Software Management of Hardware Memory Versioning

Tehila Mayzels
Software Management of Hardware Memory Versioning

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Tehila Mayzels

Submitted to the Senate
of the Technion — Israel Institute of Technology
Adar 5777 Haifa March 2017
This research was carried out under the supervision of Prof. Yoav Etsion, in the Faculty of Computer Science.

Acknowledgements

I would first like to express my sincere gratitude and appreciation to my advisor, Prof. Yoav Etsion, for his support, guidance, patience and encouragement. The door to Prof. Etsion’s office was always open whenever I needed help or advice. I could not have completed this work without his guidance. Yoav is not only a talented advisor, but also a friend who is always a pleasure to talk to.

I wish to thank my friends and colleagues in the Technion, who made this period more enjoyable, and helped a lot with good advice and discussions. In particular, I would like to thank Eran Gilad and Michal Horovitz.

Last, but certainly not least, I would like to thank my family:

- To my dearest parents, Naomi and Zevik Glidai, for their assistance and for continuous moral support throughout this period and much more.
- To my parents in law, Rina and Haim Mayzels, for their support and encouragement.
- To my beloved children, Taeer, Eliya, Eitam, Yifah, Nehorah and Hallel, who light up my days and teach me everyday what is really important in life.
- To my dear husband, Yinon, who has always believed in me, even when I did not believe in myself. I appreciate the patience, the technical assistance and the full support in my choices and dreams.

The Technion’s funding of this research is hereby acknowledged.
# Contents

List of Figures

List of Algorithms and Sample Codes

Abstract 1

Abbreviations and Notations 3

1 Introduction 5

2 Background and Related Work 7
   2.1 Out-of-Order Execution ................................. 7
   2.2 Task-based Programming ................................. 7
   2.3 Automatic Task Partitioning ............................. 8
   2.4 Task Dependencies ...................................... 10

3 The Versioned Memory Execution Model 11
   3.1 Memory Versioning and Renaming ......................... 11
      3.1.1 Task Dependencies ................................. 12
      3.1.2 Out-of-Order Execution of Read-Only Operations .......... 12
      3.1.3 O-structure API ................................... 13
      3.1.4 Advantages of versioned memory ..................... 14
   3.2 Unipath Data Structure Algorithms and Unipath Property ........ 14
   3.3 Parallelize Unipath Data Operations Using O-Structures ........ 14

4 Automatic Dependency Management 17
   4.1 Compiler Transformation Passes .......................... 17
   4.2 Programmer Annotations ................................ 18
   4.3 Task Handling ........................................... 19
   4.4 Version Management ..................................... 19
      4.4.1 Reader Task ....................................... 20
      4.4.2 Writer Task ....................................... 22
      4.4.3 Control Flow Graph ................................. 25
      4.4.4 Type Safety ....................................... 27
List of Figures

3.1 Binary Search Tree Using Versioning Memory ........................................... 16
4.1 Pipelined traversal on a linked list, during three unipath operations .......... 20
4.2 Adding a Basic Block to the CFG ............................................................... 26
4.3 Adding a $\phi$ node ......................................................................................... 27
6.1 Benchmarks: Data Structures ................................................................. 39
List of Algorithms

3.1 Out-of-order example ........................................ 12
4.1 Look a needle value up in a list ............................. 21
4.2 Look a needle value up in a list, with versioned memory 22
4.3 Insert a \texttt{val} node into the back of a list .......... 23
4.4 Insert a \texttt{val} node into the back of a list, with versioned memory ................................. 24
4.5 Process annotations and process O-structure variables ....... 28
4.6 Create tasks ..................................................... 29
4.7 Process unipath tasks .......................................... 29
4.8 Handle Writer Task: Fix CFG, Add $\phi$ Nodes and Missing Releases ... 30
Abstract

Task-based programming is an emerging paradigm that simplifies parallel programming. The idea is to partition the code into logical operations called tasks, when one or more independent tasks running concurrently, as threads, out of program order. Task-based models, however, mostly focus on expressing concurrency and, for the most part, do not reason about data synchronization. Memory dependencies among tasks must be taken into consideration in order to get correct results, but this may be difficult or impossible to decide before run time.

The recently proposed O-structures memory versioning model is intended to fill this gap and dynamically track data dependencies. The model extends the concept of register renaming to the entire memory and thereby eliminates false- and anti-dependencies between tasks. Using this model, each task views the same memory snapshot it would have viewed if executed sequentially. However, it only provides low-level primitives, which makes programming more complex and requires deep understanding of the system.

In this research, we extend the LLVM compiler to support O-structures and to manage the hardware memory versioning with minimal programmer work. Our compiler uses simple programmer annotations to identify which memory elements should be tracked by the versioning hardware. It then replaces those memory elements with special O-structures elements and replaces regular memory accesses with special versioned memory accesses. In order to maximize parallelism, it also takes care to efficiently acquire and release those O-structures, using the correct versions.

Our work focuses on handling a class of unipath algorithms on dynamic irregular data structures. In this class of algorithms, there is exactly one path from the structure’s root to each of its elements. The potential of parallelization here is high, but since the data structure is pointer based, it is difficult or impossible to specify and handle dependencies. With versioning memory model, pipelining of tasks is achieved by hand-over-hand synchronization. Our compiler enables programs that use unipath data structure algorithms to run successfully with versioned memory model, using only a few annotations. Hence, reaches maximal parallelism with minimal programmer intervention.
## Abbreviations and Notations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>TLP</td>
<td>Task Level Parallelism</td>
</tr>
<tr>
<td>TLS</td>
<td>Thread Level Speculation</td>
</tr>
<tr>
<td>OoO</td>
<td>Out of Order</td>
</tr>
<tr>
<td>OoOE</td>
<td>Out of Order Execution</td>
</tr>
<tr>
<td>RaW</td>
<td>Read after Write</td>
</tr>
<tr>
<td>WaR</td>
<td>Write after Read</td>
</tr>
<tr>
<td>WaW</td>
<td>Write after Write</td>
</tr>
<tr>
<td>O-structure</td>
<td>ordered and versioned memory element</td>
</tr>
<tr>
<td>$\theta$</td>
<td>a future, infinite, version</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>DAG</td>
<td>Direct Acyclic Graph</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>BB</td>
<td>Basic Block</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>$\phi$</td>
<td>a PHI node in the CFG</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the last years, the further scaling of processor frequency has been limited by physical and technological constraints. As a result, performance can only scale through parallelism, by placing multiple execution cores in a processor. However, even though contemporary programming languages include support for parallel programming, writing a parallel code is still considered an artisan’s job.

In an effort to simplify parallel programming, special frameworks have been developed. Amongst them we can find: Cilk [8], Intel threading Building Blocks [25], CellSs [3] and OpenMP [20, 19]. However, data dependencies are sometimes impossible to decipher or specify, synchronization is complicated and parallelism possibilities are limited to particular patterns. Therefore, in practice, software developers still find it difficult to use these frameworks, whether they need to parallelize a serial legacy code or write a new parallel code.

Simplifying parallel programming (possibly compiler-assisted) is an ongoing effort. Nowadays, compilers can only parallelize data-independent loops, and even then, they are limited by the use of pointers because of memory aliasing. The root cause of the problem is that the compiler cannot deduce the specific addresses stored in individual variables and, therefore, must conservatively assume that the variables are aliases of the same memory location.

Software parallelization requires correct identification of data dependencies between individual memory accesses. These are classified into three types: Read-after-Write (RaW or true dependencies), Write-after-Read (WaR or anti dependencies) and Write-after-Write (WaW or false dependencies). The first type is a simple producer-consumer dependency, in which an operation consuming (reading) data must wait for the data to be produced (written). The other two dependency types emanate from the reuse of memory locations across different instructions (or invocations thereof).

Prevalent Out-of-Order (OoO) processors employ this dependence classification to execute instructions out of program order. While program semantics require adhering to true dependencies, they extract parallelism by eliminating WaR and WaW dependencies using a technique known as register renaming. Since these dependencies are artifacts
of the processor’s finite resources (number of registers in the case of OoO processors),
this technique uses an interim, reusable buffer and creates a new version of a register
whenever it is written to. As a result, each register read can be mapped to the correct
version of the register data.

In this research, we explored the impact of extending the register renaming concept
to the entire memory on task-based program parallelization. Specifically, since task-
based programming models decompose sequential programs into potentially parallel
tasks, a versioning memory system eliminates false- and anti-dependencies between
tasks at run time and facilitates higher degrees of task parallelism. The ultimate goal
of this research was to explore the impact of a versioning memory system on compiler
transformations and compiler-assisted parallelization, while the aspiration is to create
as many parallel parts as possible.

Our compiler enhances a versioning memory system, based on research by Gilad
et al. [9], by adding the capability of using it on a class of serial programs. Using a
few necessary clues from the programmer, it partitions the original sequential code
into tasks and deals with data dependencies by utilizing versioned memory elements.
The compiler takes care of all necessary transformations and code additions; Thus, the
programmer does not need to take care of it or understand the versioning memory
system in detail.

