adopt. The description of benchmarks is given next. Finally, the chapter concludes with the presentation of the obtained results.

7.1 Platform

This section describes the hardware and software components of the platform we have built in order to evaluate Roko.

7.1.1 Hardware Platform

To explore the feasibility of the hardware mechanisms proposed in Chapter 5, we have assembled an FPGA-based symmetric multiprocessor (SMP) system, and then modified its processing units in order to support versioning. A real-hardware FPGA platform is used instead of a simulation in order to verify that our proposed design can indeed be implemented. A simulation represents a higher level of abstraction, and it is easy to overlook the intricacies associated with the hardware implementation. For instance, the
issues mentioned in Section 5.2.1 would not appear in a simulation, leading to a different design that would not be feasible for hardware implementation. An Altera Stratix II FPGA device is used as the basis of our platform, the characteristics of which are shown in Table 7.1.

<table>
<thead>
<tr>
<th>FPGA Board</th>
<th>Altera Stratix II EP2S180 DSP Development Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Device</td>
<td>Altera Stratix II EP2S180F1020C3</td>
</tr>
<tr>
<td>Logic Elements (LEs)</td>
<td>179,400</td>
</tr>
<tr>
<td>On-chip RAM</td>
<td>9,383,040 bits</td>
</tr>
<tr>
<td>Off-chip SRAM</td>
<td>1 MB</td>
</tr>
<tr>
<td>Off-chip SDRAM</td>
<td>32 MB</td>
</tr>
</tbody>
</table>

Table 7.1: FPGA device and board characteristics.

The SMP system is based on the template design of the GRLIB IP library. This library is chosen because of its multiprocessing support, licensing (GNU GPL license) and portability. The library is an integrated set of logic cores, designed for system-on-chip (SOC) development. GRLIB is written in VHDL and it is mostly technology independent. Therefore, systems built on top of it can be implemented on both ASIC and FPGA technologies. As such, GRLIB comes with support for different CAD tools and target technologies. The design tools we use are listed in Table 7.2.

<table>
<thead>
<tr>
<th>IP cores</th>
<th>GRLIB IP library (release 1.0.20-b3403)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis CAD tool</td>
<td>Altera Quartus II 8.0</td>
</tr>
<tr>
<td>Debug Monitor</td>
<td>GRMON Academic version (release 1.1.35)</td>
</tr>
</tbody>
</table>

Table 7.2: Design tools.

The block diagram of our hardware prototype is shown in Figure 7.1. As a base processing unit we use a LEON3 processor, which is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The system interconnection is AMBA 2.0 AHB/APB, which is widely used as the on-chip bus for ARM processors. Shared memory is implemented using the off-chip SDRAM. The multiprocessor interrupt controller, together with the cache coherence protocol, enables symmetric multiprocessing. Communication with the platform is established through through the JTAG (used
for downloading executables and controlling the execution) and a serial port (program I/O). The main system characteristics are displayed in Table 7.3.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>1 - 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>LEON3</td>
</tr>
<tr>
<td>Interconnection</td>
<td>AMBA 2 AHB/APB</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>32 MB (SDRAM)</td>
</tr>
<tr>
<td>System Frequency</td>
<td>60 MHz</td>
</tr>
</tbody>
</table>

Table 7.3: SMP system characteristics.

The parameters of the LEON3 processor are depicted in Table 7.3. Although the GRLIB IP provides a multiply/divide and a floating point unit for LEON3 processor, the limited FPGA resources prevent us from using it. The cache system consists of separate data and instruction L1 caches. The L1 cache sizes, associativity and replacement policy are configurable. For both data and instruction cache, we choose the highest possible associativity and the cache line size that is supported by GRLIB. The cache size was se-
lected so that we could fit a SMP system with 4 processors on the target FPGA device\(^1\). The cache hit time is one clock cycle for load instructions and two clock cycles for store instructions. On a cache miss, for the instruction cache the whole cache line is fetched, while for the data cache only 4 (for a load byte, half-word or word instructions) or 8 (for a load double-word instruction) bytes of data are loaded. The data cache write policy is write-through with no-write allocate and cannot be modified. This policy enables simple implementation of cache coherence, which is done by snooping write accesses on the AHB bus and by invalidating cache lines when needed. To decrease the latency of write operations, a 1-entry double-word write buffer is used.

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>LEON3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way in-order, 6-stage pipeline</td>
<td></td>
</tr>
<tr>
<td>8 SPARC register windows</td>
<td></td>
</tr>
<tr>
<td>No MUL/DIV unit</td>
<td></td>
</tr>
<tr>
<td>No FPU</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Data Cache</th>
<th>16 KB, 4-way (LRU), 32 byte blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-through with no-write allocate on write-miss</td>
<td></td>
</tr>
<tr>
<td>1-entry (double word) write buffer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Instruction Cache</th>
<th>16 KB, 4-way (LRU), 32 byte blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction burst fetch enabled</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Processor characteristics.

The processors are modified to support versioning. As described in Section 5.1.3, the modifications of the processor core are fairly modest. Specifically, we have enhanced the exception handling mechanism so that a version miss-match, signaled to the processor by the Roko coprocessor, is presented to the software as an instruction-induced exception. In addition, the decode unit is changed in order to recognize versioning instructions and activate the Roko coprocessor. Rather than adding new instruction to the SPARC ISA\(^65\), the existing Coprocessor Operate (CPop) instructions are used, which alleviates the need to modify the compiler used to produce the SPARC machine code.

\(^{1}\)The maximum cache size supported by GRLIB is 256 KB + 256 KB (data + instruction)
The CBF is implemented similar to [29, 62], and the details of our implementation can be found in Appendix B. The CBF parameters, shown in Table 7.5, are strongly dependent on the application. For instance, applications with a small number of exposed accesses require only a small number of counters in the CBF to keep the false positive rate minuscule, while the same CBF configuration for applications with a larger number of exposed accesses will result with a significant false positive rate. Additionally, the optimal number of the hash functions depends on the current number of elements in the CBF and the total number of counters. Further, the width of counters must be chosen large enough to avoid overflow\(^2\). Since the design space spawned by these parameters was too large for us to investigate, we set the CBF parameters to the values typically used in hardware implementation of transactional memory [20, 75], where Bloom filters are used to approximate transactional read and write sets. The hash functions are implemented by bit-selection and bit-interleaving, identical to [62]. For the selected values, the CBF requires 4KB on-chip storage, as shown in Table 7.6.

The CBF granularity, and consequently, the minimum granularity of versioning (Section 5.2.4), is set to double-word, for the following reason. In the platform described above, the largest memory unit for which the cache tag comparison is made in one cycle is a double word (this happens on a load double word instruction). Hence, in order to hide the CBF latency, it must be possible to test, within one cycle, if a given double word is in the CBF. Since, in our implementation, the CBF access is executed in one cycle, this yields double-word granularity\(^3\).

The size of the on-chip memory dedicated to LVT (Table 7.6), was determined as follows. First, the maximum number of LVT entries (Table 7.5) is chosen so that this parameter is not a performance bottleneck for our set of benchmarks. Next, the size of an

\(^2\)When the overflow happens this is handled simply by not modifying the counter once it reaches its maximum value, and presents only a performance but not a correctness issue.

\(^3\)The granularity could have been decreased, for example by exploiting a dual-ported memory as the CBF storage, or having multiple CBFs accessed in parallel. However, since the double-word granularity did not cause fault sharing in our set of benchmarks, we did not do so.
LVT entry was determined. For a given number of LVT entries \((N_{LVT})\), the GVT entries \((N_{GVT})\) and constants \(W\) and \(R\) of versioning, the size of an LVT entry is calculated using the formula\(^4\) \((65 + 2 \times \lceil \log_2 W \rceil + \lceil \log_2 N_{GVT} \rceil) + \lceil \log_2 R \rceil + \lceil \log_2 N_{LVT} \rceil\). Since in most practical cases we expect that the number of version numbers will be greater than the number of entries in the LVT of any task, this yields \(N_{GVT} > N_{LVT}\). Hence, for the chosen number of LVT entries, the minimum width of an LVT entry is 86. The final values of the parameters are determined so that the width of an LVT entry is 128, for which the FPGA memory blocks used to store the LVT are fully utilized. Write and read version widths of 12 allow at least 4095 tasks in the case of 2-level task hierarchy, or task nesting of depth 12.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBF</td>
<td></td>
</tr>
<tr>
<td>Number of counters</td>
<td>1024</td>
</tr>
<tr>
<td>Number of hash functions</td>
<td>4</td>
</tr>
<tr>
<td>Counter width</td>
<td>4</td>
</tr>
<tr>
<td>Versioning</td>
<td></td>
</tr>
<tr>
<td>The width of read version numbers</td>
<td>12</td>
</tr>
<tr>
<td>The width of write version numbers</td>
<td>12</td>
</tr>
<tr>
<td>The maximum number of LVT entries</td>
<td>128</td>
</tr>
<tr>
<td>The maximum number of GVT entries</td>
<td>1024</td>
</tr>
<tr>
<td>Roko Kernel</td>
<td></td>
</tr>
<tr>
<td>Task stack size</td>
<td>8 KB</td>
</tr>
<tr>
<td>Task arguments size</td>
<td>64 Bytes</td>
</tr>
<tr>
<td>The maximum number of coexisting tasks</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 7.5: Roko parameters.

<table>
<thead>
<tr>
<th>On-chip (per processor)</th>
<th>CBF</th>
<th>4 Kb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LVT</td>
<td>16 Kb</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td>20 Kb</td>
</tr>
<tr>
<td>Shared memory</td>
<td>GVT</td>
<td>4 KB</td>
</tr>
<tr>
<td></td>
<td>Task memory (per task)</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td>1 MB</td>
</tr>
</tbody>
</table>

Table 7.6: Roko memory requirements for the values of the parameters from Table 7.5.

