Verifying Permutation Rewritable Hazard Free Loops

Verifying Permutation Rewritable Hazard Free Loops

By
Michal Dobrogost B.Math.
A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University
© Copyright by Michal Dobrogost, August 4, 2011

MASTER OF SCIENCE(2011) McMaster University

COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: Performance Simulation with the Coconut Multi-core Framework for the Cell/B.E.

AUTHOR: Michal Dobrogost B.Math. (McMaster University)

SUPERVISOR: Dr. Christopher Anand

NUMBER OF PAGES: xiii, 49

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defense committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

We present an extension to the language of Atomic Verifiable Operation (AVOp) streams to allow the expression of loops which are rewritable via an arbitrary permutation. Inspired by (and significantly extending) hardware register rotation to data buffers in multi-core programs, we hope to achieve similar performance benefits in expressing software pipelined programs across many cores. By adding loops to AVOp streams, we achieve significant stream compression, which eliminates an impediment to scalability of this abstraction. Furthermore, we present a fast extension to the previous AVOp verification process which ensures that no data hazards are present in the program's patterns of communication. Our extension to the verification process is to verify loops without completely unrolling them. A proof of correctness for the verification process is presented.

Acknowledgments

I would like to thank my supervisor Dr. Christopher Anand for his flexibility and guidance during the many days he welcomed me to work at his house. My co-supervisor, Dr. Wolfram Kahl also deserves thanks for teaching the interesting course that motivated me for this thesis.

Many thanks to my whole extended family for their never ending support and my brother for his never ending belief.

Contents

Abstract iii

Acknowledgments v

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Motivation 1

1.2 Novelty 2

1.3 Impact 2

1.4 Trademarks 3

1.5 Thesis Organization 3

2 Notation 5

2.1 Map Modification Notation 5

2.2 Disjoint unions 6

3 Background and Previous Work 7

3.1 Concurrency Verification 7

3.2 AVOp Langauge Restriction 9

3.3 Motivating Example 9

3.4 Strictly Forward Inspection of Partial Order 10

3.5 The Follows Map (Φ) 12

3.6 Discussion of the Follows Map (Φ) 13

3.7 Merging Φ Slices to Strengthen Our Partial Order . . . 15
3.8 State . . . 16
3.9 Verification Step . . . 16

4 The Loop AVOp **19**
4.1 Loop AVOp Indexing . . . 19
4.2 Loop definition . . . 20

5 Efficient Safety Verification of Looping Programs **23**
5.1 Locally Sequential Loops . . . 23
5.2 Bumping State to the Next Iteration via Λ . . . 24
5.3 Loop Short Circuit Theorem . . . 25
5.4 Verifying Nested Loops . . . 27
5.5 Verifier Run Time . . . 28

6 Rewritable Loops **31**
6.1 Motivating example . . . 32
6.2 The rewrite map: ρ . . . 33
6.3 Formulation of AVOps as functions on the State . . . 34
6.4 Induced Rewrites and Rewriting AVOps . . . 35
6.5 Short Circuit Theorem for Rewritable Loops . . . 36
6.6 Verifier Run Time on Rewritable Loops . . . 37
6.7 Memory requirements . . . 38

7 Extension to Full AVOp Set **39**
7.1 Extension of AVOp State and Hazard Checking . . . 39
7.2 Extension of the Rewrite Map . . . 42
7.3 Extension of Loops for Accessing Global Memory . . . 43
7.4 Extension of Verification Algorithm . . . 44

8 Future Work **45**
8.1 Restricted Global Memory Access for Verification . . . 45
8.2 Stronger Short Circuit Theorem for Rewritable Loops . . . 46

8.3 Breaking up AVOp Streams to Increase AVOp Feed Rate 46
8.4 Scheduling 47
8.5 Fault Tolerance 47

9 Conclusion **49**

List of Figures

3.1 Locally sequential program 11
3.2 Φ is defined for other cores 13
3.3 Visualization of Φ-dependency. 14
3.4 Φ map at instruction $SendSignal\ s_3 \rightarrow c_3$. 14
3.5 Φ map at instruction $WaitSignal\ s_3$. 15

4.1 Example using the Loop AVOp 21
4.2 Example of an unrolled loop 21
4.3 Example with nested loops 21

5.1 Non-rewritable loop example 29

6.1 Affects of an inner loop 35
6.2 Rewritable Loop unrolled into a loop without a rewrite. 36

7.1 Defining diagram for projected rewrite 43
7.2 Accessing global memory 44

List of Tables

TODO : latex not generating output.

Chapter 1

Introduction

This thesis is concerned witht the extension of a language for parallel computation with loops. A major strength of this language is an algorithm which quickly verifies the determinisim of the program. The algorithm is modified to take into all our other extensions.

1.1 Motivation

Parallel computer architectures have now become almost universal, with even mobile phones containing dual core processors. Taking full advantage of the performance available in a computer system requires taking full advantage of the parallelism offered by that system. Opposed to the high performance requirement is the need to verify the code's requirements. Programming such architectures has long been recognized as a difficult task. Both requirements for performance and verifiability are impeded by such hazards.

In a traditional locking architecture, performance can be degraded by thread contention over the same lock. It is fundamental to locks that they cause blocking and it is the programmers job to attempt to structure the programming in order to minimize this. This is a difficult task requiring careful structuring and ordering of acquiring and releasing the locks. The programmer is also burdened with the job of balancing between the overhead of an additional lock versus contention for the same lock.

Verifying parallel code using locks is difficult. There are a plethora of things that may go wrong, including: deadlocks, livelocks, priority inversion, and race conditions. Since the underlying environment is non-deterministic, it is difficult to create test suites or even debug specific problems.

While locks synchronize by letting only one thread touch a specific piece of data, we synchronize by a pair of instructions: SendSignal (non-blocking) and WaitSignal (blocking). The specific coupling of which core is sending the signal and which one is waiting allows us to keep track of the relative ordering of instructions across cores. This is differnt to locking in which any core may acquire a lock and block any other core. In fact, we can keep track of exactly which instructions are known to have finished on a core, given that an instruction completed on another core.

In general, locks require two way messages between two cores to acquire and release the lock since the lock must exist on some core, or otherwise, be in shared memory. This means that locking is a fundamentally synchronous mode of communication. Synchronous communication scales poorly as more cores are added. Our asynchronous approach avoids this problem and exposes opportunities for out-of-order execution. The fact that this approach scales up to many cores makes loops all the more necessary to prevent two possible bottlenecks: first, the language interpreter itself and second, the transportation of large program texts.

Our approach is especially well suited to Architectures such as the Cell/B.E. [CIC].

1.2 Novelty

In this thesis we address these issues by extending Atomic Verifiable Operation (AVOp) streams (introduced by [AK08]) to allow loops. An AVOp is the basic instruction in our Domain Specific Language (DSL). It is analogous to a processor's instruction.

AVOp streams allow the performance to be maximized by introducing an algorithm for scheduling across different threads of execution so as to minimize contention. In this context, contention means the need to stop execution while blocked waiting to synchronize with another thread. At the same time, a verification algorithm can determine whether the program is deterministic despite any possible execution order across the threads.

AVOp streams form a programming language. The key contribution of this thesis is making this programming language more expressive by supporting repetition via a looping construct. At the same time, the verification algorithm is extended to efficiently verify in the presence of looping constructs. Other language constructs had to be modified to accommodate these changes.

The looping construct is then extended to apply a rewrite on the cores, signals and data buffers. This is inspired by the register rotation mechanism of the IA-64 architecture but is much more general. The aim is to aid with software pipelining.

1.3 Impact

The AVOp language presented in [AK08] is meant to be highly scalable and limits itself to asynchronous communication in order to do so. However, because the program text is presented in a linear fashion, the text length is proportional to run time length in the "Loopless AVOp language" of [AK08]. Adding a looping construct is important for two reasons. First, it significantly extends the set of complicated communication patterns that can be expressed in the language while still being able to reason about them. Second, it is vital for compressing the AVOp stream which minimizes the overhead of feeding AVOps to a large number of cores. This is a key property required for the scalability of the AVOp language to super computers or large clusters.