The research is focused on a common class of algorithms, called *unipath algorithms*,
which deal with irregular data structures, such as linked-lists and binary trees. These
algorithms have high parallelization potential using hand-over-hand synchronization, but
are challenging because of the widespread use of pointers in such data structures. Memory
layout is dynamically modified, making it impossible to know the data dependencies at
static time. Our versioning memory system is very useful in this case, since it does not
require full alias analysis at compile time and also does not require speculative execution.
The key point of our work is that each task in the original serial program sees a different
snapshot of the memory. Therefore, we can say that each task has a different version
number, which it uses in all its versioned memory accesses. As a result, even when the
program runs concurrently and OoO, the sequential semantics are guaranteed. We will
show the applicability of the whole system in this case, and the compiler contribution
as part of it in particular, as a software that manages memory versions and uses them
correctly and intelligently.
Chapter 2

Background and Related Work

2.1 Out-of-Order Execution

Parallel computing has become a dominant paradigm in computer architecture and programming. There are several granularities of parallelism; one of the most basic is instruction parallelism, i.e., the concurrent execution of a few instructions during a single clock cycle. Superscalar architecture [12] is a microprocessor design that enables this. Running several instructions during each cycle exploits instruction level parallelism (ILP) and minimizes the execution time of each thread. In a superscalar design, the processor, or the compiler, is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it is dependent on another instruction and must be executed in sequence with it. When data is not available, an instruction stalls, but OoOE processing prevents the stalled instruction from stalling the entire pipeline. Hence, other instructions can still execute. However, ILP has a major drawback, which is the control dependencies; branch instructions limit the instruction window size. An instruction that is dependent on a branch cannot be carried out before the branch, and an instruction that is control-dependent on a branch cannot be carried out after the branch. Task-based programming is aimed at uncovering parallelism at scales that an OoO processor cannot reach.

2.2 Task-based Programming

Task level parallelism (TLP) provides parallelism with much larger granularity compared to ILP. The idea is to take a sequential program, partition it into logic portions of the code or the control flow graph (CFG) and execute them on several processing units concurrently. These portions are called tasks. ILP parallelizes instructions within a task, while TLP parallelizes across tasks. Various programming language-based frameworks exist for this purpose. The programmer’s mission is to develop code that is made up of tasks, which are, as described before, code regions that can be executed in parallel. In this way, he can utilize available processing cores more efficiently. Examples of these
programming models include Cilk[8], CellSs[3], Intel Threading Building Blocks[25] and more [23, 2].

Cilk[8] is a C-based programming language that aims to run efficiently on contemporary symmetric multiprocessors (SMP’s). Intel Threading Building Blocks (Intel TBB)[25] is a C++ template library which provides data structures and algorithms to facilitate writing task-based parallel code. Both Cilk and Intel TBB leave the decision of where to start a new task in the code and how to synchronize between tasks to the programmer. Keywords like spawn (fork) and sync (join) were added to the languages, and the programmer can use them in the code in order to make it parallel. Inter-task dependencies should be resolved by the programmer (using sync, for example). CellSs[3] works differently. The programmer is required to annotate a task code and specify the input/output parameters. The CellSs run-time library uses these annotations to build a task dependency graph. The graph is then used to schedule independent tasks, thereby eliminating the need for user-guided synchronization points.

DAG-consistent distributed shard memory[4] is an example of a relaxed consistency model for distributed shared memory that respects the dependencies between the tasks defined in Cilk. The programmer can define an order between the tasks/threads using a direct acyclic graph (DAG) and this model ensures that any thread that reads a shared memory variable will get the latest written value of this variable.

OpenMP [20, 19] is another way to create a parallel program. The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Avoiding dependent tasks is the programmer’s responsibility.

Grappa [18] is a runtime system that enables running tasks concurrently, by adding calls to its API, and is very useful on big data. The programmer is encouraged to think of the system as a single large shared memory, use data structures that intelligently distribute themselves across the cores, and write parallel loops that are executed in parallel across the whole machine. Realm [33] is another runtime system for concurrent execution, where the programmer partitions a program into tasks and uses a graph to define the constraints: nodes represent operations to be performed and edges represent ordering constraints, such as control or data dependencies. Realm uses a light-weight event system to dynamically represent the program graph, then compresses the representation of many events into a single generational event, making the system much faster and more efficient.

### 2.3 Automatic Task Partitioning

OpenMP, Cilk, CellSs, Intel TBB, Grappa and other similar frameworks all share a basic drawback: they all require the programmer to write a parallel code and analyze dependencies. Unfortunately, writing efficient parallel code still requires substantial programming expertise. This is even more difficult in cases of rewriting a legacy serial
code. When the user does not use one of these frameworks and there is no ability to statically figure out the dependencies, speculative execution and dynamic scheduling may be used. The processor then uses multiple execution units to simultaneously carry out multiple independent instructions.

Many studies have proposed architectures that can exploit TLP, such as multiscalar architecture[28]. Multiscalar architecture [28] is a parallel architecture consisting of several processing units that can work concurrently. The partitioning of the program into tasks is done by the programmer or by the compiler, based on the CFG. Dependencies between different tasks are solved by the hardware and the software:

- Output/False (write-after-write (WaW))- and anti (write-after-read (WaR))-dependencies that result from a limited number of architectural registers are resolved using register renaming.

- Loads and stores are ordered using an Address Resolution Buffer (ARB), which stores multiple versions of a single location.

ARB [7] is one option for tracking the versions of the data, in order to maintain sequential semantics of the original program. The problem with ARB, as a centralized buffer, is poor latency and bandwidth. Speculative Versioning Cache[10] tries to overcome this problem by using an organization similar to that of snooping bus-based coherent cache. Multiscalar architecture relies heavily on speculative execution technique. This way, tasks that include loads can be executed without waiting for the previous task to complete storing. Speculative execution improves the utilization of the processor, but might also fail if incorrect values are read. When this happens, the reading task and all its successors are squashed and restarted, and any completed memory/register changes are reverted.

Data speculation technique is also commonly used in other architectures, like the Hydra chip multiprocessor [11] and other CMP systems[13]. Speculation execution is not used in this research. Instead, information of a tasks creation order is used by the hardware to optimize parallelism. DiscoPoP[16] is a tool for automatically finding potential parallelism in serial programs. Its strategy relies on profiled execution by automatically inserting instrumentation to track memory accesses. Then, at program runtime, profiling information is collected, and according to this information, the tool finds parallelization opportunities. POSH compiler [17] is a compiler for multi-core architectures with thread level speculation (TLS [28, 21, 1, 13, 29, 5, 26, 30]). POSH leverages the code structures created by the programmer, such as subroutines and loops, in order to partition the code into tasks. It also uses a profiling pass to discard ineffective tasks. The Mitosis compiler[24] partitions programs into speculative threads, based on pre-computation of thread input values. This work is based on speculative thread execution; therefore, it can relax parallelization constraints. When a speculative thread ends, the speculation is verified and, in case of mis-speculation, the thread’s work is discarded.
2.4 Task Dependencies

The main issue in partitioning a program into tasks is guaranteeing that there are no dependencies between tasks. Otherwise, task order should be provided to the compiler, to the runtime environment or to the hardware. This research uses a different memory model, presented by Gilad et al [9]. This is a model of versioned memory, which is a similar idea to the register renaming concept, but for memory locations. Memory versioning and renaming enables matching producers and consumers to ensure RAW dependency and eliminates WAR and WAW dependencies (by using multiple versions per memory location). The idea of memory versioning or data replication was shown in previous studies [6, 27], but those studies were limited to eliminating one false dependence at a time, while the number of memory places that could be versioned was limited. Chapter 3 elaborates on the model that have used. The compiler’s responsibility is to manage the tasks, so that each task will read from and write to the correct versions, for every versioned memory location. This should be done with minimal effort from the programmer, and this is the main contribution of this work.
Chapter 3

The Versioned Memory Execution Model

This section describes the execution model that our work is based on, expands and completes. The model is taken from the research by Gilad et al [9], where it is described in detail; this section will provide a general overview/description of the model. With this model, concurrent execution of sequential programs is possible thanks to special storage elements called $O$-structures.

3.1 Memory Versioning and Renaming

$O$-structures were so named because they provide an ordered access to multiple versions of a program state for the same memory address. $O$-structures are novel architectural memory elements that can be used to facilitate parallelism in task-based execution models. $O$-structures are designed to be coupled with task-based execution models, where a single imperative program thread is carved via software or hardware into multiple program tasks. Tasks are expected to be ordered, yet it is not expected that tasks are spawned in sequential order [26].

As in the notion of register renaming, each write to an $O$-structure creates a new version of program memory location. These versions can be accessed concurrently, and also out of original program order. $O$-structures provide a set of semantics that match the needs of task-based execution models, specifically allowing tasks to synchronize for specific versions of memory, as well as to coordinate access when the necessary version is not known at compile time. The memory system based on $O$-structures provides a consistency model we call statically ordered consistency. The model provides an ordering of memory operations when certain requirements are fulfilled by the program. It guarantees that, any task will observe the same state that its code would have viewed in the original sequential program.
3.1.1 Task Dependencies

Similarly to the notion of register renaming, using O-structures helps to get rid of false dependencies and anti-dependencies and to handle true dependency, so that statically ordered consistency is guaranteed:

- WaR dependency - The write must be to a version that is different (further) from the version of the memory location that is being read.