\(^4\)The first term represents the space needed for the \(baddr\) (28), \(eaddr\) (28), \(read\) (1), \(rs\) (1), \(aq\) (1), \(u\) (2) and \(s\) (4) elements. The second term is space needed to store the \(a\), \(i_A\), \(r\) and \(i_B\) elements. The third term corresponds to the delta element, while the last term is space needed for maintaining the pointer to the next LVT entry.
7.1.2 Software Platform

The Roko kernel is implemented in C and assembly. There are three parameters that can be adjusted when compiling the kernel, as shown previously in Table 7.5. The values of these parameters are chosen to match the characteristics of our benchmark applications. The total memory requirement of the Roko kernel for chosen parameters has been shown previously in Table 7.6.

The Roko compiler is implemented as a source-to-source compiler using the Clang development tool. We choose Clang due to its library-based architecture and the simplicity of its code base. In its current state, the Roko compiler operates on one source file at the time. For our set of benchmarks this posed no limitation. However, this constraint can be easily relaxed in future implementations of Roko.

The transformed source code of a benchmarks is compiled and linked to the code of the Roko kernel, producing a single executable which is then downloaded to the FPGA. We use the Bare-C Cross Compilation System (BCC), which is a GNU-based cross-compilation system for LEON2 and LEON3 processors. Specifically, the used system consists of a GNU C/C++ crosscompiler (3.4.4), GNU Binutils-2.16.1 and Newlib 1.13.0 Embedded C-library. All benchmarks are compiled with O2 optimization level.

7.2 Evaluation Methodology

We measure performance of Roko in terms of the speedup, which is defined as the ratio of the execution time of the original, non-transformed sequential program ($t_{\text{sequential}}$) to the execution time of the parallel program on $p$ processors ($t_{\text{parallel}}$). Ideally, $t_{\text{parallel}} = \frac{t_{\text{sequential}}}{p}$, yielding speedup of $p$ on $p$ processors. However, due to application characteristics, shared memory contention, task handling and overheads introduced by hardware support for versioning, less than ideal speedups are observed. In order to understand the impact of these factors, we carry out a breakdown analysis of execution time.
The aggregate execution time of the parallel program on \( p \) processors \( (T_{\text{parallel}}) \) is obtained by summing execution times on all processors, and can be expressed as:

\[
T_{\text{parallel}} = \sum_i t_{\text{parallel},i} = t_{\text{sequential}} + t_{\text{memory}} + t_{\text{monitoring\_overhead}} + t_{\text{task\_management}} + t_{\text{idle}}
\]

The term \( t_{\text{memory}} \) represents the increase in the execution time due to the memory subsystem. Usually, due to the memory contention, \( t_{\text{memory}} \) increases with the number of processors. However, due to cache effects (the working set size of a task can be smaller than the working set size of the entire application) \( t_{\text{memory}} \) can be negative as well. The term \( t_{\text{monitoring\_overhead}} \) denotes the increase in the execution time caused by Roko’s monitoring of memory accesses. Ideally, a delay should be avoided for accesses to all regions for which the version miss-match would not occur. However, due to the construction of monitored set \[5.2.2\] and the false positives of CBF, delays can occur for other memory accesses as well, leading to the increase of the \( t_{\text{monitoring\_overhead}} \) component. Overheads due to the task creation and termination, storing the currently running task when version miss-match occurs, and setting up the execution of the task that should be run next, are represented with the term \( t_{\text{task\_management}} \). Finally, \( t_{\text{idle}} \) embodies the time during which processors wait for tasks, not performing any useful work. This component increases with load imbalance and lack of parallelism.

In order to precisely quantify each component of the aggregate execution time, the built hardware is enhanced with counters that enable very detailed, but non-intrusive program analysis. For each execution, we collect the relevant statistics and report them.

### 7.3 Benchmarks

We have evaluated Roko using a set of six benchmarks. Two benchmarks are synthetic micro-benchmarks used to investigate the performance of the Roko Kernel and the latencies of the versioning instruction. Their descriptions and obtained results are presented in Sections \[7.4.1\] and \[7.4.2\]. The other four benchmarks are used to evaluate Roko on more
realistic applications: matrix multiplication, merge sort, search and water. The merge sort benchmark was described earlier, and the other three applications are described below.

### 7.3.1 Matrix Multiplication

The matrix multiplication benchmark multiplies two square matrices. The fragment of the code of this benchmark is shown in Figure 7.2. Function `multiplyRows` calculates \( i \)th to \((i + B - 1)\)th row of the result matrix. The computation is done in a blocked manner (with the size of a block \(10\))\(^5\), in order to improve cache reuse for large matrix sizes.

```c
#pragma exposed a[i * N : (i + B) * N - 1] (read)
#pragma exposed b[0 : N * N - 1] (read)
#pragma exposed c[i * N : (i + B) * N - 1]
void multiplyRows(int *a, int *b, int *c, int i);

void multiplyMatrix(C_TYPE *a, C_TYPE *b, C_TYPE *c){
    int i;
    for (i = 0; i < N; i += B){
        #pragma parallel
        multiplyRows(a, b, c, i);
    }
}
```

Figure 7.2: Matrix multiplication code fragment.

As shown in Figure 7.2, the function `multiplyRows` is marked for asynchronous execution. The function makes exposed read accesses to a region of array \(a\), to the whole array \(b\), and writes to the region of array \(c\). The four inserted pragmas are all that it is necessary to expose parallelism to Roko. During parallel execution, for a matrix of size \(n \times n\), the main task will create \(\frac{n}{10}\) child tasks, and each of these will calculate 10 rows of the target matrix.

\(^5\)For the cache size \(M\), the block size \(B\) should be chosen so \(B < \sqrt{\frac{M}{3}}\). For our platform \(B < 36.95\).
7.3.2 Search

Search application simulates the interaction of electron beams with solids [14], using a
Monte-Carlo technique. The result of this simulation is used for a comparison of an
empirical equation for electron scattering to a full quantum-mechanical expansion of
the wave equation stored in tables. The application was obtained from the set of Jade
benchmarks. The code fragment of this application is shown in Figure 7.3.

The simulation of electron beams is implemented in function back, which calculates
the backscattering coefficient for the given element and energy. The function receives
three arguments: a four element array point which stores the element information and
electron energy for the given electron-solid pair, a four element array coef holding the
simulation parameters, and the pointer to the variable in which the calculated scattering
coefficient should be stored. As the majority of the application time is spent in this
function, we consider this function for asynchronous execution.

```c
#pragma exposed pair[0 : 3] (read)
#pragma exposed coef[0 : 3] (read)
#pragma exposed *backsc

void back(double pair[], double coef[], double *backsc);

int main()
{
    double coef[4];
    double point[NPAIRS][4];
    double backsc[NPAIRS];
    ...
    for (i = 0; i < NPAIRS; i++){
        #pragma parallel
        back(&pairs[i][0], coef, &backsct[i]);
    }
    ...
}
```

Figure 7.3: Search code fragment.

In this benchmark, the only exposed accesses are to input arrays and the write of
calculated backscattering coefficient. The inserted exposed pragmas are shown in Figure
7.3. Further, before the function invocation, we have inserted the parallel pragma. This
is all the work needed to be done to expose parallelism to Roko.

### 7.3.3 Water

Water is a parallel benchmark from the SPLASH benchmark suite \[^{[64]}\]. The application evaluates forces and potentials in a system of water molecules in the liquid state. The sequential code of the application was obtained from the set of Jade benchmarks.

The computation is performed over a user-specified number of steps. Each iteration step consists of a several $O(n)$ and two $O(n^2)$ phases. One $O(n^2)$ phases calculates the total potential energy of the system, summing partial contribution of each molecule. The other phase computes the intermolecular forces acting on each molecule and the total system energy.

The main data structure is a set of eight large array, each representing the variable of the system: the first seven for the displacement and its first six derivates, and the last for the computed forces. Every array contains three values (corresponding to spatial directions) for each atom of each molecule. Furthermore, additional array stores center of mass (three doubles) for each molecule. Finally, there exists a globally shared structure, holding system parameters and potential and total energy of the system.

Rinard \[^{[60]}\] reported that good speedups can be achieved by only parallelizing the two $O(n^2)$ phases. The functions that carry on the execution of these phases contain a top loop, and each iteration of this loop performs the needed calculations for a subset of molecules. To extract parallelism on Roko, we first had to extract the body of this loop into a separate function, which will be executed asynchronously. The declaration of created functions are shown in Figure \[^{[7.4]}\]. We have marked the invocations of these functions using the parallel pragma. Next, the exposed accesses had to be determined.

The pointer d points to the eight large arrays, v is a pointer to the array storing center of mass, and dg is a pointer to the global structure. Both functions read parameters from *dg structure, this yields the first exposed pragma of these functions, as shown in the
figure. Similarly, the large displacement array and the center of mass of molecules are accessed within these functions, in a read-only manner. The function `inter_poteng_task` writes calculated potential energy to the variables `dg->g_pot[1]` and `dg->g_pot[2]`. The function `interf_task` function writes calculated intermolecular forces to the large array determined with parameter `DEST`, and updates the total energy (`*dg->vir`).

Although writes to shared data in these functions introduce dependences between tasks that will execute functions calls of the same function, these writes happen at the very end. Since Roko allows that the execution of tasks is overlapped even when there are dependences between tasks, a speedup should be observed.

```c
// Figure 7.4: Water code fragment.

#pragma exposed *dg (read)
#pragma exposed d[DISP][0 : dg->nmol - 1] (read)
#pragma exposed v[0 : dg->nmol - 1] (read)
#pragma exposed dg->pot[1:2]
void inter_poteng_task(struct glb *dg, double (***d)[NDIR][NATOM], double (**v)[NDIR], int p);

#pragma exposed *dg (read)
#pragma exposed d[DISP][0 : dg->nmol - 1] (read)
#pragma exposed v[0 : dg->nmol - 1] (read)
#pragma exposed dg->forces[p][0 : dg->g_NMOL - 1]
#pragma exposed d[DEST][0 : dg->nmol - 1]
#pragma exposed *dg->vir
void interf_task(struct glb *dg, double (**d)[NDIR][NATOM],
                double (**v)[NDIR], int DEST, int p);
```

Figure 7.4: Water code fragment.
7.4 Results

This section presents the obtained results. The performance of the Roko Kernel was analyzed first, followed by the investigation of the latencies of the versioning instructions. The section concludes with the results obtained when running the applications on Roko.