1.4 Trademarks

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Cell/B.E. is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both.

SPARC is a trademark of Oracle Corporation.

Other company, product, or service names may be trademarks or service marks of others.

1.5 Thesis Organization

We first introduce notation that may be unfamiliar to the reader. A restricted form of the AVOp language is presented after which simple loops are introduced. The fast verification algorithm is presented, and a proof of correctness is shown. The language is then extended to handle loops with permutation rewrites and the proof is extended to handle this new construct. We incorporate all the AVOps of the language. Finally, some future avenues for research are covered.

Chapter 2

Notation

We introduce some notation that will be used throughout this thesis. First, we present our notation for describing modifications to a map. Second, we present our notation for disjoint unions. In both cases we present some simple examples to get a feel for the notation.

2.1 Map Modification Notation

Our map modification notation is defined here. Given a map $f : X \to Y$, $x \in X$ and $y \in Y$ then $[x \mapsto y](f) : X \to Y$ is defined by:

$$([x \mapsto y](f))(x') = \begin{cases} y & \text{if } x = x' \\ f(x') & \text{otherwise} \end{cases} \tag{2.1}$$

As an example of this notation, suppose m is defined as:

$$m = \begin{pmatrix} 0 \mapsto 1 \\ 1 \mapsto 2 \\ 2 \mapsto 3 \end{pmatrix}$$

$[0 \mapsto 0](m)$ specifies the same mapping as m except for the single entry at 0 which now maps to 0.

$$[0 \mapsto 0](m) = \begin{pmatrix} 0 \mapsto 0 \\ 1 \mapsto 2 \\ 2 \mapsto 3 \end{pmatrix}$$

Multiple modifications are denoted using common function syntax:

$$[1 \mapsto 100]([1 \mapsto 10](m)) = \begin{pmatrix} 0 \mapsto 1 \\ 1 \mapsto 100 \\ 2 \mapsto 3 \end{pmatrix}$$

2.2 Disjoint unions

Disjoint unions, denoted by $\oplus$, are used to represent many of the data types used by the verifier algorithm. Disjoint unions allow us to treat multiple distinct types as a single type. For example, we could specify the type for a non-empty singly linked list of integers as

$$IntList = \begin{pmatrix} \mathbb{Z} \\ \oplus \\ \mathbb{Z} \times IntList \end{pmatrix}$$

Notice how the data type could be in either of the cases separated by $\oplus$. First, a list can consists of a single integer. Second, an integer can be prepended to a list to form another, longer, list. In this case the type is recursive but there is no such requirement in general.

This can be represented as an Algebraic Datatype in the programming language Haskell (see [PJ+03]) as

```
data IntList = Base Integer
             | Recr Integer IntList
```

Chapter 3

Background and Previous Work

We repeat some introductory material from [AK08] but rewritten for clearity and compactness. New notation is required in the context of the looping construct, which changes all AVOps (see Section 3.1). The aim is to introduce the current state of verifying AVOp streams and detail the benefits of verification.

The AVOp language is introduced, followed by motivating examples which illustrate example hazards that are detected by the verification algorithm. The follows map Φ and signal map σ are introduced and we present an extended discussion of these maps in Section 3.6. The state that is kept by the verification algorithm along with a single step of the algorithm is then covered.

3.1 Concurrency Verification

Verification needs to satisfy two use cases. First, developers are interested in using it to understand the source of defects in their code. Second, users (such as regulators) are interested in guaranteeing that safety-critical software is correct. Concurrent errors are inherently non-local, yet, the type of error and one of the instructions involved in creating an unsound program are identified by this verification algorithm.

We envisage two verification steps which make sense in the context of multi-core parallelism: Reduction of the parallel program to a pure data-flow graph, the harder of the two, and matching the data-flow graph to a specification. For problems such as structured linear algebra, it is easy to generate single- and multi-level pure data-flow graphs and convert them to an appropriate normal form. This makes the second step easy.

The ideas influencing this approach are those of the superscalar, out-of-order execution model of a CPU. Instructions may be executed out of order, but have to be presented in a linear order. The out-of-order semantics must match the sequential program semantics. This is the case for our AVOp programs that pass verification as defined below.

We will use the following terminology (definitions directly from [AK08] but extended to our purposes):

- A *program* is a list of AVOps. The order of AVOps in this list is referred to as the *presentation order*.

  AVOps are transmitted to every core in this order. The ordering of execution of AVOps on a single core corresponds to the restriction of the program to that core. However, each core buffers its instructions and executes them without additional synchronization. Thus, the original program order is not necessarily the same as the resulting execution order (AVOps on separate cores may execute in an order different from the presentation order).

  In [AK08] a program was a list of pairs of AVOps and the core that the AVOp is executing on. We need to change this for our results later in the thesis. The core that an AVOp is running on is the first argument of the AVOp.

- A program is *order independent* if, given the same input in main memory, all possible execution orders terminate and produce the same output in main memory. Intermediate values computed and temporarily stored in private or global memory may differ.

- A program is *locally sequential* if every (SendSignal c_1 c_2 i) is followed by a corresponding (WaitSignal c_2 i) in the presentation order.

  Note that it is easy to construct order-independent programs without this property, simply by ordering the instructions by core.

- A program is *safely sequential* if it is locally sequential and order independent.

The importance of verification is discussed in [AK08]:

> The programmer wants to know that programs are order independent. Checking this is very expensive. The programmer understands sequential programs, and with some difficulty can understand parallel programs "close to" sequential programs. Locally sequential programs are our attempt to formalize this concept. The key to usability is the fact that we can efficiently identify the safely sequential programs within the class of locally sequential programs.

The definitions above are presented in relational language:

1. The presentation order is a total order.
2. AVOps on a single core are totally ordered.
3. AVOps on different cores have a partial order derived from the use of synchronizing primitives. This partial order is the intersection of all possible execution orders.
4. There is an algorithm (see [AK08]) that can identify the partial order (3) as a subrelation of the presentation order (1) if the presentation order is safely sequential. Doing this will require memory in $\mathcal{O}(|\text{cores}|^2 \cdot |\text{signals}|)$.

3.2 AVOp Langauge Restriction

The material presented in this thesis focuses on the SendSignal and WaitSignal AVOps. Other AVOps, which are handled similarly to this case, are presented in section 7.

- SendSignal c_1 c_2 s — core c_1 sets the boolean signal s on core c_2. If the signal was already set, it remains set (this is a hazard).
- WaitSignal c s — halts execution on core c until signal s is set. Clears the signal s and resumes execution once this condition has been met.

We provide a brief overview of AVOps whose full coverage we delay until later:

- SendData, WaitData — transfer data between cores.
- LoadMemory, StoreMemory, WaitDma — transfer data to and from main memory.
- RunComputation — runs a computation kernel.

3.3 Motivating Example

Before explaining the efficient method of verification, it is important to understand that locally sequential programs are not necessarily sound. We start with the following program:

index	AVOp
1	long computation on c_2
2	SendSignal c_1 $s \rightarrow c_2$
3	WaitSignal c_2 s
4	computation on c_2
5	SendSignal c_3 $s \rightarrow c2$
6	WaitSignal c_2 s

To be able to better visualize the meaning of this program we expand it by listing each core's AVOps separately. The remainder of this section is quoted from [AK08].

index	core 1	core 2	core 3
1		long computation	
2	SendSignal $s \rightarrow c2$		
3		WaitSignal s	
4		computation	
5			SendSignal $s \rightarrow c2$
6		WaitSignal s	

Remember that each core executes independently of the other cores, except for blocking AVOps which halt execution until some kind of communication (signal, change in data tag, DMA) is confirmed to have completed. Therefore, in this case the most likely instruction completion order has core 3 executing the SendSignal as soon as it is queued, allowing the signal to be sent before core 2 has received core 1's signal and cleared the signal hardware:

index	core 1	core 2	core 3
2	SendSignal $s \rightarrow c2$		
5			SendSignal $s \rightarrow c2$
	second signal overlaps the first, only one registered		
1		long computation	
3		WaitSignal s	
4		computation	
	no signal is sent, so the next WaitSignal *blocks*		
6		WaitSignal s	

To be precise, completion of the SendSignal means that the signal has been initiated by the sender, and reception may be delayed, so the signal from core 3 could even arrive before the signal from core 1. In either case, neither signal will arrive after the first WaitSignal, so the second WaitSignal will wait forever, and this program execution will not terminate.