- WaW dependency - Same as WaR. The two writes use different versions.

- RaW - This is a true dependency. The versions will be the same, and the read cannot succeed until the write is completed.

3.1.2 Out-of-Order Execution of Read-Only Operations

The versioned memory execution model naturally allows the concurrent running of read-only operations, since these do not change the memory state. But the model provides an even stronger guarantee: it allows read-only operations to operate out of program order with respect to the surrounding mutating operations. Since O-structures maintain previous versions of the stored value, the read operation can read data (and pointer) values earlier in version order than a mutating operation can. The memory state that the operation sees is like a ”memory snapshot” which is identical to what it would have seen in the serial execution of the original program. For example, given the following sequence of operations on a sorted linked list:

```
Sample Program 3.1 Out-of-order example
1: linkedlist.insert(5)
2: linkedList.lookup(5)
3: linkedList.delete(5)
4: linkedList.lookup(5)
```

In this example, the mutating operations are the insert and delete on line 1 and line 3, respectively, and the lookup operation is a read only. According to program order, the first lookup (line 2) should succeed in finding the value 5, because it follows an insertion of value 5. The second lookup (line 4) should fail, since it follows a deletion of the value 5. By versioning the memory, it is completely possible for these operations to run out of order, in any order, and the result will be the same. Specifically, the first lookup (line 2) should issue accesses to the version that represents the memory state just after the insertion and before the deletion, where linkedList includes the value 5. Therefore, it should succeed. On the other hand, the second lookup should access the memory version after the deletion (line 2), where linkedList does not include the value 5, and it should fail. Further, if the first lookup happens to start running before the insertion, it will be blocked until the insertion gets to run and releases a root pointer.
The concurrent execution of read-only operations, which does not need to respect the original order of mutating operations, is one of the major advantages of the O-structures system.

### 3.1.3 O-structure API

The O-structure model includes an application program interface (API) that is used by the compiler to manage versioning. Before accessing an O-structure, acquiring the specific version is needed. Once the acquiring is done, accessing the version is allowed. Acquiring for write or for read-write locks it from accessing of other tasks and blocks them. Once accessing is completed, the version of the O-structure should be released so that other tasks that are waiting for it would also be able to acquire and access it.

Following are the signatures and explanations of the API functions:

- **Acquire the permission to write**: `void acquire_w(uint32_t* addr, uint32_t version)`
  
  Used for writes (exists only in mutating operations); locks the version `version` for write access and does not blocked by other acquires. Ends when the release of the same O-structure and version is completed.

- **Acquire the permission to read and write**: `uint32_t acquire_rw(uint32_t* addr, uint32_t version)`
  
  Used for writes and reads that should wait for a previous write; locks the version `version` for write/read access and is blocked until previous acquire releases its lock. Ends when the release of the same O-structure and version is completed.

- **Acquire the permission to read**: `uint32_t acquire_r(uint32_t* addr, uint32_t version)`
  
  Used for reads that are in immutating operations, but should wait for a previous write; does not lock the version `version` for write/read access. No release is needed.

- **Store**: `void store(uint32_t* addr, uint32_t version, int32_t value)`
  
  Writes a value `value` to a specific version `version` of an O-structure that stores versions of `addr`.

- **Load**: `int32_t load(uint32_t* addr, uint32_t version)`
  
  Reads from a specific version `version` of an O-structure that stores versions of `addr`.

- **Release**: `void release(uint32_t* addr, int32_t fromVer, int32_t toVer)`
  
  Releases the acquired permission for write access of `addr` memory, from `fromVer` to `toVer`. The specified `toVer` can be a specific version, indicating a range of versions from `fromVer` to `toVer` (inclusive), or a special version `theta` which indirectly specifies the next version written.
3.1.4 Advantages of versioned memory

There are several advantages to this model:

- Simplify parallel programming. No need to deal with atomicity or mutual exclusion.
- Non-speculative.
- Deterministic execution.
- Modular. No changes to processor pipeline, caches or coherence protocol.

3.2 Unipath Data Structure Algorithms and Unipath Property

Irregular data structures, such as linked lists, trees and hash tables are commonly used in programming; in particular, data structures with immutable data, meaning the value of an element of the structure cannot be changed. The data structure layout is mutable and dynamic. For operations on irregular data structures, that do not change the data structure layout, like looking up a value, serialization is not needed and parallelism can be exploited. On the other hand, mutating operations on such data structures, such as the insertion of a new node or deletion of a node, are challenging. Not only do they change the values stored in the node, they also modify the data structure layout, by changing pointers. Therefore, such operations should be serialized, so that the program is deterministic and will keep a statically ordered consistency. Serialization is required at the point of mutation only. Therefore, the order in which data structure operations traverse the same path in the structure must be serialized, and where these paths diverge - parallelism can be exploited. This is why we distinguish between unipath and multipath data structures, and emphasize the unipath property itself.

The access pattern that we call unipath is a traversal on an irregular data structure that always begins at the root, and for each node in the data structure, there is a single path to reach it. More generally, unipath property is an algorithmic characteristic of operations which always follow the same path in order to reach a certain element of a data structure.

3.3 Parallelize Unipath Data Operations Using O-Structures

Using O-structures allows operations on unipath data structures, like linked lists, trees and hash tables, to be executed in parallel, by pipelining subsequent operations. Given a single path to any node in the data structure, proper acquiring and releasing guarantees that mutations take place in sequential order.

Figure 3.1 illustrates how memory versioning can be used to reason about fine-grain synchronization in a concurrent, task-based execution of common operations on a binary
search tree. Figure 3.1(a) presents code that consists of a sequence of insert and lookup operations on a binary tree. The canonical task-based code loosely follows the OpenMP syntax — the insert and lookup functions are defined as task code, and each of their invocations creates a new task.

Figures 3.1(b) and 3.1(c) illustrate the memory versions and potential concurrency extracted from the code in Figure 3.1(a). When executing the code, insertions to the tree generate new versions of the tree, which are depicted in Figure 3.1(b), and a versioned memory system presents each running task a memory view defined by the virtual task timeline. Consequently, insert tasks $T_1$, $T_5$, and $T_7$ can be ordered with respect to one another. Each of these tasks will be blocked when trying to access memory versions that were not yet generated by their preceding tasks. Similarly, lookup tasks will be associated with the version of the data structure they would have viewed on a sequential execution. These tasks will be blocked until their memory versions have been generated by their preceding insert task.

One possible concurrent (dual processor) task schedule is illustrated in Figure 3.1(c) (the shape of each task visualizes the version of the binary tree it views). In this schedule (which is intentionally simplified and does not account for fine-grain aspects such as memory stalls and the memory-level parallelism they enable), insert task $T_1$ is first executed on CPU1. Next, lookup task $T_2$ and insert task $T_5$ on CPU1 and CPU2, respectively. Memory versioning enables reader task $T_2$ and mutator task $T_5$ to execute concurrently, since the latter generates a new version of the data structure that is not viewed by the former. Similarly, insert task $T_7$ (mutator) can execute concurrently with lookup task $T_3$ (reader).

Memory versioning also uncovers parallelism by eliminating false- and anti-dependences in memory, similar to the way register renaming eliminates dependences across registers in out-of-order processors [31, 22]. For example, lookup task $T_3$, which searches the tree for the value 8 is executed on CPU2 right after insert task $T_5$, which inserts the same value into the tree. Nevertheless, the lookup in $T_3$ will fail since $T_3$ precedes $T_5$ in the virtual timeline and therefore views a version of the binary tree that precedes the insertion of the value it seeks.
```
#pragma task
void insert(int v);

#pragma task
bool lookup(int v);

int main()
{
    insert(5); // task T1
    lookup(5);  // T2
    lookup(8); // T3
    lookup(3); // T4
    insert(8); // T5
    lookup(5); // T6
    insert(3); // T7
    lookup(5); // T8
}
```

(a) Binary search tree code example.

Figure 3.1: Binary Search Tree Using Versioning Memory
Chapter 4

Automatic Dependency Management

Compiler support is crucial to the success of O-structures because the system is not easy to program for. This chapter describes the compiler which lies at the heart of this work. This compiler automatically changes sequential code into concurrent code that uses O-structures in order to run correctly and deterministically. Our compiler uses a few annotations added by the programmer in order to know which code should be changed and where. The success criterion is functionality of the transformed code that should use the versioned memory model correctly.

4.1 Compiler Transformation Passes

Generally, compilation consists of three phases:

- Front end - Conversion from source code into intermediate representation (IR).
- Middle end - Transformations and optimizations that are not target-specific. This phase processing IR and transforms it.
- Back end - More IR transformation and optimizations that are target-specific, and conversion from IR into machine code (code generation).