7.4.1 Roko Kernel Overheads

Here, we investigate the performance of the Roko Kernel for different task granularities, different number of tasks and across 1 to 4 processors. We first run the following experiment. The main task spawns a given number of children tasks and then yields until all the created tasks are executed. A child task executes an empty loop, the number of iteration of which controls task granularity. Tasks make no exposed accesses and during this experiment the versioning instruction are not invoked. The speedups for 96 child tasks\(^6\) and varying task granularity are shown in Figure 7.5. The figure shows that, for the granularity of 10000 cycles, good speedups are achieved. When a task reaches the granularity of 100000, the speedups are almost ideal.

Figure 7.5: Varying task granularity for a fixed number of tasks (96).

---
\(^6\)96 was chosen to have a perfect load balance for 2, 3, and 4 processors
The breakdown of task management overhead for the task granularity of 10000 cycles is shown in Figure 7.6. The figure shows that the overhead of task management is relatively small when compared to the task granularity, which leads to the observed good speedups. The relatively big overhead of storing and setting task’s context is due to the SPARC register window architecture: the context of a task includes all active register windows. In conclusion, the overhead of the Roko Kernel allows for fine grain tasks to execute efficiently and the overhead does not significantly increases with the number of processors, for the given number of tasks.

Next, we vary the number of tasks for a fixed granularity of 1000000 cycles. Figure 7.7 shows the resulting speedup when the number of tasks is divisible by the number of processors, resulting with perfect load balance. In this case, almost ideal speedup is obtained as soon as there are at least one task per processor\(^7\). The little decrease in speedup can be seen for 128 tasks. For the chosen values of Roko parameters, maximum of 128 tasks can be in flight at the time. Therefore, the creation of 128th child task will fail, and the corresponding action will be taken (Section 6.1.3). To conclude, no

\(^7\)In a case of a load imbalance, speedups are less than ideal for small number of tasks, but the gap between measured and ideal speedup decreases as the number of tasks increases.
significant performance degradation is observed for large number of tasks.

![Figure 7.7: Varying number of tasks for a fixed task granularity (1000000) for perfect load balance.](image)

**7.4.2 Versioning Instructions Latencies**

In this section the latencies of the instructions `v_generate`, `v_release`, `v_store_lvt`, and `v_restore_lvt` are investigated\(^8\). Similar to the previous section, the benchmark consists of the main task creating a given number of children tasks, of a certain granularity. Here, however, child tasks make exposed accesses to \( n \) consecutive memory regions of size \( r \) double words. The benchmark is run on 1 processor for a fixed number of child tasks (100) and a fixed granularity (1000000 cycles).

In the first experiment, the number of regions is varied, while the region size is held constant (1 double word). The results are shown in Figure 7.8. As it can be seen, the latency of `v_generate` instruction increases quadratically with the number of regions, while the latencies of other instructions scale linearly. In the second experiment, the number of regions is hold constant (50), while the size of a memory region is varied.

\(^8\)The latencies of the instruction `v_init`, `v_monitor_enable`, `v_monitor_disable` are 1 cycle. The latency of the instruction `v_acquire` heavily depends on the state of the CBF, LVT and the GVT, and input parameters of the instruction. Hence, the overhead of the acquire algorithm is analyzed in Section 7.4.3.
Figure 7.9 shows that the latency of \texttt{v\_restore\_lvt} instruction increases linearly with the region size, while latencies of other instructions remain constant. These results are in excellent agreement with the analytical expression presented in Section 5.4.1.

Figure 7.8: The latencies observed when varying the number of regions for a fixed region size of 1 double word.

Figure 7.9: The latencies observed when varying the region size for a fixed number of regions (50).
### 7.4.3 Applications

The inputs used to obtain applications speedup, together with the sequential execution times, are shown in Table 7.7. Table 7.8 describe the shape of the task-graph and the number of tasks created during the parallel execution. This table also shows the maximum observed number of LVT and GVT entries. Finally, the right-most column of the table indicates if fragmentation of LVT entries occurred.

<table>
<thead>
<tr>
<th>Application</th>
<th>Input set</th>
<th>Sequential execution time (millions of cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>200x200 integer matrix</td>
<td>343</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>array of 32367 doubles, cutoff parameter 513</td>
<td>3644</td>
</tr>
<tr>
<td>Search</td>
<td>51 electron-solid pair, 10 iterations per pair</td>
<td>4895</td>
</tr>
<tr>
<td>Water</td>
<td>343 molecules, 4 iteration steps</td>
<td>125176</td>
</tr>
</tbody>
</table>

Table 7.7: Applications characteristics.

<table>
<thead>
<tr>
<th>Application</th>
<th>Task hierarchy</th>
<th># tasks</th>
<th># LVT</th>
<th># GVT</th>
<th>Fragment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>2-level</td>
<td>21</td>
<td>multiply_rows: 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>main: 41</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>4-level</td>
<td>127</td>
<td>merge: 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>main, sort: 8</td>
<td>168</td>
<td>yes</td>
</tr>
<tr>
<td>Search</td>
<td>2-level</td>
<td>52</td>
<td>back: 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>main: 103</td>
<td>103</td>
<td>no</td>
</tr>
<tr>
<td>Water</td>
<td>2-level</td>
<td>33</td>
<td>inter_poteng_task: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>interf_task: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>main: 12</td>
<td>12</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 7.8: Characteristics of the parallel execution. For Water, the number of tasks, and consequently the number of LVT and GVT entries, depends on the number of processor, and shown numbers are for the execution on 4 processors.

The speedup of the applications is shown in Figure 7.10. The overall speedup is good, and for Search and Water close to ideal speedup is observed. The low speedup for the
execution of the merge sort benchmark on 3 processors is due to a high degree of load imbalance, caused by the scheduling policy.

Figure 7.10: The speedup of the benchmarks.

Consider the execution of the task graph shown earlier in Figure 2 on 3 processors. When the first four (in the sequential order) leaf sort tasks are created \((sort(0,3), sort(4,7), sort(8,11)\) and \(sort(12,15)\)), the first three of them will be scheduled to execute. After these are executed, one processor will start executing the fourth sort task and the other two processors will start executing the two tasks that follow in the sequential order: \(merge(0,7)\) and \(merge(8,15)\). The processor executing \(merge(8,15)\), will generate a version miss-match since this task is dependent on the task \(sort(12,15)\), which is still being executed. Since there are no task sequentially younger than \(merge(8,15)\) that are waiting to be executed, the processor will continuously try to re-execute faulting instruction, failing to do so until \(sort(12,15)\) finishes. However, there are \textit{sequentially older} tasks that can make further progress, such as \(sort(16,31)\). This clearly shows that scheduling the sequentially youngest task first does not always result with the optimal execution. We leave the improvements of the current scheduling policy for future work.
To further investigate the less-than-ideal speedups observed for the other benchmarks, we carry out a breakdown analysis of execution time, described earlier in Section 7.2. The results are shown in Table 7.9. The overheads of the memory contention and of monitoring memory accesses could not be explicitly measured, and the following approximation was used. For the memory contention, the ratio of memory accesses placed on the bus and the total number of instructions, measured in the sequential run and multiplied by 100, is reported. This indicates the memory pressure of the sequential execution, which should translate to the memory contention during the parallel execution. To quantify the overhead that stems from monitoring memory accesses, the percentage of instructions for which a delay due to monitoring has occurred is reported, i.e., the value shown in Table 7.9 represents the ratio of memory accesses for which the LVT lookup is made and the total number of instructions, multiplied by 100. The detail analysis of the impact of this overhead on the performance is carried out in Section 7.4.4. The overheads caused by task management and load imbalance are explicitly measured, and the two right-most columns of the table show the proportion of these overheads in the total execution time.

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Contention factor</th>
<th>Monitoring Overhead factor</th>
<th>Task Management</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>0.89</td>
<td>3.22</td>
<td>0.26%</td>
<td>0.60%</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>7.93</td>
<td>0.02</td>
<td>0.02%</td>
<td>1.32%      (26% for 3CPUs)</td>
</tr>
<tr>
<td>Search</td>
<td>2.40</td>
<td>0.00</td>
<td>0.07%</td>
<td>0.07%</td>
</tr>
<tr>
<td>Water</td>
<td>2.99</td>
<td>0.03</td>
<td>0.00%</td>
<td>2.97%</td>
</tr>
</tbody>
</table>

Table 7.9: Overhead analysis at 4 processors.

We make the following observations:

1. The matrix multiplication has the largest monitoring overhead, and this is the main reason why less than ideal speedup is obtained. The overhead of monitoring memory accesses is significant due to the large amount of data accessed in exposed
manner (each task accesses 220x200 integers). This translates to large monitored sets, which leads to the high false positive rate of the CBF, as explained in the next section.

2. The merge sort benchmark has the largest memory contention factor. Although the working set size of most tasks fits in the cache, the contention is the result of the fact that the cache is write-through, and consequently, all the writes are presented to the bus. Further, this application suffers from load imbalance since the “conquer” (merge) phase of the algorithm has to wait until the “divide” (sort) phase finishes. As previously explained, the poor result for 3 processors, is due to scheduling. Precisely, processor spent 26% of the execution time not performing useful work.

3. Search is a representative of an application with ideal characteristics for parallelization with Roko: a low degree of memory contention, a small amount of data to which exposed accesses are made, very good load balance, and a large task granularity. This makes the sum of overheads negligible when compared to the total execution time, leading to almost ideal speedup.

4. For Water, the biggest source of overhead is the load imbalance, due to the fact that only $O(n^2)$ phases have been parallelized, and for the given number of molecules (343), the amount of the work done sequentially is not completely negligible compared to the total execution time.

7.4.4 Filtering Memory Accesses

Figure 7.11 shows, for each benchmark, the percentage of memory accesses that resulted only with the access to the CBF, for which no delay occurred. For the rest of the memory accesses, the LVT is searched, and the average measured delay in terms of the additional cycles spent in accessing the LVT is shown in Table 7.10. Figure 7.12 shows a further
breakdown of LVT accesses. In case of false positives of the CBF, the memory access proceeds with no additional delay. Otherwise, the corresponding version number is read from the GVT, which adds more delay, the average value of which is shown in Table 7.10.