The problem is caused because there are no signals or data transmissions enforcing completion of instruction 5 to follow completion of instruction 3.

This example, when considered as part of a longer program, also demonstrates a possible safety violation with the valid completion order:

index	core 1	core 2	core 3
1		long computation	
5			SendSignal $s \rightarrow c2$
3		WaitSignal s	
4		computation **using wrong assumptions**	
2	SendSignal $s \rightarrow c2$		
6		WaitSignal s	

In this case, the computation 4 might rely on assumptions that are only available once the SendSignal 2 completes — for example, it might initiate a DMA to core 1, which could arrive before some earlier instructions on core 1 are still writing to the DMA's target area, thus invalidating the DMA transfer.

3.4 Strictly Forward Inspection of Partial Order

We directly quote [AK08] in this section with terminology standardized on "locally sequential" as opposed to "weakly sequential".

We have seen that locally sequential programs are not always sound. The advantage that locally sequential programs have over general programs is that order-inducing instructions must occur before instructions whose well-definedness depends on them. Using simple inference to maintain a list of known ordering relations between instructions on different cores, we can inspect the program in order (even though it can execute out of order).

Good Program

index	core 1	core 2	core 3
1		long computation	
2	SendSignal $s \rightarrow c2$		
3		WaitSignal s	
4		computation	
5		SendSignal $s \rightarrow c1$	
6	WaitSignal s		
7	SendSignal $s \rightarrow c3$		
8			WaitSignal s
9			SendSignal $s \rightarrow c2$
10		WaitSignal s	

Figure 3.1: Locally sequential program

Referring to Figure 3.1, we see that after instruction 3, we know that instructions on core 2 will complete after instruction 2 on core 1 (and any previous instructions). We know nothing about the relative execution of instructions other than this until instruction 6. At this point we know that instructions after and including 6 on core 1 must execute after any instruction before instruction 5 on core 2. After instruction 8, we have the relation that this and further instructions on core 3 execute after instructions up to instruction 7 on core 1, but since instruction 7 on core 1 executes after instruction 5 on core 2, we also know that instruction 8 on core 3 executes after instruction 5 on core 2.

Contrast this with the example in 3.3, where we never have any relations between instructions on core 3 and other cores. Keeping track of these relations is the key to efficient verification of correctness.

There is one other class of (bad) example to keep in mind: the situation where, in presentation order, two cores send the same signal to a third core without a wait on the third core to separate them. This case is visually apparent, because the sends overlap in program order, but we have to safeguard against it nonetheless.

Clearly, the same considerations apply to data transfers. With data transfers, we must additionally check the values of the data tags, because if the existing tag matches the tag being transferred in, the WaitData instruction will never block execution, even if the data is not available.

3.5 The Follows Map (Φ)

Recall that a locally sequential program is a sequence of AVOps (communication primitives) with a totally ordered presentation. The order does not fully determine the execution order, but it does define the analysis order for the verification algorithm. Without this order, there would be no linear-time verification algorithm. Given the order, it makes sense to talk about the state maintained by the verification algorithm. This state encodes the information about the possible execution states required to verify the soundness of the program in this way.

We introduce the following notation:

- I — is the set of AVOps (Instructions).
- C — is the set of core IDs that cores are referenced by.
- S — is the set of Signal IDs that may be signaled by SendSignal.
- P — is the fixed program we are verifying.
- $N = \{0, \ldots, \text{length}(P) - 1\} \subseteq \mathbb{N}$ — is the set of indices into P considered as a list. This allows us to consider the program as a total function $P : N \to I$. N is considered to be ordered with the standard ordering of the natural numbers.
- $N_c = \{n : N | \, \text{core}(P(n)) = c\}$ — is the restriction of the index set to instructions on c, for all $c : C$ where "core" is a function that returns what core the AVOp is running on.

Form [AK08]:

> We are interested in the partial strict-ordering $\prec$ on N where $n_1 \prec n_2$ means "the AVOp at n_1 *necessarily* completes before the AVOp at n_2". The reflexive closure of this is frequently known as "happens-before relation" [FG], or "Mazurkiewicz's trace" [Maz, God96].
>
> For a fixed core, $c : C$, the instructions in the program assigned to c are completed in order, so we demand that, for each core $c : C$, the restriction of $<$ to N_c is included in $\prec$:
>
> $$\forall c : C \,.\, \forall n_1, n_2 : N_c \,.\, n_1 < n_2 \Rightarrow n_1 \prec n_2 \tag{3.1}$$
>
> There is no *a priori* order between AVOps assigned to distinct cores, and this may lead to unsound programs like the example in 3.3. Ordering is determined by transition dependencies, which in our case means blocking communication.
>
> It would be sufficient to calculate $\prec$ completely, but only a small portion of this partial order is required at a time. Since AVOps on a single core are totally ordered, it is easy to calculate the entire relation $\prec$ if, for each AVOp index n and distant core, the latest AVOp known to complete before n, and the earliest AVOp known to complete after n are stored.

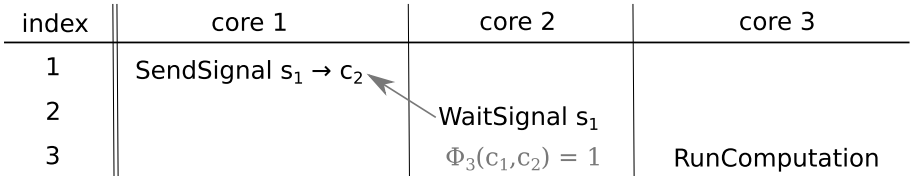


Figure 3.2: Φ is defined for other cores. For convenience, $\Phi_n(c_1, c_2)$ is defined for all n, even if $n \notin N_{c_2}$, to be the index of the last instruction on c_1 known to complete before an instruction on c_2 up to and including index n. This figure is not from [AK08].

We will see that it is sufficient to store only the latest instruction known to complete before the current instruction, and convenient to repeat information for all cores, in the form of a Follows map indexed by $n \in N$:

$$\begin{aligned} F &= C \times C \to N \\ \Phi_n &: F \\ \Phi_n(c_1, c_2) &= \max_{\leq}\{m : N_{c_1} \mid \forall n' : N_{c_2} \,.\, n \leq n' \Rightarrow m \prec n'\} \ . \end{aligned} \tag{3.2}$$

Note, in particular, that if $n \notin N_{c_2}$ this definition does not say that is the latest instruction on c_1 such that all future instructions on c_2 follow m.

Figure 3.2 indicates that the AVOp at index $\Phi(c_1, c_2)$ is always a send instruction executing on core c_1.

In [AK08] use is made of an implicitly defined $\leq$ relation which is the ordering of AVOps in the presentation order.

3.6 Discussion of the Follows Map (Φ)

Since Φ is at the core of our Verification algorithm, the details of the definition given above are reiterated here. Φ keeps track of dependency information between each core. Specifically, it keeps track of how far execution is guaranteed to have progressed on a given core given an instruction on another core.

These dependencies are visualized for a very simple program in Figure 3.3. We see here an execution trace of a program running on three cores. The execution is broken down with a single AVOp executing at any given time. An AVOp a points to another AVOp b if b must complete execution before a can complete execution.

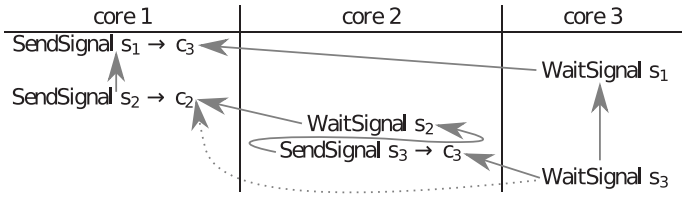


Figure 3.3: Visualization of Φ-dependency.