Transformation and optimizations are implemented as passes. Our complier enhancements are implemented as transformation passes at the beginning of the back end phase, as they are target-specific (for versioning memory system only). It transforms the IR, which will be used for code generation afterwards. In the compiler we have expanded, LLVM [14] [15], there are five types of transformation passes:

1. ModulePass: general interprocedural pass
2. CallGraphSCCPass: bottom-up on the call graph
3. FunctionPass: processes a function at a time
4. LoopPass: executes on each loop independently

5. BasicBlockPass: processes one basic block at a time

Two additional compiler passes manage dependency issues using versioning memory system; the first is a ModulePass and the second is a FunctionPass:

1. Preliminary versioning memory pass - processes the whole module (program); it looks for annotations, saves important information, and changes memory elements that are annotated as ”should be versioned” into O-structures. Another important job of this pass is creating tasks.

2. UniPath operation processor pass - processes each function that became a task; it deals with different types of tasks in different ways, transforms accesses to O-structures, manages versions, fixes control flow graph when needed, and more.

4.2 Programmer Annotations

As mentioned above, the programmer should add annotations in order to let the compiler know what code should be changed. There are two types of annotations:

- Annotation for task/unipath operations:

  `unipath` - An annotation that is attached to a function signature. It tells the compiler that this function is a unipath function, which should become a task. In addition, the code should be updated to use versioned memory. For example: `bool lookup(uint32_t needle) __attribute__((annotate("unipath")))`

- Annotation for memory elements that should be versioned:

  `unipath_head` - An annotation attached to a variable that is the root of the unipath data structure, like a head of a linked list. It means that this variable should become a versioned memory and should be synchronized for every read or write. For example: `__attribute__((annotate("unipath_head"))) node_t* head = NULL;`

  `vermem` - An annotation attached to a variable that should become a versioned memory and should be synchronized for writes and overlapped reads and writes. For example: `__attribute__((annotate("vermem"))) node_t* next;`

In the code examples for linked list: look a value up 4.1 and value insertion 4.3 these annotations are presented. Theoretically, if enhanced static analyses, such as shape analysis, were used, we could get rid of the annotations, but this is outside the scope of this work.
4.3 Task Handling

Partitioning of the code into tasks is a difficult problem and it is not the main target of this research. Therefore, it is based on hints (as annotations) from the programmer. Since this research focuses on unipath algorithms, the unipath functions become tasks, carried out by the compiler. Each task performs a path traversal on the data structure, either a reader task or a writer (i.e., mutator) task. The compiler finds a call to each unipath function and replaces it with a call to \textit{CreateTask} function, sending a pointer to the unipath function as a parameter. \textit{CreateTask} will run the function as a new thread, in parallel. At the end of each unipath function, before the return instruction, the compiler adds a call to \textit{EndTask} function. \textit{EndTask} will end thread execution, and free the CPU for another thread.

4.4 Version Management

Managing O-structure versions is a major responsibility for the compiler. To achieve this goal, the compiler needs to perform the following operations:

- Replace the original memory elements, annotated by the programmer, with O-structures. Sometimes allocation is needed. In most cases a type matching is required. A special case is the data structure’s root node.
- Find all accesses to those memory elements and change/remove them.
- Add acquires, loads, stores and releases. The correct version of O-structure should be calculated and used.
- Identify the uses of those variables and update their accesses too.
- Fix the control flow graph, especially addressing existing \(\phi\) functions and missing \(\phi\) functions. \(\phi\) functions, or \(\phi\) nodes, are artifact variables, used internally by the compiler, when the definition of a variable has a few sources in the control flow graph (CFG).
- Verify that each acquired O-structure is released correctly.

Unipath traversals facilitate fine-grained serialization in a \textit{hand-over-hand} fashion. First, tasks are only allowed to acquire for access privilege to the O-structure containing the pointer to the root node, in program order. During unipath traversal, a task first acquires the O-structure containing a pointer to the next node and only then releases the O-structure holding the pointer to the current node (thus \textit{handing it over} to the next task). Consequently, tasks can only traverse a node path in program order and cannot bypass each other. This traversal guarantees the serialization of mutations that go through the same path, while exposing pipelined parallelism between tasks. Moreover, it does not limit mutating operations that traverse different paths. Figure 4.1 illustrates
this pipelined traversal over a linked-list data structure. The figure demonstrates three linked list unipath operations. Each unipath operation task first acquires an access permission to the O-structure that stores the pointer to the next node and then it releases the O-structure that stores the pointer to the node. The result is a pipelined traversal that maintains sequential semantics.

Figure 4.1: Pipelined traversal on a linked list, during three unipath operations

Every time the compiler needs to add a release, it should understand what version it is meant to release to, and call the correct API. There are four different cases here:

- Release to the next task, when the thread that runs the next task is spawned from the current thread (i.e., the next thread is the "child" of the current thread), and the current task releases the root node to that task. In this case the compiler calls $GetChildEpoch()$ function.

- Release to the next task, when the thread that runs the next task is spawned from the "parent" thread, and the current task releases the root node to it; i.e., the next thread is a "cousin" of the current thread. In this case the compiler calls $GetChildEpoch(GetNextEpoch(GetParentEpoch(currentEpoch)))$ functions.

- Release to all future tasks (infinite version), when we release a node that is not the root node. In this case the compiler calls $GetTheta()$ function. $\theta$ is a special version, which indirectly specifies the next version written.

- Release to current task - This is a special case. Before writing to the O-structure, a release to the current thread is needed, and right after it the write (store) can take place.

4.4.1 Reader Task

When the unipath operation does not mutate the data structure, the case is quite simple. As demonstrated in subsection 3.1.2, read only operations can run concurrently. In
order to avoid RaW and WaR, a synchronization of the head of the data structure is required. From the compiler’s point of view, it means that the load of the head (from the O-structure that stores its versions) is replaced with an acquire for read write permission, a load immediately after the acquire and a release, immediately after the load. The version of the head O-structure that is acquired for access permission is the version of the current task. The release is to the consecutive version (next task). Theoretically, the best release is to the next writer task, but in order to simplify its work, the compiler acts similarly to a writer task. Hence, the original serial order is kept by the order of the accesses to the head. All other loads are replaced with an acquire for read permission and a load (no release) which is a non-blocking acquire, in order to maximize parallelism. Lookup for a value in a data structure is an example of a reader (read-only) task.

**Example of a Reader Task**

Following is a detailed look at the unipath un-mutator operation of back insertion into a linked list. Consider having a linked-list with `head` as a pointer to the root node. Each node in the list consists of `data` field and `next` field that points to the next node. Here is an example of a possible code of loop a needle value up in a linked list, before the compiler transformation:

```c
Sample Program 4.1 Look a needle value up in a list
1:   _attribute_(_annotate("unipath_head")) node_t * head;
2: function Lookup(uint32_t needle) _attribute_(_annotate("unipath"))
3:    node_t* curr;
4:    uint32_t val;
5:    curr = head;
6:    while curr != NULL do
7:        val = curr → data;
8:        if val == needle then
9:           return true;
10:    end if
11:    curr = curr → next;
12: end while
13: return false;
14: end function
```

There are two variables here that are annotated as memory elements that should be versioned:

- `node_t *head` - root of the linked list.
- `curr → next` - the next field of current node. This annotation does not appear
here, since the next field is part of the structure node_t. Therefore that annotation will appear there.

The compiler will replace these variables with O-structures and update the accesses to them accordingly. The LOOKUP procedure, that is annotated as a unipath operation will become a task.

Next is the updated code after our compiler’s transformations:

Sample Program 4.2 Look a needle value up in a list, with versioned memory

1: Ostructure ostruct_head;/*holds versions of pointer to head. Has a special O-structure type*/
2: function LOOKUP(uint32_t needle)
3:    Ostructure ostruct_curr.next;/*ostruct_curr.next holds versions of a pointer to the next node. Has a special O-structure type*/
4:    node_t * curr;
5:    uint32_t val;
6:    unsigned int curr.version = GetCurEpoch();
7:    unsigned int to.version = GetNextEpoch(curr.version);
8:    acquire_rw(ostruct_head,curr.version);
9:    curr = load(ostruct_head,curr.version);
10:   release(ostruct_head,curr.version,to.version);
11:   while curr ≠ NULL do
12:      val = curr → data;
13:      if val == needle then
14:         return true;
15:      end if
16:      ostruct_curr.next = curr → next;
17:      acquire_r(ostruch_curr.next,curr.version);
18:      curr = load(ostruch_curr.next,curr.version);
19:   end while
20:   return false;
21: end function

4.4.2 Writer Task

Mutator operations are a different issue. Here, the synchronization is implemented for every access to the data structure. Each element should be accessed using a blocking acquire and released only after acquiring of the next element in the path: any load should be replaced with an acquire for read-write permission, a load immediately after the acquire, and a release after the next acquire. Stores are replaced with a release to current version, an acquire for write permission, a store and a release of current O-structure to the next writer. We assume that there is only one store here, and that it is the last memory access operation in the task, because after the mutation the task
completes its job. The version of the O-structures that are asked to be acquired for access permission is the version of current task. The release of the head (root) of the data structure is to the consecutive version (next task). For non-head O-structures the release is to \( \theta \) (infinite version, any future tasks). Hence, the original serial order is kept by the order of accesses to the head. After acquiring the next node in the path (in the structure), the previous node may be released. This way, ”hand-over-hand” locking is achieved. Hence, the order of original operations is saved and dependencies are resolved.