![Figure 7.11: The breakdown of memory accesses.](image1)

![Figure 7.12: The breakdown of LVT accesses.](image2)

The figures show that the CBF successfully filters the memory accesses, except in the case of the matrix multiplication benchmark. For this benchmark, the false positive rate of the CBF was 42%, due to the large large monitored sets of the tasks created for this benchmark. For the other benchmarks, the false positive rate of the CBF was miniscule, and exactly zero for Search.
### Table 7.10: Average LVT and GVT access times.

<table>
<thead>
<tr>
<th>Application</th>
<th>Average LVT access time (cycles)</th>
<th>Average GVT access time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Search</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Water</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

The impact of using different matrix size for the matrix multiplication benchmark is shown in Figure 7.13. The curve labeled *no cbf* denotes the performance observed when the CBF was “turned off”, i.e., all memory accesses resulted with a lookup in the LVT. The curve labeled *ideal*, on the other hand, presents the speedup obtained when the monitoring of memory access is disabled, showing the maximum speedup that can be obtained for this benchmark. The figure shows that, as the matrix size increases, the observed speedup decreases, and eventually becomes equal to the speedup obtained when no filtering mechanism is used. This is due to the fact that, as the size of the monitored set increases, the CBF counters overflow. Once that a CBF counter overflows, its value can never be decreased, leading to a large number of false positives. This shows that the used filtering mechanism does not perform well in the case when tasks access a very large amount of data to which exposed accesses are made.

To show the impact of filtering mechanism on the applications performance, we measured the speedup in the case when all memory accesses resulted with a lookup in the LVT, for the input parameters earlier presented in Table 7.7. The results are shown in Figure 7.14. When compared with Figure 7.10 it can be seen that filtering memory accesses provides a significant performance improvement.

---

9Since the children tasks of this benchmark are independent, this still results with a correct run.
Figure 7.13: The speedup of the matrix multiplication benchmark on 4 processors for different matrix sizes.

Figure 7.14: The speedup of the benchmarks run on the hardware with no CBF.
Chapter 8

Related Work

In this chapter, we present the summary of the related work. Section 8.1 surveys the previous work that tries to exploit task-level parallelism. Section 8.2 describes hardware schemes for dynamic monitoring of memory accesses. Finally, Section 8.3 discusses numbering schemes similar to versioning, but applied in different domains.

8.1 Task-level parallelism

There has been a number of systems that exploit task-level (thread-level, function-level, coarse-grain) parallelism. Roko draws inspiration from, and improves on three previous systems that extract dynamically available concurrency: Jade, pTask and zJava. These three systems are briefly described in Section 8.1.1 where we first compare them to Roko from the user perspective, and later discuss the differences in their implementations. Section 8.1.2 compares Roko to other systems that exploit task-level parallelism.

8.1.1 Jade, pTask, zJava

Jade is an implicitly parallel language, in which programmers augment sequential C programs with Jade constructs to enable parallel execution. In Jade, the user encloses a
part of the code that should be executed asynchronously with the \texttt{withonly} construct. Similar to Roko, it is the responsibility of the user to express accesses (\textit{side-effects}) done by this part of the code, and this information is used during run-time to extract and enforce dependences. However, Roko puts no restrictions on the objects to which accesses are made. In contrast, the Jade programmer, before expressing task’s accesses, must first annotate declarations of all objects to which multiple tasks will make accesses with the \texttt{shared} construct. This requires analysis of \textit{all code}, as opposed to Roko where it suffices to analyze the function that is executed asynchronously. Additionally, stack allocated objects cannot be shared. Next, after expressing accesses to the shared objects for a part of the code enclosed with the \texttt{withonly} construct, the Jade programmer also marks accesses in the code for which, during run-time, it will be checked if a task can execute the accesses without endangering sequential execution semantics or it has to wait. In Roko, this is avoided by monitoring memory accesses, but at the cost of additional hardware support. Finally, Jade determines dependences at the granularity of shared objects. For instance, if an array, when being declared, is annotated with \texttt{shared} construct, then tasks that write to disjoint areas of the array will be unnecessary serialized. To avoid this, the programmer would have to declare two separate arrays (each representing a fragment of the original array) and mark each with the \texttt{shared} object. In contrast, Roko employs variable-granularity versioning which adjust the granularity to application-specific sharing pattern.

\texttt{pTask} \footnote{1} is a truly automatic system for detecting and exploiting dynamic task-level parallelism in C programs. Hence, unlike Roko, \texttt{pTask} does not require any user assistance in order to parallelize programs. However, \texttt{pTask} is limited to array-based C programs. Like Roko, it does not suffer from false sharing, but under the cost of additional overhead as it will be explained later.

\texttt{zJava} \footnote{21} is a framework envisioned for automatic parallelization of general Java programs. \texttt{zJava} employs a compile-time analysis to extract \textit{data access summaries} of
Java methods that will be executed asynchronously, which correspond to our exposed accesses for C functions. Furthermore, during compile-time, zJava determines which accesses need to be checked in order to preserve sequential execution semantics and then insert calls to the run-time library which enforces the proper ordering of accesses. Unfortunately, although stated in [21] that the needed compiler analysis is simple due to the fact that there are no pointers in Java, the compiler part of zJava has never been built. Hence, in its current state, zJava requires more user assistance than Roko.

In order to maintain the proper order of accesses to objects shared among multiple tasks, all three above mentioned systems maintain object queues. For each shared object, there is a corresponding object queue that contains an entry for each task that accesses the object. These entries appear in the same order as the corresponding tasks would execute in sequential execution. Entries are added and removed from queues as tasks are created. A task can write to an object only when its entry is at the front of the queue. A task can read an object only when there are no writer tasks (i.e., their entries) in front of the task being considered. Versioning improves over this scheme in the following manner.

First, with object queues, to test whether a task can read an object, the object queue has to be searched in order to determine if there are any preceding writers. In versioning, on the other hand, all that is needed is a comparison of a few numbers. Second, when a task is being created and the corresponding entries added to object queues, no other task can access object queues. In effect, the monolithic nature of queues leads to unnecessary serialization. In contrast, with versioning, only the parent and child task are involved in the process of task creation, while other tasks can freely access memory, create new tasks or finish. Third, since versioning distributes information about the proper ordering between tasks, it is feasible for hardware implementation, while it is hard to envision an efficient hardware implementation of a monolithic, queue-based approach. Finally, avoiding false sharing with object queues involves more overhead than
the variable-granularity versioning scheme, as follows. Consider the situation where there
is an object queue for some array, and the task which access only a part of the array is
being created. In order to avoid false sharing, two actions can be taken. One option is to
insert the entry of the new task to the existing object queue, and to augment the entry
with the information about the range being accessed. Consequently, to test if a task
can access an array, all entries preceding the entry of the task being considered, need to
be checked for conflicting regions. This is approach taken by pTask. The other option,
taken by zJava is to “duplicate” the corresponding object queue, adjust the ranges and
inserts the task entry in only one object queue. Although this avoids traversing object
queues on each access, it introduces additional copying overhead. Versioning, as explained
in Section 4.3 resolve this issues by simply generating a new version number. However,
variable-granularity versioning potentially hinders parallelism among multiple concurrent
readers (see Section 4.3.3).

8.1.2 Other Related Systems

Rugina and Rinard [61] make use of complex analysis to extract parallelism in divide-and-
conquer algorithms that use pointer arithmetics to access arrays. In contrast to Roko,
they do not require any user input. However, Roko is capable of exploiting task-level
parallelism in general-purpose C programs, not limited to a certain type of algorithms
or to a certain type of pointer usage. Furthermore, Roko allows a higher degree of
parallelism by overlapping execution of tasks even when there exist a data dependence.

Cilk [35] is a multithreaded language for parallel programming. Cilk adds certain
keywords to C programming language, the most important of which are spawn and sync.
The keyword spawn has the same role as the pragma parallel in Roko, while sync is
used to explicitly enforce synchronization between tasks. In contrast, Roko requires that
users specifies exposed accesses, and then implicitly synchronizes accesses of tasks. Also,
by choosing to use pragmas instead of adding additional keywords, code augmented for
Roko can be compiled by standard C compilers and executed sequentially, alleviating the need of having separate sequential and parallel source code versions of the same program. Moreover, Cilk does not allow overlapping execution of tasks that are dependent. On the other hand, Cilk run-time system employs more elaborate scheduling and load balancing mechanisms (THE protocol), which could be applied in the future releases of Roko.

Thread-level speculation (TLS) \cite{66} has shown to be a promising parallelizing approach for irregular, pointer intense, and unanalyzable codes. Here, it is always optimistically assumed that no dependences between tasks exists. If it is detected, during execution, that there indeed exists a dependence between tasks, the execution is rolled back to the point where sequential execution semantics was still preserved. Although purely software approaches exist \cite{26, 54}, only hardware implementations \cite{38, 66} promise efficiency and low-enough overhead. In contrast to Roko, no user involvement or compiler analysis is necessary. However, thanks to user provided exposed accesses, Roko never needs to rollback the execution, and avoids overheads associated with that action. Further, hardware schemes for TLS track accesses at the granularity of the cache-line size of the underlying cache hierarchy and are therefore susceptible to false sharing \cite{15}. On the contrary, variable-granularity versioning used by Roko, adjusts to application-specific sharing pattern and avoids false sharing. Finally, both hardware TLS and Roko, use on-chip resources to hold the information about accesses made by tasks. However, while TLS has to record all accesses to be able to detect dependence violations, Roko stores the information only about the exposed accesses.

Colorama \cite{19} is an architectural support for data-centric synchronization. In this model, the programmer organizes data structures in *data consistency domains*, and the hardware support guarantees that the structures in the same domain are kept consistent with each other. In order to provide consistency, Colorama, similar to Roko, relies on a hardware support that monitors memory accesses, and notifies the software layer upon a first access to a data consistency domain. For this purpose, Colorama uses Mondrian
Memory Protection [71], described in the next section. Although Colorama eases parallel programming by releasing the user of the task of synchronization, the programmer still needs to have a global knowledge of the code in order to create data consistency domains. Moreover, Colorama employs a non-deterministic execution model and can cause deadlocks when locks are used as the underlying synchronization mechanism.