Direct dependencies are visualized by arrows. In the diagram we see two different cases. First, there is a dependency between *WaitSignal* s_1 and its corresponding *SendSignal* as can be seen by the top most arrow. Second, there is an obvious dependency, made explicit, between AVOps executing on the same core as can be seen by the curved arrow on core 2. The second type of dependency is not actually kept track of in the Φ map as it only tracks dependencies between different cores.

Indirect dependencies are visualized by the dotted arrow. These dependencies are computed by merging together Φ maps stored at different points in the program's execution. In this example we can see that *WaitSignal* s_3 cannot complete execution before the corresponding *SendSignal* on core 2, or the vertical arrows on cores 1 and 3. However, *SendSignal* $s_3 \rightarrow c_3$ cannot complete execution before *SendSignal* $s_2 \rightarrow c_2$ by following the dependency arrows. By deduction we can draw the inference that *WaitSignal* s_3 cannot complete before *SendSignal* $s_2 \rightarrow c_2$ completes execution. Indirect dependency analysis and Φ merging is performed by the $\widetilde{\bigvee}$ function defined in the next section.

The Φ map at any point in the execution will contain all dependencies, both direct and indirect, between any two cores. Equivalent information can be obtained by starting at any point and tracing all arrows between the two cores of interest.

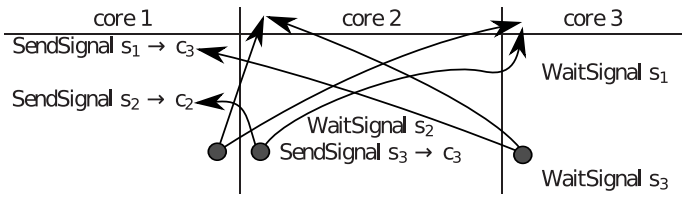


Figure 3.4: Φ map at instruction *SendSignal* $s_3 \rightarrow c_3$.

Figures 3.4 and 3.5 provide a look at two Φ maps at different locations in the execution of the program. Each core is marked with a dot and has dependency arrows going to all the other cores. Despite an AVOp executing on only one core, Φ is defined for all pairs of cores at that point. Additionally, the Φ map is initialized to -1 at the beginning of the program for all pairs of cores which signifies that there have been no dependency inducing AVOps yet.

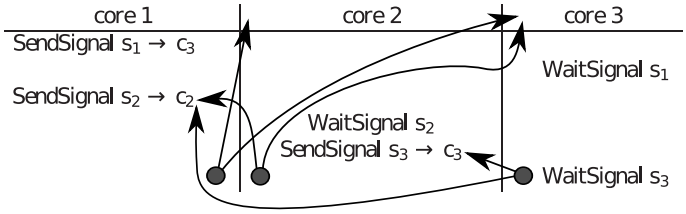


Figure 3.5: Φ map at instruction $WaitSignal\ s_3$.

Each *SendSignal* saves the Φ map at that point in the σ map and we make use of it when processing the corresponding $WaitSignal$. The Φ map from Figure 3.4 is used here to infer the dependency on core 1 from core 3. This exact computation occurs in the $\widetilde{\bigvee}$ function when it merges the Φ map stored by the SendSignal and the previous instruction's Φ map (in this simple case they happen to be the same Φ maps).

While a SendSignal stores the Φ map at that point in memory, a WaitSignal will look up this information and use it to strengthen dependency information when updating the Φ map.

3.7 Merging Φ Slices to Strengthen Our Partial Order

For the verification method, [AK08] define a function:

$$\widetilde{\bigvee} : (C \times N \times F) \times (C \times F) \to F$$

where F models a slice of Φ, that is, Φ restricted a specific index $n : N$. The function $\widetilde{\bigvee}$ merges slices of the partial order at corresponding SendSignal / WaitSignal AVOps that have an affect on ordering of WaitSignal AVOps on *other cores*. By doing this, we potentially strengthen our knowledge about $\Phi_{n_2}(c, c_{\text{wait}})$ for all c. Specifically, about the lastest AVOps known to complete on all other cores before the WaitSignal at index n_2 and core c_{wait} can complete.

To define $\widetilde{\bigvee}$ we make use of the map modification notation introduced in Section 2.1. One thing to notice is how the outermost modification (c_1, c_2) will override the inner modification (c', c_2). The left argument (subscript 1) is the information we had at the time of a SendSignal and the right argument (subscript 2) is the information at the time of the WaitSignal that is being currently processed. See the definition of WaitSignal in Section 6.3.

$$\widetilde{\bigvee}((c_1, n_1, \Phi_{n_1}), (c_2, \Phi_{n_2})) = \begin{matrix}\big[(c_1, c_2) \to \max\{\Phi_{n_2}(c_1, c_2), n_1\}\big](\\ \big[(c', c_2) \to \max\{\Phi_{n_2}(c', c_2), \Phi_{n_1}(c', c_1)\}\big](\\ \Phi_{n_2}))\end{matrix} \tag{3.3}$$

As a more complete example of the notation we use, the above definition is equivalent to

the one below using case notation.

$$\widetilde{\bigvee}((c_1, n_1, \Phi_{n_1}), (c_2, \Phi_{n_2}))(c', c'') = \begin{cases} \max\{\Phi_{n_2}(c', c''), n_1\} & c' = c_1, c'' = c_2 \\ \max\{\Phi_{n_2}(c', c''), \Phi_{n_1}(c', c_1)\} & c'' = c_2 \\ \Phi_{n_2}(c', c'') & \text{otherwise.} \end{cases}$$

Imagine the situation where we know that some core c is further along executing the program at index n_1 (on core c_1) than we do at the current instruction on core c_2. $\widetilde{\bigvee}$ will then incorporate this strengthened dependency information from Φ_{n_1}, incorporating it into the current slice Φ_{n_2} and producing the slice at Φ_{n_2+1}.

3.8 State

In addition to Φ_n, our method of verification constructs the following function inductively in the instruction index, n.

$$G = C \times S \to \begin{pmatrix} C \times N \times F \\ \oplus \\ N \end{pmatrix} \tag{3.4}$$

$$\sigma_n : G \tag{3.5}$$

where $\oplus$ means disjoint union (reviewed in Section 2.2).

To calculate X_{n+1}, where $X_n = (\sigma_n, \Phi_n)$, it is only necessary to have access to X_n, because we make copies of slices of Φ which are needed in future steps.

The disjoint unions can be understood in the following way for a given core c and signal s:

1. Receive pending (signal sent) — $\sigma(c, s)$ records the (core c', index n' and follows map slice Φ') on a SendSignal. Thus, signal s is expected to arrive to core c from core c'.
2. Signal cleared (signal received) — $\sigma(c, s)$ records the (index n) at which the signal was received by a WaitSignal. Thus, signal s was received on core c at instruction n.

We initialize all entries in σ_0 to -1 (all signals have last been received at index -1) at the start of program verification. Similarly, all entries in Φ_0 are initialized to -1 indicating that nothing is known about ordering.

3.9 Verification Step

Relevant sections quoted from [AK08] but extended for our purposes:

The maps σ_{n+1} and Φ_{n+1} are constructed from the maps and AVOp executing at index n according to the universal rule $\Phi_{n+1}(c, c) = (n)$, and specific rules depending on the executing AVOp:

SendSignal $c\ c'\ s'$

We need to verify that there is no current use of this signal in any program order. Divide into two cases according to the form of $\sigma_n(c', s')$, the first case excludes programs in which the signal is in use in the presentation order. This is not sufficient because the presentation order is not the only possible execution order, so we need to use the follows relation to check this.

If $\sigma_n(c', s') = (c'', n', \Phi')$, this is an error, because this signal s' may be sent by both cores (from c at the present instruction n and from c'' at n') at the same time.

If $\Phi_n(c', c) \geq \sigma_n(c', s')$, then we know that the last instruction to consume the signal s' on c' at $\sigma_n(c', s')$ completes before $\Phi_n(c', c)$ on c' which (by definition) occurs before the current instruction on c. (This includes the case that this signal has never been used, in which case $\sigma_n(c', s') = -1$.) Let $\Phi_{n+1} \coloneqq \Phi_n$, $\Delta_{n+1} \coloneqq \Delta_n$, and $\sigma_{n+1}(c', s') \coloneqq (c, n, \Phi_n)$, $\sigma_{n+1}(c'', s'') := \sigma_n(c'', s'')$ for other values of (c'', s'').