**Example of a Writer Task**

Next is a closer look at the unipath mutator operation of back insertion into a linked list. Here is an example of possible code before the compiler transformation:

```
Sample Program 4.3 Insert a val node into the back of a list
1: _attribute_(annotate("unipath_head")) node_t * head;
2: function Insert(uint32_t val) _attribute_(annotate("unipath"))
3:    node_t * curr, * newnode = new node_t(val);
4:    curr = head;
5:    if curr == NULL then
6:       head = newnode;
7:       return;
8:    end if
9:    curr = curr -> next;
10:   while curr != NULL do
11:      curr = curr -> next;
12:   end while
13:   curr = newnode;
14: end function
```

The variables that should be replaced with O-structures are the same as in the 4.1 example.

Following is the updated code after our compiler’s transformations:
Sample Program 4.4 Insert a `val` node into the back of a list, with versioned memory

1: Ostructure ostruc_head; /*holds versions of pointer to head. Has a special O-structure type*/

2: function Insert(uint32_t val)

3: ostructure ostruc_curr; /*ostruct_curr holds versions of a pointer to the current node. Has a special O-structure type*/

4: ostructure ostruc_curr_next; /*ostruct_curr_next holds versions of a pointer to the next node. Has a special O-structure type*/

5: node_t * curr, *newnode = new node_t(val);

6: unsigned int curr_version = GetCurEpoch();

7: unsigned int to_version = GetNextEpoch(curr_version);

8: acquire_rw(ostruc_head, curr_version);

9: curr = load(ostruc_head, curr_version)

10: if curr == NULL then

11: release(ostruc_head, curr_version, curr_version);

12: acquire_w(ostruc_head, curr_version);

13: store(ostruc_head, curr_version, newnode);

14: release(ostruc_head, curr_version, to_version);

15: return;

16: end if

17: ostruc_curr = ostruc_head;

18: ostruc_curr_next = curr -> next;

19: acquire_rw(ostruc_curr_next, curr_version);

20: release(ostruc_head, curr_version, to_version);

21: curr = load(ostruc_curr_next, curr_version);

22: to_version = GetTheta();

23: while curr ≠ NULL do

24: ostruc_curr_next = curr -> next;

25: acquire_rw(ostruc_curr_next, curr_version);

26: release(ostruc_curr, curr_version, to_version);

27: curr = load(ostruc_curr_next, curr_version);

28: end while

29: release(ostruc_curr_next, curr_version, curr_version);

30: acquire_w(ostruc_curr_next, curr_version);

31: store(ostruc_curr_next, curr_version, newnode);

32: release(ostruc_curr_next, curr_version, to_version);

33: end function

Following are some comments on the code:

• Although lines 18-21 seem redundant because they are similar to lines 24-27, they are in fact required. The only different between them is the value of `to_version`,
which is the next version in the former and \( \theta \) in the latter. The compiler does not write redundant code, but use a \( \phi \) function to select between the versions at the head of the while loop.

- On line 25 and line 26 the hand-over-hand is implemented. First, we acquire the next node and then we release the current one. Hence, the next task can acquire and access the current node, in parallel.

- On lines line 29 and line 30 we release the current acquire to the current version (therefore, no later task can reach here yet, and order is saved) and reacquire it for the insertion of the new node.

### 4.4.3 Control Flow Graph

During code transformations, some changes in the Control flow graph (CFG) may be needed as well. The CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. A node in the CFG is a basic block (BB) that contains instructions. Sometimes adding an instruction (acquire, release, etc.) is not straightforward. If we need to add an instruction between two basic blocks, the algorithm should create a new basic block, add the instruction into this basic block and update the edges in the CFG accordingly. In addition, using O-structure in new places usually leads to a need to add \( \phi \) functions that select the value used, when there are few predecessors to the specific basic block.

**Example of a CFG Transformation**

Consider having the following simple CFG, representing a unipath function or part of it. We assume that \( x \) is a variable that should be versioned. In BB1 we have the declaration of \( x \), and in BB2 a load from it. In BB3 there is some branch condition: if the condition is true, continue to BB5; otherwise, continue to BB4. In BB4 there is a store to \( x \) and then continue to BB5. BB5 is the end of the function. Figure 4.2a demonstrates this CFG.

The compiler reads the annotations and analyzes the code. Next, it replaces the calls to this function to *CreateTask* call and adds a call to *EndTask* before function termination. It also replaces \( x \) with an O-structure that saves versions of \( x \) (\( x' \)) and transforms all the accesses to \( x \) with the O-structure API functions. Figure 4.2b demonstrates the CFG after these transformations. Looking at the CFG, we can see that there are two control paths:

1. \( BB1 \rightarrow BB2 \rightarrow BB3 \rightarrow BB4 \rightarrow BB5 \)
2. \( BB1 \rightarrow BB2 \rightarrow BB3 \rightarrow BB5 \)

We can now observe the acquiring and releasing of \( x' \):

In the first control path there is an acquire in BB2, which has a matching release in
BB4 (first instruction). Then there is another tuple of acquire-release in BB4.

In the first control path there is an acquire in BB2, which does not have a matching release. The compiler recognizes it and solves it. Unfortunately, there is no BB in this path that does not appear in the first path, so the only way is to add a new BB in the CFG. Therefore the compiler inserts a new BB (BB6 in figure 4.2c) and adds the release there.

![Figure 4.2: Adding a Basic Block to the CFG](image)

**Example of a Phi Function Addition**

As previously described, the compiler sometimes has to add $\phi$ functions into the CFG. A $\phi$ function is an instruction used to select a value depending on the predecessor of the current BB. Figure 4.3a demonstrates such a situation. The CFG here is based on the CFG of back insertion into a linked-list. We removed part of the CFG and instructions that are irrelevant for this case. Next, we look at the O-structure API functions; in particular, acquires and releases. In BB2 we acquire for read write permission a version $v'$ of head' O-structure. In BB6 the matching release appears, after acquiring a version of the next node (next' O-structure) for a read write permission. However, in a case where the program reaches BB6 again (back to B5 and then to BB6 again), the release should free previous acquiring, which is the acquire in BB6 (of next' and not head'). This is the reason why, in BB5, the compiler inserts the variable $x'$ that gets the selection made by the $\phi$ function, as shown in figure 4.3b. When the program reaches BB5 from BB4, head' is assigned into $x'$, and when it reaches BB5 from BB6, next' is assigned...
into $x'$. Thus, in BB6 we release access to a version of $x'$.

A similar transformation is performed by the compiler for calculating the version $v''$. $v''$ is the version that the value of $x'$ is released to. In case $x'$ is the $head'$ O-structure, $v''$ is supposed to be the next task’s version. In case $x'$ is $next'$ O-structure, $v''$ is supposed to be $\theta$. Thus, an additional $\phi$ for $v''$ will be added in BB5.

Sometimes a $\phi$ node creation is more complicated. For instance, when there are more than two predecessors, or when another $\phi$ is an input to the current $\phi$. The compiler identifies and processes these cases, using analysis of the CFG structure and the control paths.

![Diagram](image)

**Figure 4.3:** Adding a $\phi$ node

### 4.4.4 Type Safety

LLVM IR is a Static Single Assignment (SSA) representation that provides type safety. Therefore, LLVM does no typecasting; all types must be exact. As a result, all these compiler transformations cannot simply work, and require adaptations. The compiler also remedies this, using bitcasts for pointers and truncation/extension for integers in
order to create a code that can be compiled and run.

4.5 The Algorithm

The Algorithm implemented as two passes of the compiler includes three phases:

1. Phase 1 - This phase operates on the whole program. Finds all the annotations and saves the information about them in data structures, for future needs. For instance, it saves all functions that are annotated with a "unipath" annotation; these will become tasks in phase 2. In addition, it replaces all memory elements that should be versioned, and their accesses, with O-structure’s accesses. See algorithm 4.5.

2. Phase 2 - This phase operates on the whole program too. It looks for all function that are annotated with a "unipath" annotation. Creates tasks: every unipath operation (function) becomes a task. Calls to these functions are replaced with CreateTask calls, with a pointer to the task’s code. A call to EndTask is added at the end of each task. See algorithm 4.6.

3. Phase 3 - This phase operates on each function that is annotated with a "unipath" annotation. Updates the O-structure’s accesses with the appropriate acquiring and releasing, using the correct versions. The root node is released to the next task and other nodes are released to future tasks. Traverses all control paths in the CFG and makes sure that the acquire-release tuple is inserted as hand-over-hand synchronization. Changes CFG when needed and adds $\phi$ nodes when needed. See algorithms 4.7 and 4.8.

4.5.1 Phase 1

Algorithm 4.5 Process annotations and process O-structure variables

1: Process Annotations:
2: Process local variables with "vermem" annotation
3: Process global variables with "vermem" annotation
4: Process global variables with "unipath_head" annotation
5: Process functions with "unipath" annotation
6: For each "vermem"/"unipath" annotated variable update the code:
7: Replace it with a new O-structure variable /*including allocation*/
8: Replace its load with Acquire-Load operations
9: Replace its stores with Acquire-Store-Release operations.
10: Update its uses.