## 8.2 Memory Monitoring

Roko relies on the dynamic monitoring of memory access in order to precisely identify the accesses that need to be stalled to preserve sequential execution semantics. In order to decrease the overhead associated with this operation, Roko employs Bloom filters. Different approaches to dynamic monitoring of memory accesses are described in [8, 71, 77].

Mondrian Memory protection [71] is a fine-grained access protection scheme. Unlike traditional memory protection schemes, Mondrian allows to set access permissions at variable granularity, all the way down to a single byte. In order to make this scheme plausible, similar to Roko, every memory access must be monitored with low overhead. Hence, Mondrian uses an additional hardware support, mainly a ternary content addressable memory (CAM) structure, which holds the permission information about memory regions and can be accessed in parallel with the cache. Compared to our monitoring mechanism with Bloom filter, this approach is more precise, since a CAM is not a probabilistic structure. However, ternary CAMs are expensive, having greater power consumption, large size, cost and longer access time relative to SRAM [31].

iWatcher [77], an architectural support for dynamic monitoring of watched memory location, explicitly differentiate between large and small memory region, storing information about large regions in a small ternary CAM, and small regions into additional cache lines bits. Further, iWatcher requires a victim cache that stores cache lines of watched memory
locations that have been evicted from the cache. UFO builds on iWatcher, alleviating the need to use ternary CAM or a victim cache. However, this comes with the price of increasing the granularity to cache lines, and adding two bits per every cache block of data, in the whole memory hierarchy (caches, main memory, hard disk).

8.3 Numbering Schemes

Amza et al. use a numbering scheme to provide 1-copy serializability and scalability in replicated databases. In their model, a version number is maintained for each table in the database. Before a transaction is executed, it receives one number for each database table that it will access. Similarly to our work, it is the responsibility of the programmer to report all the tables that will be accessed by a transaction. When a transaction tries to update a certain table of some database replica, the number that the transaction received for the table is compared with the version number for that table. The described scheme enforces that all database replicas are updated in the same order, however, the order between transactions can be arbitrary, which simplifies the generation of version numbers. Indeed, their scheme can be viewed as a specialization of the fixed-granularity versioning to a 2-level task hierarchy.

A hardware TLS system proposed by Steffan et al. uses epoch numbers to enforce ordering between speculative threads. In order to do so, they assign consecutive numbers to consecutive epochs, which is trivial to accomplish in a case of a flat task hierarchy. In other cases, such as recursive creation of tasks, they create a “gap” in the epoch numbers, based on the compiler estimate of the recursion depth. This is similar to how versioning leaves a gap between the acquire and release version number of a child task when the compiler cannot prove that the child task will not create tasks. Roko could also make use of the compiler based estimate of the number of task that will be created by a certain task in order to make the number of the generate algorithm failures due to the reasons
mentioned in Section 5.1.2.

For XML documents, which are represented as a tree of XML nodes, numbering schemes are used to label the XML nodes so that the structural relationships (parent/child, ancestor/descendant, previous-sibling/following-sibling, previous/following) between nodes can be determined by comparing their labels without accessing the XML file [27, 39, 48, 69, 70]. Keller and Ullman [45] present a general scheme of labeling tree nodes, that can be used to determine if one node is an ancestor of another, and, if so, the path between these nodes can be easily determined. The similarity between these schemes and versioning is limited to the fact that they are both dynamic: the mentioned schemes support insertion and deletion of nodes in the tree, while versioning maintains order between tasks as they are created and terminated. The similarity, however, stops here, due to very different domains of application.
Chapter 9

Conclusions and Future Work

In this thesis, we presented Roko, a system that allows parallelization of sequential C codes with modest user assistance. To benefit from multiprocessors systems, the user starts from a sequential version of the code, then marks the function calls that should be executed in asynchronous manner, and provides the description of data usage for the corresponding functions. Based on the information expressed by the user with the \texttt{parallel} pragma and the \texttt{exposed} pragma, Roko distributes the execution over processors in the system, guaranteeing that the results of the parallel run will be the same as the results of the sequential execution.

Architecturally, Roko consists of three components: a compiler that analyzes pragmas, a run-time component that spreads the execution over multiple processors and a custom hardware support that implements versioning, a novel synchronization scheme. The experiment showed that the overheads stemming from the run-time component are low enough to allow efficient asynchronous execution of medium granularity (> 10000 cycles) functions. Although versioning could also be implemented in software, we presented its hardware design that minimizes the overheads associated with this scheme and takes the burden of inserting calls to synchronization routines from the compiler and/or the user. The proposed design can indeed be implemented in real hardware, as we shown...
by building a prototype SMP system with a hardware support for versioning on the FPGA platform. In our opinion, this gives a strong indication that versioning can be implemented on other platforms (e.g., ASIC) as well.

We believe that the required level of user assistance is low enough to make Roko attractive for the programming community. Also, as shown on the set of four benchmarks, Roko achieves good results in terms of application speedup. Of course, Roko should be evaluated more, both in terms of usability and performance, to make the final conclusion about its feasibility.

9.1 Future Work

There are two main directions in which the work presented in this thesis can be expanded: improvements of the current implementation of Roko, and extension of the experimental evaluation. These directions are discussed below.

Roko is a multi-layered (compiler, run-time, hardware) system, and each component could be improved. On the compiler side, the expressibility of the exposed pragma could be increased. For instance, when describing accesses to array regions, currently only the range for the lowest dimension (i.e., the rightmost dimension) of an array can be described, while accesses that span multiple consecutive elements in other dimension require the use of multiple exposed pragmas. Also, in its current state, the Roko compiler operates on one source file at the time and this constraint should be relaxed in the future releases. Next, the compiler could assist the user in expressing data usage of functions. For instance, the data usage of functions that do not use pointers could be extracted with a relatively simple compiler analysis, removing that burden from the user. Similarly, the compiler could assist the user in achieving the inclusion requirement, which proclaims that, besides accesses made directly by a function, all the functions called from the function being considered must be analyzed as well.
The scheduling policy currently employed by the Roko kernel, as shown on the obtained results, is not optimal. A better scheduling policy should use the information about exposed accesses of tasks; the information that is already present in versioning. The challenge is, however, how to extract and use that information without significantly increasing the overhead of scheduling. Also, the processor affinity should be taken into consideration as well.

Versioning could be improved both on the algorithmic and implementation sides. For instance, variable-granularity versioning in some cases hinders the parallelism available among multiple consecutive reader tasks. Future work could overcome this limitation. On the implementation side, the employed mechanism for filtering memory accesses could be further investigated and improved. The design space of the CBF parameters is large, and it was not explored in this thesis. Also, the CBF currently operates on the fixed granularity, which causes the decrease of performance when a large amount of data is accessed in an exposed manner. Therefore, it should be investigated if filtering mechanism could operate on the variable granularity, or if multiple CBFs, of different granularities can be used.

Finally, as mentioned in the previous section, Roko should be evaluated more to make a decisive conclusion about its usability. Therefore, a usability testing could be performed, which would give a direct input on how real users use Roko and how it compares to other programming models. This would be a valuable asset and guidance in the further work on Roko.
Appendix A

Correctness of Versioning

A.1 Task Graph and Chained Tasks

Definition A.1.1. A task graph of a program is a graph where each node corresponds to a task of the program and where two nodes are connected by an edge if and only if for the corresponding tasks holds that one task creates the other one.

Corollary. A task graph of a program is a tree.

Proof. It suffices to show that there exists a path between every two nodes in a task graph and that there does not exists any cycle. The first claim is a direct consequence of the fact that all tasks are created by the main task. The second claim is a direct consequence of the fact that each task is created by exactly one task.

Definition A.1.2. Tasks $t$ and $t'$ are chained on some memory address $addr$ if the path in the task graph from $t$ to $t'$ is such that each task of the path, except maybe the common ancestor of $t$ and $t'$, makes an exposed access to some object stored in $addr$.

Definition A.1.3. Let $t$ denotes some task, $p$ a program point inside that task, and $addr$ some memory address. A task $t'$ is a chained preceding task of the task $t$ on $addr$ at $p$ if and only if $t$ and $t'$ are chained on $addr$ and in the sequential execution $t'$ finishes before the program point $p$. 
Theorem A.1.1. Versioning provides the following guarantees during a parallel run:

(i) A task $t$ can write (read) a memory address $addr$ at some program point $p$ only if all the chained preceding tasks (writers) of $t$ on $addr$ at $p$ have finished.

(ii) A task $t$ can access a memory address $addr$ at some program point $p$ if all the entries from LVTs of all the chained preceding tasks of $t$ on $addr$ at $p$ have been released.

Theorem A.1.2. Versioning provides the following guarantees during a parallel run:

(i) Dependences between tasks are preserved

(ii) Accesses to different objects that share the same storage are serializable

(iii) No deadlocks can occur

A.2 Proof of Theorem [A.1.1]

Definition A.2.1. A task $t$ is write-linked with a task $t'$ on some memory address $addr$ if and only if there exist tasks $t_1 = t, ..., t_n = t'$, $n \geq 2$, and entries $l_{1,I}, l_{1,O} \in LVT_{t_1}, ..., l_{n,I}, l_{n,O} \in LVT_{t_n}$ such that:

(i) $l_{k,I}.baddr \leq addr \leq l_{k,I}.eaddr$, and $l_{k,I}.baddr \leq l_{k,O}.baddr$, for $1 \leq k \leq n$, and

(ii) $(l_{k,O}.iR, l_{k,O}.r, l_{k,O}.rs, l_{k,O}.read \land (l_{k,O}.i_A = l_{k,O}.i_R)) = (l_{k+1,I}.i_A, l_{k+1,I}.a, 1, 0)$, for $1 \leq k \leq n - 1$.