If the inequality is not satisfied, the program is not sound, because we do not have $n' \prec n$, meaning that this signal could arrive before the previous WaitSignal s' on c', which is the case of the bad example program.

WaitSignal $c\ s'$

If $\sigma_n(c, s') = (c'', n', \Phi')$, then we know that the signal s' will come from core c'', and we can use the fact that the wait completes after the send, to update the follows map. Let $\Phi_{n+1} \coloneqq \widetilde{\vee}((c'', n', \Phi'), (c, \Phi_n))$, $\Delta_{n+1} \coloneqq \Delta_n$, $\sigma_{n+1}(c, s') \coloneqq (n)$ otherwise $\sigma_{n+1}(c''', s''') \coloneqq \sigma_n(c''', s''')$.

If $\sigma_n(c, s') = (n')$, this is an error, or the program is not locally sequential, as required for this analysis.

Additional AVOps are covered in Section 7.1.

Theorem 3.9.1 *A locally sequential program is order independent* iff *the inductive verification described in this section terminates without error.*

The full proof of this theorem is presented in [AK08].

Chapter 4

The Loop AVOp

We introduce a loop AVOp to specify repetition in our DSL. This has a number of benefits. First, to reduce the communication overhead when transmitting AVOps to their corresponding cores. As the number of cores in the system increase, the program, which lists AVOps for all cores, becomes more and more unwieldy. Thus, loops provide a compression mechanism for the program (see Section 8.3). Second, the DSL becomes more expressive and it becomes easier to specify complex patterns of communication across many cores.

4.1 Loop AVOp Indexing

Within a program AVOps are assigned unique indices. The ordering between indices corresponds to the ordering of execution of the AVOps. Loops complicate the requirements for indices. We need to be able to refer to a specific AVOp within the body of the loop as well as a specific execution of that AVOp in some iteration of the loop. Nested loops also need to be supported. To achieve this the index type, N, needs to be extended with a case for loops.

$$N = \begin{pmatrix} Index \\ \oplus \\ Index \times Iteration \times N \end{pmatrix} \tag{4.1}$$

$$Index = \mathbb{Z} \tag{4.2}$$

$$Iteration = \mathbb{Z} \tag{4.3}$$

AVOps that are executed outside of any loop are numbered by their $Index$ as before. Once a loop is encountered, it is assigned an $Index$ just like any other AVOp and it uniquely identifies the loop. The $Iteration$ specifies which iteration of the loop the AVOp is executing. This recursive definition will nest once for each level of loop nesting. The the 3^{rd} AVOp in the 10^{th} iteration of a loop which itself is the 1^{st} AVOp of the program would have index $(0, 9, 2)$ (remember 0-based numbering). An example is presented in Figure 4.2. We define a lexicographic ordering of the indices. It corresponds to the ordering in

the presentation order.

$$i_1 < i_2 = \text{usual ordering of } i_1 \text{ and } i_2 \text{ in } \mathbb{Z} \tag{4.4}$$

$$i_1 < (i_2, j_2, n_2) = \begin{cases} true & \text{if } i_1 \leq i_2 \\ false & \text{otherwise} \end{cases} \tag{4.5}$$

$$(i_1, j_1, n_1) < i_2 = \begin{cases} true & \text{if } i_1 < i_2 \\ false & \text{otherwise} \end{cases} \tag{4.6}$$

$$(i_1, j_1, n_1) < (i_2, j_2, n_2) = \begin{cases} true & \text{if } i_1 < i_2 \\ true & \text{if } i_1 = i_2,\ j_1 < j_2 \\ true & \text{if } i_1 = i_2,\ j_1 = j_2,\ n_1 < n_2 \\ false & \text{otherwise} \end{cases} \tag{4.7}$$

$$\tag{4.8}$$

We define a helper function "ninr" which will retrieve the innermost loop index from an index. For example $\text{ninr}(0, 1, (2, 3, 4)) = (2, 3, 4)$ and not 4 which would eliminate the context of the innermost loop. In general this is useful for removing the context that a loop is in and focusing only on its component in the index. It is defined as follows:

$$\text{ninr}(i) = i \tag{4.9}$$

$$\text{ninr}(i, j, i') = (i, j, i') \tag{4.10}$$

$$\text{ninr}(i, j, n) = \text{ninr}(n) \tag{4.11}$$

We define a helper function to modify the *Iteration* component of the innermost loop index by x.

$$\text{nchg}_x(n) = \text{nchg}_x{}'(\text{ninr}(n)) \tag{4.12}$$

$$\text{nchg}_x{}'(i, j, n) = \text{ninr}(i, j + x, n) \tag{4.13}$$

Finally, we also define a function (denoted by $+1$) for getting the next successive AVOp in an AVOp stream.

$$n + 1 = \min_{\leq}\{n' : N | n' > n\} \tag{4.14}$$

4.2 Loop definition

Loops are introduced as a new AVOp: Loop k β. Where k is the number of iterations the loop should perform and β is the body of the loop, consisting of a sequence of instructions. Below is an example of a loop where $k = 2$ and $\beta = [(c_2, \text{WaitSignal } s), (c_1, \text{SendSignal} c_2\ s)]$. We refer to this loop as starting at index $l = 1$.

index	AVOp
0	SendSignal c_1 $s \rightarrow c_2$
1	Loop 2 [
$(1, *, 0)$	WaitSignal c_2 s,
$(1, *, 1)$	SendSignal c_1 $s \rightarrow c_2$
	]
2	...

Figure 4.1: Example using the Loop AVOp. The *Iteration* component of indices is specified by $*$ as it cannot be represent in the text of the program.

We present the same loop but with the AVOps unrolled as if they had been executed. Notice that this lets us fill in the *Iteration* component of the index.

index	AVOp
0	SendSignal c_1 $s \rightarrow c_2$
(1,0,0)	WaitSignal c_2 s
(1,0,1)	SendSignal c_1 $s \rightarrow c_2$
(1,1,0)	WaitSignal c_2 s
(1,1,1)	SendSignal c_1 $s \rightarrow c_2$
2	...

Figure 4.2: Example of an unrolled loop from Figure 4.1.

Definition 4.2.1 IterIxs — *We let* IterIxs(β, i) *denote the set of indices of all AVOps of* β *at the* i^{th} *iteration of the loop.*

In Figure 4.1, IterIxs$(\beta, 0) = \{(1, 0, 0), (1, 0, 1)\}$ and IterIxs$(\beta, 1) = \{(1, 1, 0), (1, 1, 1)\}$. We also assign an index to the loop itself. This is important when dealing with nested loops as IterIxs$(\beta, *)$ will give the outer most indices (it does not penetrate into an inner loop).

index	AVOp
0	SendSignal c_1 $s \rightarrow c_2$
1	Loop 10 $\beta_1 = [$
$(1, *, 0)$	WaitSignal c_2 s,
$(1, *, 1)$	Loop 20 $\beta_2 = [$
$(1, *, (1, *, 0))$	SendSignal c_2 $s \rightarrow c_3$,
$(1, *, (1, *, 1))$	WaitSignal c_3 s
	],
$(1, *, 2)$	SendSignal c_1 $s \rightarrow c_2$
	]
2	...

Figure 4.3: Example with nested loops. IterIxs$(\beta_1, *)$ treats the inner loop as a single AVOp so that IterIxs$(\beta_1, 5) = \{(1, 5, 0), (1, 5, 1), (1, 5, 2)\}$. The inner loop is assigned a single index of $(1, 5, 1)$ as if it was a single AVOp.

Chapter 5

Efficient Safety Verification of Looping Programs

If we were to naively extend the existing safety verification algorithm to support loops we would have to completely unroll the loop to verify the program. This would mean that verification would take time proportionally to the execution of the program which could be extremely large for a program running on a supercomputer or cluster. We are able to do much better by taking advantage of the features of our AVOp stream language.