On line 9 the acquiring is from the version that $GetCurEpoch()$ function returned. The releasing depends on the case. If it is a unipath data structure’s head (root), the
releasing is to the next task, and if it is not a unipath data structure’s head, the releasing is to the next writer task, expressed as $\theta$ release.

### 4.5.2 Phase 2

**Algorithm 4.6 Create tasks**

1. For each unipath function (or outlined function that should become a task) in the program:
2. Replace its calls with a `CreateTask` call.
3. Add call to `EndTask` before termination instruction.

### 4.5.3 Phase 3

**Algorithm 4.7 Process unipath tasks**

1. For each unipath task:
2. \hspace{1em} if task is a reader task then:
3. \hspace{2em} Synchronize only the head (root) loads:
4. \hspace{3em} Replace the Acquire-Load with RdWrAcquire-Load and Release after using the value.
5. \hspace{1em} else /*a writer (mutator) task*/
6. \hspace{2em} See algorithm 4.8
7. \hspace{1em} end if

There is no need to synchronize anything except the head, because all other nodes were synchronized on phase 1.
Algorithm 4.8 Handle Writer Task: Fix CFG, Add $\phi$ Nodes and Missing Releases

1: Replace the Acquire-Load with RdWrAcquire-Load
2: BFS traversal from the loop header forward till end of loop (or next iteration)
3: BFS traversal from the loop header backward till beginning of function (or previous iteration)
4: BFS traversal from the loop header forward till end of function
5: Create $\phi$ node and releases:
6: Create $\phi$ nodes in loop header (one for O-structure and one for version).
7: For each O-structure-to-be-released that is not acquired in the loop, and there is a path between its acquires and the loop header, not through the loop:
   8: Add the O-structure and its version to the $\phi$ nodes in the loop header
9: For each loop header predecessor that is inside the loop:
   10: Add the relevant $\phi$/O-structure and relevant $\phi$/version to the $\phi$ nodes in the loop header (this is recursive).
11: For each acquire inside the loop and after it:
   12: Add a release to the loop header $\phi$ O-structure afterwards.
13: Verify acquires and releases:
14: Verify each path in the CFG (starts from entry BB and ends at termination BB):
   15: Count Acquires and Releases
   16: if NumAcquires > NumReleases then
   17: Look for O-structures that were acquired but not released and add a matching release after last acquire-store.
   18: if still unreleased O-structure then
   19: Look for the correct BB and add the matching release there
   20: Add a release-acquire-store-release to the unreleased O-structure there.
   21: end if
   22: end if

Following are some comments on the algorithm:

- On line 2, during CFG BFS traversal, the compiler saves all the information regarding O-structures that should be released, and related $\phi$ nodes.
- On line 3, during CFG BFS traversal, the compiler saves all the information regarding O-structures that should be released.
- On line 4, during CFG BFS traversal, the compiler saves all the acquires (inside loop and after the loop).
- In order to find all control paths in the CFG, as described on line 14, the compiler performs a DFS traversal.
On line 19, the compiler traverses the control path from the end to the beginning and looks for the last BB with exactly one successor. If there is no such BB, it adds a BB just before the terminator BB (and fixes the CFG accordingly).

### 4.6 Time Complexity Analysis

We now consider the input size as $n$, where $n$ is the number of IR instructions.

#### 4.6.1 Analysis of Phase 1

The first phase of the algorithm performs several steps:

- Processing the annotations is a finite and linear operation. There are two kinds of annotations: for tasks and for memory elements that should be versioned. There is a finite number of tasks and a finite number of memory elements that should be versioned. In fact, the number of tasks and the number of these memory elements are linear in the number of IR instructions, and the complexity is therefore $O(n)$.

- Replacing a specific memory element with an O-structure includes a constant number of operations. Therefore, all the replacements will also take less than $n$ operations, which are linear in the number of IR instructions, and the complexity is $O(n)$ again.

- Modifying the accesses (loads and stores) to O-structures is also a finite process with a constant number of operations. As in the previous step, all the modifications are in the number of IR instructions, and the complexity is $O(n)$ as well.

- Updating the uses is also linear in $n$, which means $O(n)$ complexity, because we update only the immediate uses (no need to recursively traverse the def-use chain). The def-use information is already known from previous passes of the compiler, so there is no need to take that into account here.

The total complexity of phase 1 is linear, i.e., $O(n)$ complexity.

#### 4.6.2 Analysis of Phase 2

The second phase of the algorithm performs a few steps:

- Replacing unipath function calls with a `CreateTask` call is limited by the number of IR instructions, because the number of call instructions is smaller than the number of IR instructions. Therefore, the number of replacements is linear in the number of IR instructions and the complexity is $O(n)$.

- Adding a call to `EndTask` before the termination instruction of each task performs a constant number of operations. Thus, the complexity is $O(1)$. These operations
are carried out for all the tasks. Hence, the number of operations is linear in \( n \) and the time complexity is \( O(n) \).

The total complexity of phase 2 is linear, i.e., \( O(n) \) complexity.

### 4.6.3 Analysis of Phase 3

In this phase the work is performed separately for each task, and it is different for reader tasks and writer tasks. Since we defined that there are \( n \) IR instructions in the whole program, the average number of IR instructions per task is \( n/numOfTasks \). We can assume that \( numOfTasks \) is finite and that it has constant complexity. For complexity analysis of this phase we will mark \( n' = n/numOfTasks \). The part of the algorithm that transforms the reader task code synchronizes only the head node. Other nodes were already processed in the first phase. This synchronization takes a constant number of operations; therefore, the time complexity is \( O(1) \).

For mutator (write) tasks, the algorithm traverses the CFG several times, so we should analyze complexity in relation to the number of nodes or basic blocks (BBs) in the CFG (\(|V|\)) and to the number of edges between BBs (\(|E|\)). Each BB includes some IR instructions, although in principle, it may include the same order as the size of the entire function (\(n'\)). \(|E|\) depends on the number of branches in the program. Therefore, both \(|V|\) and \(|E|\) have the same order of magnitude as \(n'\).

The algorithm performs several steps:

- **Replacing the Acquire-Load with acquire\_rw and load** takes a constant number of operations. Thus, the complexity is \( O(1) \).

- **Three BFS passes**: each of the passes requires \( O(|E| + |V|) \). Since both \(|V|\) and \(|E|\) have the same order of magnitude as \(n'\), the complexity is linear in the number of IR instructions. The complexity is therefore \( O(n') \).

- **The creation of \( \phi \) nodes in the loop header** takes a constant number of operations, i.e., time complexity is \( O(1) \).

- **Adding the information for each O-structure-to-be-released that is not acquired in the loop into the \( \phi \) node** requires maximum \( n' \) operations, because in the worst case it requires a traversal of all the CFG. Hence, it is linear in \( n' \), which means \( O(n') \) complexity.

- **The next step is adding information about every O-structure that is acquired in the loop and should be released into the \( \phi \) function**. Although this is a recursive pass, the algorithm does not process a BB twice. In the worst case, in order to build the first \( \phi \) the algorithm needs to traverse all other BBs (except the one that this \( \phi \) is included in). This traversal requires maximum \( n' \) operations. The second \( \phi \) requires \( n' - 1 \) operations, etc. This is an arithmetic sequence and
its summary is $O(n'^2)$ operations. In practice, in common cases it does not take more than $O(n')$ instructions.

- Adding releases inside the loop depends on the number of acquired O-structures that should be released there. Therefore, it is linear in the number of IR operations. Thus, it is an $O(n')$ complexity.

- Verifying control paths requires analyzing every possible control path through the CFG, which is linear in the number of paths. The number of paths is exponential in the number of branches in the task code, and thus exponential in the number of nodes in the CFG, $|V|$, which has the same order as $n'$. Therefore, the complexity of this step in the worst case is exponential. i.e., It is $O(2^{n'})$. This worst case seems to be a pathological case, and there are a couple of reasons for that:

1. An order of $2^{n'}$ paths means that in every BB there is a branch instruction, and not many more instructions. Thus, the task is full of loops, conditions etc., with a very small amount of code. Such a design does not characterize unipath operations.

2. Each procedure CFG has one entry BB and one exit BB, and if not, it can be reformatted. Branches make the CFG wider and wider, while it should converge towards the exit BB.

In the common case, the number of paths is linear in the number of branches, i.e., $O(n'^2)$ paths and $O(n'^3)$ operations in this step.

The total complexity of phase 3 is exponential in the worst case; i.e., $O(2^{n'})$ complexity.