If task $t$ is write-linked with a task $t'$ on some memory address $addr$, we say that there exists a write-link on $addr$ between these tasks, and that tasks $t_1, ..., t_n$ and entries $l_{1,I}, l_{1,O}, ..., l_{n,I}, l_{n,O}$ that satisfy the above stated belong to the write-link. Furthermore, we say that task $t_i$ comes before task $t_j$ in the write-link if and only if $i < j$. Also, entries $l_{k,I}$ and $l_{k,O}$, for $1 \leq k \leq n$, are called input and output entries of the write-link, respectively.

Definition A.2.2. A task $t$ is read-linked with a reader task $t'$ on some memory address $addr$ if and only if

...
Appendix A. Correctness of Versioning

(i) \( \exists l_I \in LVT_t : l_I.baddr \leq \text{addr} \leq l_I.eaddr \text{ and } l_I.read = 0, \) and

(ii) \( \exists l_O \in LVT_{t'} : (l_O.i_A, l_O.a, l_O.i_R, l_O.rs, l_O.read) = (l_I.i_A, l_I.a, l_I.i_A, 1, 1) \)

If task \( t \) is read-linked with a task \( t' \) on some memory address \( \text{addr} \), we say that there exists a read-link on \( \text{addr} \) between these tasks. Also, entries \( l_I \) and \( l_O \) are called input and output entries of the read-link, respectively.

Proposition A.2.1. Let \( t \) denote some task, \( p \) a program point inside that task, and \( \text{addr} \) some memory address. The following claims hold:

1. If in the sequential execution there exists a chained preceding writer \( t_W \) of \( t \) on \( \text{addr} \) at \( p \), then, when \( p \) is reached in a parallel run, \( t \) is write-linked with \( t_W \) on \( \text{addr} \).

2. If in the sequential execution there exists a chained preceding reader \( t_R \) of \( t \) on \( \text{addr} \) at \( p \), and \( t \) writes to \( \text{addr} \), then, when \( p \) is reached in a parallel run, \( t \) is either read-linked with \( t_R \) on \( \text{addr} \), write-linked with \( t_R \) on \( \text{addr} \), or there exists a task \( t' \) such that \( t \) is write-linked with \( t' \) on \( \text{addr} \), and \( t' \) is read-linked with \( t_R \) on \( \text{addr} \). In the latter case, the output LVT entry of the read-link between \( t_R \) and \( t' \), and the input LVT entry of the write-link between \( t' \) and \( t \) are the same LVT entry.

3. All the tasks that are write-linked with \( t \) on \( \text{addr} \) when \( p \) is reached in a parallel run, are chained preceding tasks of \( t \) on \( \text{addr} \) at \( p \) in the sequential execution. If a task \( t' \) comes before a task \( t'' \) in some write-link, then in the sequential execution \( t' \) finishes before \( t'' \) finishes.

4. All the tasks that are read-linked with \( t \) on \( \text{addr} \) when \( p \) is reached in a parallel run, are chained preceding readers of \( t \) on \( \text{addr} \) at \( p \) in the sequential execution.

Proposition A.2.2. Let \( t \) denote some task. Then it holds:

1. There exists task \( t' \) such that \( t \) is write-linked with \( t' \) on \( \text{addr} \) if and only if \( \exists l \in LVT_t : l.baddr \leq \text{addr} \leq l.eaddr \land l.a \neq 0. \)

2. Let \( R_l \) denote the set of all entries that are the input entries of all read-links which exist between \( t \) and all the other tasks on \( \text{addr} \). Then, \( \sum_{l_i \in R_l} l_i.\Delta + l_i.\Delta = R \), where \( l_i \in LVT_t \) is the output entry of all read-links.

Proposition A.2.3. Let \( l \) and \( l' \) denote two LVT entries. Then it holds:

1. If \((l.rs, l.read \land (l.i_A = l.i_R)) = (1, 0)\), then \( l.r \neq 0. \)

2. If \((l'.i_R, l'.r) = (l.i_R, l.r) \) and \((l'.rs, l'.read \land (l'.i_A = l'.i_R)) = (l.rs, l.read \land (l.i_A = l.i_R)) = (1, 0)\), then \( l \) and \( l' \) are the same LVT entry.
3. If \((l'.i_R, l'.rs) = (l.i_A, 1)\), and \(l\) and \(l'\) belong to the same LVT, then \(l\) and \(l'\) are the same LVT entry.

4. If \((l'.i_R, l'.rs) = (l.i_A, 1)\), and \(l\) and \(l'\) belong to LVTs of two different tasks, then these tasks are chained on memory addresses \(addr : l.baddr \leq addr \leq l.eaddr\).

Proof. Propositions \[A.2.1\], \[A.2.2\] and \[A.2.3\] are proven by performing mathematical induction on the number of created tasks: assuming that a certain claim holds while there are up to \(n\) created task, and showing that it holds after \((n + 1)\)th task is created. The inductive steps are deduced in a straightforward manner, by analyzing the generate algorithm. Due to space constraints, we omit the proofs.

Corollary. Let \(l\) denote an LVT entry that has not been released such that \((l.rs, l.read \land (l.i_A = l.i_R)) = (1, 0)\). Then, \(GVT[l.i_R].wv\) has not been set to \(l.r\).

Proof. Proof is by contradiction. Assume that \(l\) is an LVT entry such that \((l.rs, l.read \land (l.i_A = l.i_R)) = (1, 0)\) has been released and that \(GVT[l.i_R].wv\) has already been set to \(l.r\).

Since the initial state of all write numbers is 0, and, by the first claim of Proposition \[A.2.3\], \(l.r \neq 0\), there must exist a released LVT entry \(l'\) such that \((l'.i_R, l'.r) = (l.i_R, l.r)\) and \((l'.rs, l'.read \land (l'.i_A = l'.i_R)) = (1, 0)\). This is, however, in contradiction with the second claim of Proposition \[A.2.3\].

Lemma \[A.2.4\]. Let \(t\) and \(t'\) be two tasks such that \(t\) is write-linked with \(t'\) on some memory address \(addr\). Then, if the output LVT entry from \(LV T_t\) of that write-link has not been released, \(t'\) cannot access \(addr\).

Proof. Let \(t\) denote a task that is write-linked with \(t'\), and let \(t_1 = t, \ldots, t_n = t'\) and \(l_{1,I}, l_{1,O} \in LV T_{t_1}, \ldots, l_{n,I}, l_{n,O} \in LV T_{t_n}\) be tasks and entries from that write-link. Assume that the output LVT entry from \(LV T_t\) of that link \((l_{1,O})\) has not been released.

Since \((l_{1,O}.i_R, l_{1,O}.r, l_{1,O}.rs, l_{1,O}.read \land (l_{1,O}.i_A = l_{1,O}.i_R)) = (l_{2,I}.i_A, l_{2,I}.a, 1, 0)\) and \(l_{1,O}\) has not been released, by the corollary to Proposition \[A.2.3\], it follows that \(GVT[l_{2,I}.i_A].wv\) was not set to \(l_{2,I}.a\). This further implies that \(GVT[l_{3,I}.i_A].wv\)
Appendix A. Correctness of Versioning

was not set to \(l_{3,I.a}\), as follows. Suppose the opposite, i.e., that at some moment \(\text{GVT}[l_{3,I.A}.wv] = l_{3,I.a}\). Since \((l_{2,O.i_R}, l_{2,O.r}, l_{2,O.rs}, l_{2,O.read} \land (l_{2,O.i_A} = l_{2,O.i_R})) = (l_{3,I.i_A}, l_{3,I.a}, 1, 0)\), by the corollary to Proposition A.2.3, it follows that \(l_{2,O}\) has been released. Consequently, since \(l_{2,I}.baddr \leq l_{2,O}.baddr\) and entries are released in the ascending order of \(baddr\) element, \(l_{2,I}\) has been released as well. However, this is in contradiction with the fact that \(\text{GVT}[l_{3,I.A}.wv] = l_{3,I.a}\). In similar manner, it can be shown that \(\text{GVT}[l_{4,I.A}.wv] = l_{4,I.a}\), and so on. To conclude, \(\text{GVT}[l_{n,I.A}.wv] = l_{n,I.a}\) and by the acquire algorithm the task \(t'\) cannot access \(addr\).

Lemma A.2.5. A task can read a memory address \(addr\) at some program point \(p\) only if all the chained preceding writers of \(t\) on \(addr\) at \(p\) have finished.

Proof. Proof is by contraposition. Assume that a task \(t\) tries to read \(addr\) at \(p\), and a writer preceding task \(t'\) has not yet finished. By Proposition A.2.1, \(t\) is write-linked with \(t'\) on \(addr\). Since \(t'\) has not finished, the output LVT entry from \(LVT_{t'}\) of that write-link has not been released. Consequently, by Lemma A.2.4 \(t\) cannot read \(addr\).

Lemma A.2.6. Let \(t\) denote some task that is accessing some address \(addr\) at a program point \(p\) and all the entries from LVTs of all the chained preceding tasks of \(t\) on \(addr\) at \(p\) have been released. Also, assume that \(\exists l \in LVT_t : l.baddr \leq addr \leq l.eaddr\). Then, if at some previous point during the execution \(\text{GVT}[l.i_A].wv = l.a\), it holds that \(\text{GVT}[l.i_A].wv\) has not been changed since then.

Proof. Proof is by contradiction. Assume that at some point it was true \(\text{GVT}[l.i_A].wv = l.a\), and that it is not true when \(t\) tries to access \(addr\). Consequently, when \(t\) tries to access \(addr\), there exists a released LVT entry \(l'\) such that \((l'.i_R, l'.rs, l'.read \land (l'.i_A = l'.i_R)) = (l.i_A, 1, 0)\) and \(l'_{r} \neq l.a\). By the third and fourth claims of Proposition A.2.3 it follows that there exists a task \(t' \neq t\) such that \(l' \in LVT_{t'}\) and \(t'\) is chained with \(t\) on \(addr\). There are two possible cases:
(i) In the sequential execution, \( t' \) finishes after \( p \).