We first extend the idea of locally sequential to programs with loops. A method of modifying one loop iteration into the next is needed. We call this bumping. Next we present the main result of safety verification, the loop short circuit theorem, which allows us to skip most loop iterations on loops with many iterations. We address safety verification of nested loops and present run time results.

5.1 Locally Sequential Loops

As a reminder, a stream of AVOps is *locally sequential* if every $(c_1, \mathsf{SendSignal}\ c_2\ s)$ is followed by a corresponding $(\mathsf{WaitSignal}\ \ c_2 s)$.

Definition 5.1.1 *An AVOp stream with loops is* locally sequential *iff the unrolled version of the stream is locally sequential.*

Lemma 5.1.2 *If a loop is unrolled twice and remains locally sequential, then it will be locally sequential when fully unrolled.*

Proof: We begin with a loop at index l.

If a loop is unrolled twice and is not locally sequential, it is clear that the fully unrolled version will not be locally sequential either.

If a twice-unrolled loop is locally sequential, for all c in C and s in S, $\sigma(c, s)$ can be in two states at the end of the second iteration.

$\sigma(c, s) = n$	We have most recently encountered a (c, WaitSignal s) AVOp. If $n < l$ then we did not encounter a WaitSignal in the body of the loop. Since the body was executed twice, there could also not have been a SendSignal$c$ s or we would have two SendSignals without a WaitSignal between them. This in turn would mean the twice-unrolled loop would not be locally sequential. Thus, the unrolled version of the loop will not change the state of $\sigma(c, s)$. If $n \geq l$ then we encounter a WaitSignal in the body of the loop. Since the body was executed twice, there must be a ($*$, SendSignal$c$ s) in the body of the loop. Otherwise, the first two executions of the loop would not have been locally sequential because of back to back WaitSignals. Subsequent iterations execute the same body and will also keep the sends and waits in balance. Since a WaitSignal was last encountered, the unrolled loop will terminate with σ set to a single integer as will the twice unrolled version. Thus, the unrolled version will also be locally sequential.
$\sigma(c, s) = (c', n', \Phi')$	We have most recently encountered a (c', SendSignal c s) AVOp. A similar argument to the one above applies.

5.2 Bumping State to the Next Iteration via Λ

We define a map Λ for specifying the transformation of one iteration's verification state into the next iteration's. It is defined in two equations 5.2 and 5.3 that are chosen based on the format of the argument. There is one equation for each item in the product of the type. Notice that these types correspond to the result types of Φ and σ.

$$\Lambda : \begin{pmatrix} N \to N \\ \times \\ C \times N \times F \to C \times N \times F \end{pmatrix} \tag{5.1}$$

In defining Λ_l we have:

- l is the index of the beginning of the loop.
- n is the index of the AVOp currently being verified.

An index is transformed the next iteration's corresponding index by incrementing the iteration component of the index. Notice that if an index is before the beginning of the loop we do not modify it.

Suppose we have Φ_n and want to transform it to the corresponding $\Phi_{\text{nchg}_1(n)}$ after 1 iteration. This transformation is given by $\Lambda_l \circ \Phi_n = \Phi_{\text{nchg}_1(n)}$ given that Φ_n is stabilized in the sense that any entry that is going to be updated during the execution of the loop has been updated.

$$\Lambda_l(n) = \begin{cases} n & \text{if } n < l \\ (i', j' + 1, n') & \text{otherwise, where } (i', j', n') = \text{ninr}(n) \end{cases} \tag{5.2}$$

Stored in σ are Φ slices which have to be updated. Once again we can decide whether a transformation is required based on whether the index was before the beginning of the loop. If the Φ slice was stored before the beginning of the loop (by a SendSignal) then we do not update it. The Φ slice is modified by composing it with Λ which effectively moves it one iteration over.

$$\Lambda_l(c, n, \Phi) = \begin{cases} (c, n, \Phi) & \text{if } n < l \\ (c, \Lambda_l(n), \Lambda_l \circ \Phi) & \text{otherwise} \end{cases} \tag{5.3}$$

5.3 Loop Short Circuit Theorem

We can verify loops by running the verifier on an unrolled version. However, this would take time proportional to the number of iterations and we can do better. If we notice that one iteration of the loop can be related to the previous iteration via Λ, then we can extrapolate the effects on Φ and σ to the last iteration of the loop.

Theorem 5.3.1 *Given Loop k β at index l, if* $\exists i : \mathbb{N}_0, i > 0, \forall n \in \text{IterIxs}(\beta, i)$,

$$\Phi_n = \Lambda_l \circ \Phi_{\text{nchg}_{-1}(n)}$$
$$\sigma_n = \Lambda_l \circ \sigma_{\text{nchg}_{-1}(n)}$$

Then $\forall n' \in \text{IterIxs}(\beta, k-1)$ *where* $(*, *, i) = \text{ninr}(n) = \text{ninr}(n')$,

$$\Phi_{n'} = {\Lambda_l}^{(k-1)-i} \circ \Phi_n$$
$$\sigma_{n'} = {\Lambda_l}^{(k-1)-i} \circ \sigma_n$$

Proof: Given Loop k β at index l, if $\exists i : \mathbb{N}_0, i > 0, \forall n \in \text{IterIxs}(\beta, i)$,

$$\Phi_n = \Lambda_l \circ \Phi_{\text{nchg}_{-1}(n)}$$
$$\sigma_n = \Lambda_l \circ \sigma_{\text{nchg}_{-1}(n)}$$

Suppose $\exists n \in \text{IterIxs}(\beta, i), \exists c, c' \in C : \Phi_{\text{nchg}_1(n)}(c, c') \neq \Lambda_l(\Phi_n(c, c'))$.

We assume that no hazards are detected while verifying AVOps up to and including n. If hazards are detected, we report the error at that point and stop the verification procedure. There are two cases to consider for Φ with no hazards detected:

$\Phi_n(c, c') < l$ There is no (c, SendSignal$c's$) or (c', WaitSignal s) instructions in β because Φ_n would have been updated since $i > 0$. If there are no such instructions, then $\Phi_{\text{nchg}_1(n)}$ will not be updated in any future iteration. Thus $\Phi_n(c, c') = \Phi_{\text{nchg}_1(n)}(c, c')$.

$\Phi_n(c, c') \geq l$ Then there must be instructions of both forms (c, SendSignal c' s) and (c', WaitSignal s) in β, otherwise Φ_n would not have been updated. Since $\Phi_{\text{nchg}(n)} \geq \Phi_n$ we have that $\Phi_{\text{nchg}_1(n)} \geq l$ as well.

Notice that both Φ_n and $\Phi_{\text{nchg}_1(n)}$ refer to the same instruction in the text of the loop since $(*, *, i) = \text{ninr}(n) = \text{ninr}(\text{nchg}_1(n))$.

However, $\Phi_n \neq \Lambda_l \circ \Phi_{\text{nchg}_1(n)}$ means that the preceding SendSignal is at a different instruction in the text of the loop across the two iterations. Since Φ_n and $\Phi_{\text{nchg}_1(n)}$ are defined over the same loop body β, the presentation order is locally sequential and no hazards came up, there is exactly one SendSignalin β which corresponds to this wait signal.

This is a contradiction.

Therefore, no such n can exist, and $\forall n \in \text{IterIxs}(\beta, i), \Phi_{\text{nchg}_1(n)} = \Lambda_l \circ \Phi_n$.

We sketch out the proof for σ. Suppose $\exists n \in \text{IterIxs}(\beta, i), \exists c \in C : \sigma_{\text{nchg}_1(n)}(c) \neq \Lambda_l(\sigma_n(c))$.