### 4.6.4 Total Complexity

Because of the last operation of the algorithm, the complexity is exponential in the number of maximum branches which appear in one unipath function, $O(2^{n'})$, $n < n'$. Although this is a high complexity, it is typical of this kind of compiler analysis, which sometimes leads to limitations and optimization of those analyses [32]. Basically, the exponential time is critical for programs with a high number of branches. Therefore, it is better for simpler and shorter programs. But, since it runs on each unipath function separately and not on the whole program, and since unipath functions should not be too complicated, we may assume it will not constitute a significant problem. In case the compilation process is too long, a different approach may be considered.
Chapter 5

Methodology

5.1 LLVM Compiler

In this work, we use the LLVM compiler [14] as a base and expand it. At the IR part there are transformations and optimizations of the code, which are target independent. Most of the work in this research was done at this stage of compilation. Our work is based on LLVM version 3.9 (revision 273806).

5.2 User Interface

5.2.1 Compiler Optimizer Flags

In order to turn on our compiler transformation, two flags should be added so that the relevant passes will be executed. We send them to LLVM optimizer ‘opt’ command:

'--verMemPre -UniPathFuncHandler'.

• 'verMemPre' turns on the pass that handles the whole module. Described in section 4.5, phases 1 and 2

• 'UniPathFuncHandler' turns on the pass that handles each of the unipath functions. Described in section 4.5, phase 3

In addition to these flags, we recommend adding the flags: 'disable-inlining -gvm' in order to obtain a simpler code to work on. The flag 'disable-inlining' turns off the inlining pass, i.e., prevents inlining of functions. The flag 'gvm' turns on global value numbering pass, which eliminates fully and partially redundant instructions. It also performs redundant load elimination. Finally, the compiler should load the shared library of our code. Therefore we should also add -load Vermem.so.

5.2.2 Compiler - How To

The compilation process for parallelizing a program using versioned memory contains three steps:
1. **Front end compilation using Clang/Clang++:**
   Input is the source code and output is the bitcode file (.bc) *For instance:* clang++ -std=c++11 -c LinkedList.cpp -o LinkedList.bc -c flag is for generating an intermediate representation of bitcode.

2. **Transformation of the IR:**
   Input is the previous steps’ output and output is a new bitcode file, transformed to use O-structures. *For instance:* opt -disable-inlining -gvn -load Vermem.so -verMemPre -UniPathFuncHandler <LinkedList.bc >LinkedListWithVersioing.bc

3. **Back end compilation using LLVM:**
   Input is the previous step’s output and output is an assembly file, which can be executed afterwards. *For instance:* llc LinkedListWithVersioing.bc -o LinkedListWithVersioing.s
Chapter 6

Benchmarks

In this work, we evaluated several benchmarks for correctness of the compilation process and the code the compiler creates.

6.1 Linked List

Unsorted singly linked list is a dynamic unipath data structure. It contains nodes which have a data field as well as a next field, which points to the next node in a line of nodes. The last node points to null. Operations that can be performed on singly linked lists include back insertion, deletion and traversal/lookup. By using O-structures to store versions of pointers to the root of the list node and of the next field, concurrent operations become quite simple. The root node is acquired for access and released to the next access, according to program order. This way, the correctness of the parallel execution is guaranteed. This serialization of the root accesses happens in all tasks (or routines). As mentioned above, there are three operations on the linked list:

- Back insertion - Iterate over the nodes, starting at the root node, until current node is NULL (i.e., end of the list). At every iteration, current node is promoted to the next node.

- Deletion a node with data $x$ - Iterate over the nodes, starting at the root node, until current node’s data field is equal to $x$. Then, delete the node, and change the previous node’s next field to point the next node. At every iteration, current node is promoted to the next node.

- Lookup for a node with a specific value $x$ - Iterate over the nodes, starting at the root node, until current node is NULL (meaning, $x$ is not on the list), or until current node’s data field is equal to $x$ (meaning, $x$ is in the list). At every iteration, current node is promoted to the next node.
6.2 Sorted Linked List

Sorted singly linked list is a unipath data structure similar to the regular linked list, but it is sorted; i.e., the nodes are ordered according to the value in the data field. Here, insertion is not into the back of the list, but into the correct location of the data on the list. Hence, operations are:

- Insertion - Iterate over the nodes, starting at the root node, until data is greater than the data in the node we want to insert. At every iteration, current node is promoted to the next node. Then, the node is inserted after the previous node and its next field is set to point to current node. When no value is greater than the data in the node we want to insert, we reach the end of the list (Null node) and insert there, as in back insertion.

- Deletion a node with data x - Similar to regular linked list.

- Lookup for a node with a specific value x - Iterate over the nodes, starting at the root node, until the current node’s data is greater than the desired value (meaning, x is not in the list), or until current node’s data field is equal to x (meaning, x is in the list). At every iteration, current node is promoted to the next node.

The transformations of the code are similar to what is shown in section 6.1.

6.3 Binary Search Tree

Binary search tree is a more complicated dynamic unipath data structure. The tree has a node that includes data and two pointers: left node and right node. The tree is sorted: the left node’s data is smaller than the root (or the current node), and the right node’s data is greater. For the binary tree we use only two unipath operations:

- Insertion - Iterate over the nodes, starting at the root node, continue to the left branch (child) if data is greater than the data in the node we want to insert, or continue to right branch (child) if data is smaller. At every iteration, current node is promoted to the left or right node, as we described. Upon reaching the correct location for inserting the new node, it is inserted as a leaf in the tree. This insertion is not ideal, since in order to keep the tree balanced, we may need to change locations of nodes inside the tree. The balancing, as well as deleting a node from the tree, is not a unipath operation. Therefore, it makes managing the concurrency and versioning much more complicated, and it is not in the scope of this work.

- Lookup for a node with a specific value x - Iterate over the nodes, starting at the root node, in a way similar to the insertion. The lookup stops when it reaches a NULL node (meaning, x is not in the tree), or until current node’s data field...
is equal to $x$ (meaning, $x$ is in the tree). At every iteration, the current node is promoted to the left or right node, as described above.

The deletion operation is not part of this benchmark, because it requires a sort of balancing of the tree, and balancing works on several paths of the tree. Therefore deletion is not a unipath operation.

![Linked-list](image1.png)

![Sorted Linked-list](image2.png)

![Binary Tree](image3.png)

Figure 6.1: Benchmarks: Data Structures

### 6.4 3-Way Matrix Multiplication

The last benchmark we examined is a basic and useful benchmark consists of multiplying three matrices in a dense representation (2D array), $A \cdot B \cdot C$. This benchmark is not a data structure with unipath operations, but it does have a potential for parallelism using versioned memory. Unlike the unipath benchmark, each operation becomes a task; in a matrix multiplication loop, iterations become tasks. In order to make it similar to the way the compiler works with unipath operations, the programmer should mark the loop so that LLVM will outline it as a function. It is a fairly simple operation, and there is an annotation for doing so. In this way, all passes of the compiler can work.
Loop parallelization was not the focus of this work, and it is easier to parallelize in other ways. But it shows potential for using the versioned memory execution model, also including this compiler for different programs.

When multiplying three matrices, the first two must first be multiplied, and the resulting temporary matrix should then be multiplied with the third matrix. At the first stage, each element of the temporary matrix can be calculated independently of the other elements (since the temporary matrix is a static data structure, writes need not be ordered). Dependencies are first introduced at the second stage, in which the temporary matrix becomes the input of the second multiplication. As a result, the second stage must stall when trying to read temporary values that have not been calculated yet. Fortunately, the total amount of work is very high on dense matrix multiplication ($O(n^3))$. This renders dependent operations negligible, yielding a high level of parallelism. Our compiler knows how to deal with these kinds of benchmarks; it successfully transforms the code to use O-structures and then manages versions of them.
Chapter 7

Correctness

Our claim is that the compiler preserves the functionality of the program, as it was before changing it. It can be seen that there are properties of the code that the compiler does not change. In particular, the loop invariant (see section 7.1) should be saved. Looking at the IR code before and after transformations made by the compiler, we can see that the functionality remains the same:

- The compiler touches only those instructions that are relevant for this transformation. Thus, a code that is not related to O-structure variables is not changed at all.

- Some variables become O-structure variables. The compiler searches for all their uses and definitions in the code and then changes their type and their accesses. This change does not affect the functionality of the program; it is just a semantic change.

- The compiler replaces every load or store that is removed (because it accesses a variable which no longer exists in the program) by a new load from/store to a matching O-structure, so that the functionality stays the same.

- Every task consistently acquires access permission to/reads from/writes to the same version number during the execution (for all memory addresses). Since the numbers of versions are determined by the order of the serial program, the task sees a snapshot of the unipath data structure, which is exactly what is would see if the program were run serially.

- Transformations in the CFG structure are carried out only when the compiler needs to add a release, and the only option for adding it is in a new basic block (BB). Since the new BB contains only the release call instruction and a branch to the next BB, it does not influence the program.

- Immediately after acquiring access permission to the next node, the compiler releases the acquiring of the previous node. No deadlock could possibly happen.
7.1 Loop Invariant

In general, dynamic data structures contain nodes with pointers that connect them. As described in section 3.2, during unipath operations, traversal of the path to reach any node in the structure is always the same. It begins at the root node and continues iteratively to next node on the path, until reaching a specific node/location on the path, or until end of the path. This *iterativity* appears as a loop which is the heart of the traversing, and this traversing is the heart of any operation (insertion, deletion or lookup). In general, unipath function (operation) code is composed of three parts:

- Code before the loop - Initializations, handling root node.
- Loop code - iterating over the data structure and finding the desired location/node.
- Code after the loop - if the location exists, mutation of the data structure, exiting from the function.