In this case, it also holds that \( t' \) finishes after \( t \) finishes, since otherwise, in the parallel execution, \( t' \) would not be created before \( p \), and consequently \( l' \) would not have been released. Now, however, when \( l' \) was being released, \( t \) was a chained preceding task of \( t' \) on \( addr \), which is in contradiction with Lemma A.2.7.

(ii) In the sequential execution \( t' \) finishes before \( p \).

In this case \( t' \) is a chained preceding task of \( t \) on \( addr \) at \( p \). Moreover, by Proposition A.2.1, it follows that \( t \) is write-linked with \( t' \) on \( addr \). Since \( l'.r \neq l.a \), there must exists a task \( t'' \) that comes after \( t' \) in the write-link and \( \exists l'' \in \text{LVT}_p : (l''.i_R,l''.r,l''.r.s,l''.r.read \wedge (l''.i_A = l''.i_R)) = (l.i_A,l.a,1,0) \).

Proposition A.2.1 implies that (i) \( t'' \) is a chained preceding task of \( t \) on \( addr \) at \( p \), and (ii) in the sequential execution \( t' \) finishes before \( t'' \). Consequently, \( l'' \) has been released. However, when \( l'' \) was being released, \( t' \) was a chained preceding task of \( t'' \) on \( addr \), and by Lemma A.2.7, \( t' \) must have finished. Hence, \( GVT[l.i_A].wv \) was already set to \( l.a \) which is in contradiction with Lemma A.2.

\[ \square \]

**Lemma A.2.7.** A task can write a memory address \( addr \) at some program point \( p \) only if all the chained preceding tasks of \( t \) on \( addr \) at \( p \) have finished.

*Proof. *Proof is by contraposition. Assume that a task \( t \) tries to write \( addr \) at \( p \), and there exists a chained preceding task \( t_R \) that has not yet finished. By Lemma A.2.5 if \( t_R \) is a writer then \( t \) cannot read, and consequently write \( addr \). Hence, assume that \( t_R \) is a reader and all the other preceding tasks of \( t \) on \( addr \) at \( p \) have finished. By Proposition A.2.1, one of the three following cases can occur:

(i) \( t \) is read-linked with \( t_R \) on \( addr \).
By the definition of read-link, \( \exists l_O \in LVT_t : l_O.baddr \leq addr \leq l_O.eaddr \). Now, if \( l_O.a \neq 0 \), Propositions A.2.2.2 and A.2.1.3, imply that at some previous point during the execution \( GVT[l.i_A].wv \) was equal to \( l.a \) and \( GVT[l.i_A].rv \) was equal to 0. The same holds in the case of if \( l_O.a = 0 \), since \( l_O \) the initial value of both components of all version number are 0.

Since \( t_R \) have not finished, Lemma A.2.6 implies that \( GVT[l.i_A].wv \) have not change since it was set to \( l.a \). By the construction of the release algorithm, the \( GVT[l.i_A].rv \) could have only be increased by releasing entries that are the input entries of all read-links which exists between \( t \) and all the other tasks on \( addr \) at \( p \). Now, since, by Proposition A.2.2.2 the sum of those entries is \( R - l_O.\Delta \), and \( LVT_{t_R} \) contains such an entry, it follows \( GVT[l.i_A].rv < R - l_O.\Delta \). Hence, by the construction of the acquire algorithm, \( t \) cannot read \( addr \) at \( p \).

(ii) \( t \) is write-linked with \( t_R \) on \( addr \)

Since \( t_R \) has not finished, the output LVT entry from \( LVT_{t_R} \) of that write-link has not been released. Consequently, by Lemma A.2.1 t cannot read \( addr \).

(iii) There exists a task \( t' \) such that \( t \) is write-linked with \( t' \) and \( t' \) is read-linked with \( t_R \). The input LVT entry of the read-link between \( t_R \) and \( t \), and the output LVT entry of the write-link between \( t \) and \( t \) are the same LVT entry.

Similarly as in the case (i), the fact that \( t' \) have not finished, implies that \( t_R \) cannot read \( addr \), and consequently, the input LVT entry of the read-link between \( t_R \) and \( t \) has not been released. Since the same entry is also the output LVT entry of the write-link between \( t \) and \( t \), by Lemma A.2.1 t cannot access \( addr \).

\[ \square \]

Lemma A.2.8. A task can read a memory address \( addr \) at some program point \( p \) if all the entries from \( LVTs \) of all the chained preceding tasks of \( t \) on \( addr \) at \( p \) have been released.
Proof. Assume that a task \( t \) tries to read \( addr \) at \( p \). One of the three following cases can occur:

(i) \( \exists l \in LVT_t : l.baddr \leq addr \leq l.eaddr \)

In this case, by the construction of the acquire algorithm, \( t \) can access \( addr \).

(ii) \( \exists l \in LVT_t : l.baddr \leq addr \leq l.eaddr \) and \( l.a = 0 \)

In this case, since the initial value of each write version number is 0, by Lemma \[A.2.6\] \( GVT[l.i].wv = 0 \). Therefore, by the construction of the acquire algorithm, \( t \) can access \( addr \).

(iii) \( \exists l \in LVT_t : l.baddr \leq addr \leq l.eaddr \) and \( l.a \neq 0 \)

In this case, by Proposition \[A.2.2\] \( l \), there exists a task \( t' \) such that \( t \) is write-linked with \( t' \) on \( addr \). Further, by Proposition \[A.2.1\] \( 3 \), all the tasks in the link, except \( t \), are chained preceding tasks of \( t \) on \( addr \) at \( p \). Therefore, if all the chained preceding tasks of \( t \) on \( addr \) at \( p \) have finished, it follows that at some previous point during the execution \( GVT[l.i_A].wv \) was equal to \( l.a \). This, by Lemma \[A.2.6\] implies that it holds \( GVT[l.i_A].wv = l.a \), and therefore, by the construction of the acquire algorithm, \( t \) can access \( addr \).

\[\Box\]

Lemma \[A.2.9\]. A task can write a memory address \( addr \) at some program point \( p \) if all the entries from \( LVTs \) of all the chained preceding tasks of \( t \) on \( addr \) at \( p \) have been released.

Proof. It is already shown that if all the chained preceding tasks of some task \( t \) on \( addr \) at \( p \) have finished, it either holds that \( LVT_t \) does not store an entry for \( addr \), or for an entry \( l \in LVT_t : l.baddr \leq addr \leq l.eaddr \) holds \( GVT[l.i_A].wv = l.a \). What is left to show is, that in the latter case, it also holds \( GVT[l.i_A].rv = R - l.\Delta \).
Assume that $\exists l \in LVT_t : l.baddr \leq addr \leq l.eaddr$ and that all the chained preceding tasks of $t$ on $addr$ at $p$ have finished. This, by Proposition A.2.4, implies that the release algorithm was executed for all the tasks that are read-linked with $l$, and consequently, by Proposition A.2.2, $GVT[l.i_A].rv$ has been increased by $R - l.\Delta$.

Now, due to the construction of the release algorithm, when LVT entries that are the input entries of above mentioned read-links have been released, it must have hold $GVT[l.i_A].wv = l.a$. Consequently, there must have been a point in the execution where $GVT[l.i_A].wv = l.a$ and $GVT[l.i_A].rv = 0$. By Lemma A.2.6 it follows that $GVT[l.i_A].wv$ was not modified nor, consequently, have $GVT[l.i_A].rv$ was set to 0 since then and to a point when $t$ tries to access $addr$. Therefore, $GVT[l.i_A].rv = R - l.\Delta$.

The claim of Theorem A.1.1 follows directly from Lemmas A.2.5, A.2.7, A.2.8 and A.2.9.

A.3 Proof of Theorem A.1.2

Lemma A.3.1. Let $t$ and $t'$ denote two different tasks, $p$ a program point inside task $t$, and $o$ some object. Also, let $t_a$ denote the least common ancestor (the ancestor task with no child task that is also a common ancestor) of $t$ and $t'$. Then, if $t'$ is an exposed preceding task of $t$ on $o$ at $p$, all the task in the path between $t_a$ and $t$ in the task graph, except maybe $t_a$ make an exposed access to $o$.

Proof. Since $t'$ makes an exposed accesses to $o$, by the definition of exposed accesses, $o$ is created before $t'$ is invoked, in the serial execution. Further, since $t'$ finishes before $t$ is invoked, all the tasks in the path between $t_a$ and $t$, except $t_a$, access $o$ after $t'$ finishes. Consequently, all the tasks in the path between $t_a$ and $t$, except maybe $t_a$, make an exposed access to $o$.

We prove each claim of Theorem A.1.2 separately.
(i) It has been shown in Section 3.4 that if it is enforced that a task can write (read) an object \( o \) at program point \( p \) only if all the exposed preceding tasks (writers) of \( t \) on \( o \) at \( p \) have finished, dependences will be preserved. Hence, it suffices to show that, under versioning, a task can write (read) an object \( o \) at program point \( p \) only if all the exposed preceding tasks (writers) of \( t \) on \( o \) at \( p \). The proof is by induction on the length of the path (the number of nodes in the path, including start and terminal nodes) in the task graph between the exposed preceding tasks (writers) of \( t \) on \( o \) at \( p \) and the main task. In the remainder of the proof, we will use the term “significant length” to denote the described length.

The induction hypothesis is that, under versioning, a task can write (read) an object \( o \) at program point \( p \) only if all the exposed preceding tasks (writers) of \( t \) on \( o \) at \( p \) for which the significant length is less than \( n \), \( n \geq 2 \), have finished. In the base case \( (n = 2) \), Lemma A.3.1 implies that the only task, if any, in the path between \( t' \) and \( t \) that is not an exposed preceding task of \( t \) on \( o \) at \( p \) is the main task. Consequently, \( t' \) is a chained preceding task (writer) of \( t \) on \( addr \) at \( p \), and hence, by Theorem A.1.1(i), it is enforced that \( t' \) finishes before \( p \).

For the inductive step, assume that there exists an exposed preceding task (writer) \( t' \) of \( t \) on \( o \) at \( p \) for which the the significant distance is \( (n + 1) \). If \( t \) and \( t' \) are chained on some address \( addr \) used to store \( o \) the proof is done. Otherwise, by Lemma A.3.1 there exists a task that is an ancestor of \( t' \) that does not make an exposed access to \( o \). Let \( t'' \) denote a task with those characteristics that is “closest” to \( t' \) in the task graph, i.e., \( t_1 \) is either the parent of \( t' \) or all the task between \( t_1 \) and \( t' \) makes an exposed access to \( o \).