$\sigma_n(c) = n$	This case follows a similar argument to that for Φ since it does not have to worry about the Φ slices stored in σ.
$\sigma_n(c) = (c', n', \Phi')$ $n' < l$	This case follows a similar argument to case 1 for Φ as none of the arguments are modified by Λ since n' is before the loop index l.
$\sigma_n(c) = (c_1, n_1, \Phi_1)$ $n_1 \geq l$	Let $(c_2, n_2, \Phi_2) = \sigma_{\text{nchg}_1(n)}(c)$, then we have $c_1 = c_2$. If $c_1 \neq c_2$ then there would have to be a different SendSignal AVOp in the loop text to bind a different core. This is impossible since the loop is locally sequential, both AVOps correspond to the same AVOp in the loop text β and no hazard was found. Similarly, we have that $n_1 = \Lambda_l(n_2)$. Finally, we have that $\Phi_2 = \Lambda \circ \Phi_1$ as proved in the section on Φ.

Therefore, no such n can exist, and $\exists n \in \text{IterIxs}(\beta, i), \sigma_{\text{nchg}_1(n)} \neq \Lambda_l(\sigma_n)$.

By induction over both these cases we see that we can relate the state of Φ_n to the state of Φ by Λ at the corresponding instruction in the final iteration of the loop, as stated in the theorem. Likewise for relating the state of σ.

5.4 Verifying Nested Loops

When verifying a loop, any nested loops are treated as a single AVOp in the context of the outer loop. To treat an inner loop as a single AVOp we must run through the initial iterations of the inner loop checking for hazards and then apply the short-circuit theorem to compute Φ and σ after the inner loop completes. We then proceed to check the following AVOps in the outer loop with the updated Φ and σ maps.

Confusion may arise in our use of the ninr function in the definition of Λ. Remember that an inner loop's index contains information about all the loops that it is nested in. The potential problem is that we will get an index that is more deeply nested than we want when working on a loop which has nested loops in its body. Because we treat nested loops as single AVOps in the context of the outer loop, this problem does not actually materialize.

5.5 Verifier Run Time

Since we will always try to apply the Short Circuit Theorem (Theorem 5.3.1) when verifying loops, we are interested in how long it takes until the preconditions of the theorem are met.

Definition 5.5.1 *Stabilized — A loop is stabilized at iteration j if the preconditions for Theorem 5.3.1 are met at iteration j.*

Although it may initially appear that a constant number of iterations are required until the loop is stabilized there are some hidden dependency chains lurking in σ. Because old Φ-slices may be stored in σ and then used to update the information of future Φ entries information does not necessarily propagate right away. Instead, dependency information can travel from one core to another until it finally reaches its final destination.

Looking at Figure 5.1 we see an example with such a dependency chain set up. The example switches between strengthening Φ ($\widetilde{\bigvee}$ merges a Φ-slice stored in σ with the current Φ) and then storing the strengthened Φ-slice in σ. This process iterates until we hit the last core. It is clear that there is no *direct* dependency on c_0 from c_3. However, there is an indirect dependency, and it takes multiple iterations to propagate this information. It is important to notice that this delay is a one time event at the beginning of the loop. As the initial propagation is happening, with every iteration a new one is started. Thus, if the diagram showed one more iteration, we would see $\Phi(0,3) = (0,1,1)$ at the end of iteration 5.

Theorem 5.5.2 *The maximum number of loop iterations executed before the preconditions for Theorem 5.3.1 are met is $|C| + 1$.*

Proof: Given Loop k β at index l, suppose $\exists c', c : C$ such that $\Phi(c', c) < l$ after iteration $|C| - 1$ (iterations are zero based). First, there is no pair of AVOps SendSignal c' and WaitSignal c because if there were then by the second iteration $\Phi(c', c) > l$. If $\Phi(c', c) < l$ for all iterations, then it does not impact the preconditions. Suppose $\Phi(c', c) > l$ after some iterations.

Since there must be an indirect dependency, there must be some $c'' : C$ and $s : S$ such that there is a SendSignal c'' c s that will eventually store a Φ-slice such that $\Phi(c', c'') > l$ (and the corresponding WaitSignal c s). Notice that $c'' \neq c'$, otherwise it would be a direct dependency. This must happen after iteration $|C| - 2$ otherwise $\Phi(c', c) > l$ at iteration $|C| - 1$. That is, $\Phi(c', c'') < l$ at iteration $|C| - 2$.

Therefore, there must be some $c''' : C$ and $s' : S$ such that there is a SendSignal $c'''c''s'$ that will store $\Phi(c', c''') > l$ but not by iteration $|C| - 3$. Similarly, $c''' \neq c'' \neq c'$ or we would have $\Phi(c', c) > l$ at iteration $|C| - 2$.

In the worst case, we go through all the cores one by one until we reach c'''' at iteration 1 (zero based second iteration) that has the relevant AVOps but $\Phi(c', c'''') < l$. This is

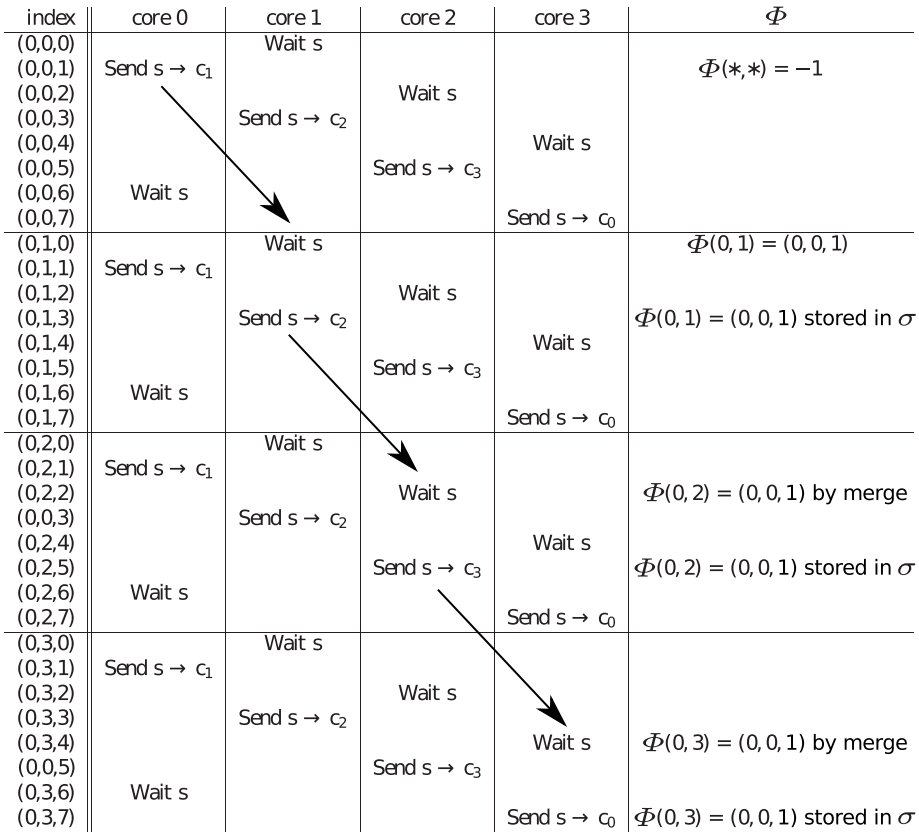


Figure 5.1: Non-rewritable loop example were Φ keeps changing across many iterations, requiring a variable number of iterations before stabilizing.

a direct contradiction since this is a direct dependency and by the second iteration all WaitSignal will be hit by their SendSignal or otherwise the program will not verify.

Now, since all entries of Φ that will be greater than l (initialized) are greater than l at iteration $|C| - 1$. All σ entries will contain initialized Φ-slices by iteration $|C|$ that is the $(|C| + 1)^{th}$ iteration. All other components of σ are updated directly from the previous iteration.

We also state an additional Lemma to make explicit the fact that it is safe to skip iterations via the Short Circuit Theorem (Theorem 5.3.1) after $|C| + 1$ iterations.

Lemma 5.5.3 *In order for the preconditions of Theorem 5.3.1 to be met, the program must verify correctly. Thus, if an error occurs it must occur in the first* $|C| + 1$ *iterations.*

Proof: This follows directly from Theorems 5.5 and 5.3.1.

Chapter 6

Rewritable Loops

Loops as presented so far are only slightly useful. The problem is that they repeat exactly the same action which restricts the communication patterns that are implementable. This is analogous to variables staying constant across all iterations of a loop in a conventional language. To make loops more expressive we introduce a rewrite which modifies which cores and signals are being referred to after each loop iteration.