If the function is not written exactly as specified above, it can either be reformatted or we can generalize our proof and show the correctness of the code after compiler transformations. In all three parts, acquires and releases may be included in the code.

Our claim is that no matter what the operation is or what is included in each part, the following loop invariant is guaranteed. The only requirement is that the loop invariant be valid before the first iteration. Otherwise, the compiler may restore the original code. A *loop invariant* is a property of a program loop that is true before (and after) each iteration. Our loop invariant is as follows: **Before entering the loop and at end of the loop exactly one O-structure is locked.**

Before the loop and inside the loop there may be a few acquires. But, as can be seen in the algorithm 4.8, starting from line 5, the previous acquire is always released when the next acquire is completed. More specifically, at section 13, the algorithm verifies that each acquire has a corresponding release after the subsequent acquire. This verification is important not only to ensure the loop invariant. It also contributes another aspect of validation of the code created by the compiler.

7.2 Acquiring and Releasing Correctness

As mentioned above, the compiler verifies that for each path in the CFG, every acquire access permission to a specific version has a matching release. As mentioned at the third phase of the algorithm (4.8), from line 13, the compiler counts the number of acquires and releases in each control path. If there are fewer releases, it finds out the missing releases and adds them to the correct location in the CFG. This is a self-check by the compiler, which guarantees that every time we acquire an O-structure we also release it, so that other tasks can access it. This guarantees the progress of at least one thread when executing multi-thread, thereby successfully avoiding deadlock.
Chapter 8

Conclusion And Future Directions

8.1 Conclusion

This work introduces an LLVM-based compiler for new hardware architecture with a versioned memory system. The system facilitates parallelism in task-based execution model. The compiler exempts the programmer from requiring a deep understanding of the hardware, and almost automatically manages the memory versioning. We describe the versioning model and its basic memory element, named O-structure. We show how it eliminates false- and anti- dependencies, while enabling OoO execution that keeps a sequential semantic. We then present the compiler and demonstrate how it transforms a serial code into a parallel one that efficiently exploits O-structures. Finally, we describe the applicability for some commonly used data structure algorithms.

8.2 Future Directions

Our current implementation relies on programmer annotations, but enhanced static analysis techniques such as shape analysis may help to avoid this necessity. There may be a way to get rid of these annotations and automate the compilation and the parallelization process. Shape analysis may find out that a function is a unipath operation and analyze the code and identify the memory elements that should be versioned. Unfortunately, this kind of research is not trivial, and is outside the scope of this work. Binary search tree is one of the benchmarks we examined in this work. For this data structure we executed only two operations, insertion and lookup, and did not check deletion, as described in section 6.3. The reason for this is that it may necessitate balancing the tree, which works on several paths, i.e., not a unipath operation. We believe that the option of converting tree balancing into several unipath operations may be explored and if found viable, it can enable the use of our compiler for deletion and other operations. Our compiler can potentially work on other types of task-based
programs, as mentioned in section 6.4. We showed that it is usable both for loops and functions, such as unipath operations. Therefore, it may be useful in other cases, where versioning memory model has the potential to achieve high parallelism.
Bibliography


46


את המודר שBrains לבןعلى מספר את הגניזה של מקוותיו של חורז וראינו שאנו.

את השיתוף הרצוי למסגרת את כל הלוח המוזיקה החרט המהירות. המודר דהירב סופי רוחם

 başילוש על המודר בשיבור, אפשירית, יושם ישילה.
O-structure: A multilayered approach to database design. It involves the use of a hierarchy of levels, each with its own set of constraints and rules. This approach is particularly useful for large, complex databases where traditional methods may not be sufficient.

The key idea behind O-structure is to break down the database into a series of layers, each with a different level of abstraction. The lowest layer contains the most detailed information, while the higher layers provide a more general view of the database.

At each level, O-structure defines a set of rules that govern how data is stored and accessed. These rules are enforced by the database management system, ensuring that data is consistent and accurate.

In conclusion, O-structure provides a powerful tool for managing large, complex databases. It allows for flexible and efficient data management, making it an ideal choice for a wide range of applications.
תקציר

บทיםأمانירה, עטנסיאלזים信阳טנוליגים של המוח, נוירה מרכזת להגדלת תדר המונברת על ח紀錄 הקים. תכנונה מתכון, די לחסניה האנה ע她的 ענינו שיש לברך ה userModel של התוכנה, ועל ידיобрיה יימשכו מבegade שמשמו במשתמשים, וב śmנים,訴נה, לותר שסיפורים עכשווית ולה להדיל את המיתר ובочנה מקילי, חבית לקוד כל עדין שלשבו שوعدת.

אבראהים

בammenת מתכון את התוכנה המוביל, פותחו ספירות שמעו וסבובייו "رصدועה, די להקל על המיתר התוכנה לכל עלי ומוחה. חתוננו יכלו ישים הקיםaira להגדלת תדר המיתר, וכפיון שהשקעה היהتشמיד לפי התקפה לשעיה בחלק לשחקית, הפיכ ממד צלולה לצלולה של הקיםaira והתקנה י_Interface. בטוח ששה מקורות העבר של הקיםaira, שידה ושם מקורות של מעבר משמעidence, שיה דאואיר שסיפורים בתקיפה חצרה עלי ותקיפה שסיפורים עכשווית של העברות של הקיםaira עם השימור של השירים: הלוח, לותר שסיפורים, וב טענו עניינו, ובדבר את עניינו, וב were עניינו, וב כל עניינו, וב נושא עניינו. מתכוןسطה הצילוח בכיסו ובשון הקיםaira (שמשתגרר את התוכנה לשפת המכות) יהיה מעטים.

מה Halifax, שימורו של שימור שבירות של רחוב המיתר, וב שימורו של שבירות של רחוב הקיםaira ור לאות של מיקום איזון. ברוד אלשלום, סל בחלקה של הקיםaira, דואיגו את הקיםaira, בחלקה של הקיםaira, בשוב את הקיםaira, בחלקה של הקיםaira, בisOpen וסבובייו "رصدועה, די להקל על המיתר התוכנה לכל עלי ומוחה. חתוננו יכלו ישים הקיםaira להגדלת תדר המיתר, וכפיון שהשקעה היהتشמיד לפי התקפה לשעיה בחלק לשחקית, הפיכ ממד צלולה לצלולה של הקיםaira והתקנה י_Interface. בטוח שше מקורות העבר של הקיםaira, שידה ושם מקורות של מעבר משמעidence, שיה דאואיר שסיפורים בתקיפה חצרה עלי ותקיפה שסיפורים עכשווית של העברות של הקיםaira עם השימור של השירים: הלוח, לותר שסיפורים, וב טענו עניינו, ובדבר את עניינו, וב כל עניינו, וב נושא עניינו. מתכוןسطה הצילוח בכיסו ובשון הקיםaira (שמשתגרר את התוכנה לשפת המכות) יהיה מעטים.
הממחק עשה בהנחייתו פורפ', יואב עיזיון.

תודעה נמוכה לטכניות – מכון טכנולוגי לישראל

על התמיכת הכספית הנדרשת מהמדים

תודה

ברצוני להביע את תודתי העומדת על הערכתי של להקתו של פרופ', יואב עיזיון, על התמיכה, ההנאה, המבɫמות והענ đâyות לאורך כל הדור. יזדז לש פרו' עץון תמידالته בהפקתו בבטן שמחתי; ריכזה עבדה ושלא. אלجتماع יכולים העמדים שלשוח ממוחזק.

אף מבסקת להוזיחו של מלביי העמדות המבחרונות, שנותינו לי את הקופס היםודים והמחנכים עזרו בברוחם בוצעו ב.radיו lắpוטו והמהמות במחנה. הפרט, ברצוני להזיחו לנעד ע SqlCommand.

אחת היסודות, ברצוני להזיחו לבני משפחתי:

• לדורו הקריין לי מצאתי,نعמי וצבי גלידיא, על הסדרי על התמיכה הנפשית לאורך על הת страховין וזרכלי.

• לדורו והמי,ريفיה וimité מיליס, על התמיכה והענדה.

• לילת הא馒ינים: אורי, אליה, יפה, נורה והולול, שאנייה את zipper יימודים אמידים ומספרฟัน להבילי הימים, נוע, סטימני האמנים ב, בג cryptoc הלאנות: בערמון. את מענייני מקו מארט

• הסבלום,너יה הנטיבת והמחפשים המלチーム בביתו ושלוחם.

תודה"
תוכנה لنיהול חומרים
היתמכת ביז_supply מрова גרסאوات

יחבר על מחקר

לשם مليי חלקי של הדרישות ל vidéות החונא
מנהסר למדעי במדעי המחשב

תהליך מיילס

האריך לשטח sağlık - �ofire טכניוגרפי ישראל

מרץ 2017

חיפה

אדר תש"ע

Technion - Computer Science Department - M.Sc. Thesis MSC-2017-08 - 2017