By the definition of exposed accesses, it follows that \( o \) is created inside \( t_1 \) (by \( t_1 \) or by descendant of \( t_1 \)). Further, since \( t \) accesses \( o \), in the sequential execution the
address of $o$ must be written inside $t'$ to some object $o_1$ that is (i) created before $t_1$, and (ii) read after $t_1$ finishes but before $p$. Let $t_2$ denotes the task that reads $o_1$ at some program point $p_1$, after $t_1$ finishes but before $p$ in the sequential execution.

It follows that $t_1$ is an exposed preceding writer of $t_2$ on $o_1$ at $p_1$.

Since $t_1$ is an ancestor of $t'$, the significant distance of $t_1$ is less than the significant distance of $t'$, and consequently less than $(n + 1)$. Therefore, the induction hypothesis can be applied on $t_2$ and its exposed preceding writer $t_1$. This implies that when $t_2$ reads $o_1$ at $p_1$ in a parallel run, $t_1$ must have finished. Consequently, since $t_1$ does not make an exposed access to $o$ but at least one of its child tasks does, $t'$ must have finished before $p_1$ is executed as well.

Now, if $t_2 = t$, the proof is done. Otherwise, there exists a sequence of objects $o_1, ..., o_n$ such that the address of $o$ is written to $o_1$, the address of $o_1$ is then written to some object $o_2$, ..., the address of $o_{n-1}$ is written to some object $o_n$, and then finally $o_n$ is read by task $t$ before $p$. However, by extending the induction to $n$ as well, it is easy to see that the claim holds in this case as well.

(ii) Two different objects can share the same storage only if they are both automatically (stored on the stack) or both dynamically (stored in the heap) allocated. The case of heap allocated objects was previously discussed in Section 4.2.3.

Assume that there exist group of tasks $T$ and $T'$ that access two different stack variables $o$ and $o'$, allocated in the stack of some tasks $t_1$ and $t'_1$, respectively. We consider two cases:

(a) $t_1 = t'_1$

Without loss of generality, let in the sequential execution tasks $T'$ finish before $T$. Since $o$ and $o'$ are created by $t_1$ and $t_1$ is the least common ancestor task of any two tasks from $T$ and $T'$, it follows that all the tasks from $T'$ and $T$ are chained on memory addresses used to store the stack variables being
considered. Consequently, when a task from group $T$ writes (reads) $o$, it will be ensured that all the tasks (writers) from $T'$ have finished.

(b) $t_1 \neq t'_1$

Since tasks do not share stack, in any parallel run, the execution of $t_1$ and $t'_1$ will not be overlapped. Without loss of generality, assume that in the parallel execution being considered $t_1$ finishes before $t'_1$. Since tasks $T'$ access $o'$ in an exposed manner, and $t_1$ does not, by the construction of generate and release algorithm, it is ensured that the release algorithm for task $t_1$ can finish only after tasks $T'$ have finished. Consequently, the accesses of tasks $T$ and $T'$ to $o$ and $o'$ do not overlap.

(iii) Deadlock freedom is a direct consequence of Theorem A.1.1(ii), since there always exists a task that can make further progress: the sequentially youngest task that has not finished.

$\square$
Appendix B

Bloom filters

A Bloom filter \(^{10}\) is a space-efficient data structure for representing an approximation of a set in order to support membership queries. For the given input set \(S\), the Bloom filter efficiently represents its superset \(S'\), and then answers the question is some element \(x\) a member of the set \(S'\). Clearly, \(x \notin S'\) implies that \(x \notin S\). On the other hand, if \(x \in S'\), \(x\) may or may not be in \(S\). In other words, the Bloom filter has no false negatives but allows false positives. The applicability of a Bloom filter lies in the fact that the false positive rate can be made arbitrary small by varying architectural attributes of this structure. Additionally, if implemented in hardware membership queries can be done very fast.

Architecturally, a Bloom filter consists of a bit array of \(m\) bits and \(k\) hash functions. Each hash function maps an element of the set to one of the \(m\) array positions. Adding an element to the input set is done by using hash functions to attain \(k\) array positions and then by setting these to 1. To test if an element is in the set, corresponding \(k\) array positions are inspected, and the answer is positive if and only if all positions hold the value 1. When implemented in hardware, \(k\) lookups can be done in parallel, so time needed to make a decision is the sum of the time needed to calculate one hash value and to do one (direct) lookup. Assuming that hash functions are independent and uniform
the false positive rate can be approximated by \( p_f \approx (1 - e^{-\frac{kn}{m}})^k \), where \( n \) represents the size of the input set \( S \).

To support deletion, modifications to the original Bloom filter model are needed. Fan et al. [30] introduced the idea of a Counting Bloom filter, where \( m \) bit array is replaced with an array of \( m \) counters which are then increased and decreased on insertion and deletion, respectively.

### B.1 Hardware implementation

In the theoretic model of counting Bloom filter, each hash function corresponds to one random lookup in the counter array. For \( k \) hash functions, in order to execute \( k \) lookups in parallel, a \( k \)-ported memory block is required. This puts an upper bound on the number of hash functions that can be used, if implemented in a straightforward way. For instance, the maximum number of ports for FPGA memory blocks which were at our disposal is two, which would limit the number of hash functions to two.

Fortunately, it has be showed in [29, 62] that this limitation can be alleviated without impacting the precision of the filter. Instead of using one \( k \)-ported memory block for storing \( m \) counters, \( \frac{k}{k'} k' \)-ported memory blocks, each storing \( \frac{mk'}{k} \) counters can be used. These blocks are then accessed in parallel, and each block uses only \( k' \) hash functions. Although hash functions are no longer uniform, under the assumption that the number of hash functions is much smaller than the number of counters (i.e. \( k/m \ll 1 \)), this has negligible effect on the false positive rate. Similar designs were proposed in [20, 52].

Our implementation is based on this approach, and the design of one CBF memory block with the corresponding logic is shown in Figure B.1. Each CBF block uses one hash function and can carry out four operations: test if the address is in the set, add the address to the set, remove the address from the set, and reset all the counters to zero.

Test operation is executed in one cycle and it is implemented as follows. Controller
logic calculates the value of the hash function for the given address, which is used to read the corresponding counter value from the memory block. The counter bits are then OR-ed to check if the counter is greater than zero, i.e., if there is a 'hit' for this CBF block. The final result of test operation is obtained by AND-ing hit signals from all CBF blocks.

![Diagram of CBF Block](image)

Figure B.1: The architecture of a CBF Block. The symbols $a$ and $d$ denote the widths of address and data signals of the memory block holding the counters.

Add and remove operations require two cycles. In the first cycle the corresponding counter is read from the memory block. If the counter has reached its maximum value (overflow signal in the figure) the counter value must remain the same. Otherwise, the value is either incremented or decremented and written back to the memory block in the second cycle.

On reset operations, all the counters in the memory block are sequentially traversed and their value is set to zero. Duration of the reset operation is therefore equal to the number of counters stored in the memory block.
Appendix C

Hardware implementation of the LVT

We have implemented LVT as a hardware linked list. In this singly-linked list, each node contains one LVT entry and a pointer to its successive node. The nodes are ordered by the start address (l.baddr field) of the region their entry corresponds to.

Architecturally, the list consists of a memory block used to store nodes and a controller which decodes inputs and updates the state of the memory block correspondingly. The address width of the memory block determines the maximum length of the list: assuming that the address width of the used memory block is $a$, the maximal length of the list is $2^a - 1$ (the maximum address value is used to denote the NULL value of the pointer).

In addition to the storage area needed for storing nodes, the controller maintains several additional registers, such as registers needed by the finite state machines and current and previous registers. The current register stores the last returned node of the list, while previous holds the predecessor of the current node. Storing this information in the controller makes traversing and updating of the list faster and simpler.

Operations that are supported by this structure are shown in Table C.1 together with the number of cycles required to execute them. The implemented structure is
Appendix C. Hardware implementation of the LVT

capable of executing the common functions of the linked-list data structure. Additionally, operations specific to versioning (search by address and three fragmentation operations) are also supported. Deletion from list are handled as follows. Once the node is deleted from the list, the corresponding location in the memory block cannot be reused until the list is reset. While it is possible to implement a memory recycle mechanism to reuse the memory space where the data have been deleted\footnote{The only deletes that occur outside the release algorithm are the ones done in the recovery phase of the generate algorithm.} for simplicity purposes and due to the fact that the majority\footnote{The only deletes that occur outside the release algorithm are the ones done in the recovery phase of the generate algorithm.} of deletes happens at the end of the list lifetime we opted not to.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>1</td>
<td>Removes all entries from the list</td>
</tr>
<tr>
<td>get first</td>
<td>1</td>
<td>Gets the first entry of the list</td>
</tr>
<tr>
<td>get next</td>
<td>1</td>
<td>Gets the successor of the current entry</td>
</tr>
<tr>
<td>search</td>
<td>$O(n)$</td>
<td>Searches for the first entry in the list such that end address element of the entry is less or equal than the input address</td>
</tr>
<tr>
<td>update</td>
<td>1</td>
<td>Replaces the current entry with the input entry</td>
</tr>
<tr>
<td>insert</td>
<td>2</td>
<td>Inserts the input entry before the current entry</td>
</tr>
<tr>
<td>append</td>
<td>2</td>
<td>Appends the input entry after the current entry</td>
</tr>
<tr>
<td>delete</td>
<td>1</td>
<td>Deletes the current entry</td>
</tr>
<tr>
<td>fragment$_2$</td>
<td>2</td>
<td>Fragments the current entry into two parts</td>
</tr>
<tr>
<td>fragment$_3$</td>
<td>2</td>
<td>Fragments the current entry into three parts</td>
</tr>
</tbody>
</table>

Table C.1: Operations of the LVT
Bibliography


BIBLIOGRAPHY