This kind of AVOp modification scheme was initially inspired by the rotating register windows of the SPARC instruction set [WG94]. However, the register rotation mechanism of the IA-64 architecture [Cor10] is much closer in spirit to what we are trying to accomplish. Because we are not limited by hardware constraints, we allow a much more general expression of these ideas than implemented in either SPARC or IA-64. The main reason for such a scheme is to enable the amortization of latencies by working on independent data via software pipelining [Lam88].

We implement this by extending the looping AVOp to $\mathsf{Loop}\ k\ \rho\ \beta$ where ρ is a rewrite. We implement rewrites as permutations since we need a bijective map from one set (for example cores) onto itself. This restriction makes sure that we never reach a state where we collapse multiple cores onto one, loosing the ability to address some of the cores. It also ensures that we deterministically know which core is being referred to.

The map $\rho_C : C \to C$ maps cores to cores and is specified as a permutation. The rewrite system consists of two rewrite maps of exactly the same type. The first, $\hat{\rho}$, is the global rewrite that is applied to each AVOp whether or not it is within a loop and starts as the identity map at the start of the program. The second, ρ, is a loop rewrite that modifies the global rewrite after the end of each iteration. A global rewrite is required to allow consistent reasoning after the end of a loop.

6.1 Motivating example

We present an example of a loop with rewrites. The loop rewrite is (given first in cycle notation, then as an explicit map):

$$\rho_C = (0\ 1\ 2) = \begin{pmatrix} 0 \mapsto 1 \\ 1 \mapsto 2 \\ 2 \mapsto 0 \end{pmatrix}$$

This type of loop pattern could be used for buffered computation in which each SendSignal signifies that a condition is met, and each WaitSignal waits for that condition. In the meantime, as much work as possible is scheduled to ensure that by the time the WaitSignal is hit the condition has already been met. This introductory example presents the signaling pattern only.

index	AVOp	comment
0	SendSignal $c_0\ s \rightarrow c_2$	*prologue*
1	SendSignal $c_1\ s \rightarrow c_0$	*prologue*
2	Loop 4 $\rho_C = (0\ 1\ 2)\ \beta = [$	
$(2, *, 0)$	WaitSignal $c_2\ s$	
$(2, *, 1)$	SendSignal $c_2\ s \rightarrow c_1$,	
	]	
3	WaitSignal $c_2\ s$	*epilogue*
4	WaitSignal $c_0\ s$	*epilogue*

Notice that we need to execute the AVOps in the epilogue with a $\hat{\rho}$ which is the composition of all the rewrites encountered during the execution of the loop. For this reason we need to have a global $\hat{\rho}$ that will apply outside (after) the scope of a loop. The two WaitSignal AVOps immediately after the loop will always wait for the correct outstanding sent signals no matter how many iterations of the loop are executed. Unrolling the loop while applying the rewrite at each iteration would result in the following AVOp sequence:

index	core 0	core 1	core 2
0	SendSignal $s \rightarrow (c_2 = 2)$		
1		SendSignal $s \rightarrow (c_0 = 0)$	
$(2, 0, 1)$			WaitSignal s
$(2, 0, 0)$			SendSignal $s \rightarrow (c_1 = 1)$
$(2, 1, 1)$	WaitSignal s		
$(2, 1, 0)$	SendSignal $s \rightarrow (c_1 = 2)$		
$(2, 2, 1)$		WaitSignal s	
$(2, 2, 0)$		SendSignal $s \rightarrow (c_1 = 0)$	
$(2, 3, 1)$			WaitSignal s
$(2, 3, 0)$			SendSignal $s \rightarrow (c_1 = 1)$
3	WaitSignal s		
4		WaitSignal s	

In the above execution trace we see that c_1 stands for different cores based on the state of $\hat{\rho}$. This is because each AVOp implicitly applies $\hat{\rho}$ at each invocation.

6.2 The rewrite map: ρ

The rewrite map, $\hat{\rho}$, performs rewrites for each AVOp of the executing program. After each iteration of a loop is complete, $\hat{\rho}$ is modified by composing it with the loop's rewrite, ρ. Suppose we are given $\mathsf{Loop}\ k\ \rho\ \beta$ and that $\hat{\rho}_j$ denotes $\hat{\rho}$ at the beginning of iteration j, the modification performed is $\hat{\rho}_{j+1} \coloneqq \rho \circ \hat{\rho}_j$.

Both types of maps have the following type:

$$\rho : R \tag{6.1}$$
$$\hat{\rho} : R \tag{6.2}$$
$$R = C \times S \to C \times S \tag{6.3}$$

Although we present definitions in terms of ρ, equivalent ones are used for $\hat{\rho}$. In later definitions we use two projections directly. First, the projection relevant to rewriting signals. Second, the projection relevant to rewriting cores.

$$\rho_S : C \times S \to S \tag{6.4}$$
$$\rho_C : C \to C \tag{6.5}$$

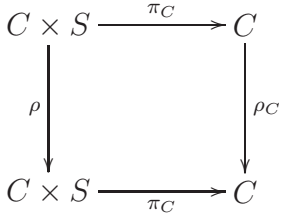
To define these two projected versions precisely, we first introduce the projections themselves. Each one selects the relevant item of the tuple.

$$\pi_C(c, s) = c \tag{6.6}$$
$$\pi_S(c, s) = s \tag{6.7}$$

Rewriting signals is defined as a direct projection.

$$\rho_S = \pi_S \circ \rho \tag{6.8}$$

Rewriting cores is more complicated. In defining ρ_C we make use of a commutative diagram that must be satisfied between ρ_C, ρ and π_C.



Notice that this places a restriction on ρ in that all entries for any given core must map to the same core.

$$\forall c \in C \quad \forall s, s' \in S \quad \pi_C(\rho(c, s)) = \pi_C(\rho(c, s')) \tag{6.9}$$

Then ρ_C is just ρ but with the domain and range restricted to C.

6.3 Formulation of AVOps as functions on the State

We redefine the effect of AVOps when updating Φ and σ here. This is an extension of the definitions laid out in Section 3.9 to take into account the global rewrite map $\hat{\rho}$. The AVOps are specified in such a way that one application of their interpretation as a function on (Φ_n, σ_n) corresponds to executing a single AVOp of that type.

$$(\hat{\rho}_{n+1}, (\Phi_{n+1}, \sigma_{n+1})) = (\mathsf{SendSignal}\ c\ c'\ s')(\hat{\rho}_n, (\Phi_n, \sigma_n))$$

We specify the type of each AVOp as a function on the Φ and σ maps:

$$\begin{aligned}
\mathsf{SendSignal} &: C \times C \times S \to (R, (F, G)) \to (R, (F, G)) \\
\mathsf{WaitSignal} &: C \times S \times N \to (R, (F, G)) \to (R, (F, G)) \\
\mathsf{Loop} &: \mathbb{N}_0 \times R \times [AVOp] \to (R, (F, G)) \to (R, (F, G))
\end{aligned}$$

If an error is detected then the verification algorithm stops immediately and reports the error. An error would be detected after $\hat{\rho}$ is applied which corresponds to the $\mathsf{SendSignal}'$ part of the definition. Let us define the $\mathsf{SendSignal}$ in the case of no errors. The definition consists of two parts. First, we apply ρ to rewrite the cores and signals appropriately. Second, we define the semantics of the AVOp as an equation.

$$\begin{aligned}
(\mathsf{SendSignal}\ c\ c'\ s')(\hat{\rho}, (\Phi, \sigma)) &= \big(\hat{\rho}, \big(\mathsf{SendSignal}'\ \hat{\rho}_C(c)\ \hat{\rho}_C(c')\ \hat{\rho}_S(c', s')\big)(\Phi, \sigma)\big) \qquad (6.10) \\
(\mathsf{SendSignal}'\ c\ c'\ s')(\Phi, \sigma) &= \big(\Phi, \big[(c', s') \to (c, n, \Phi)\big](\sigma)\big)
\end{aligned}$$

We present a similar definition for $\mathsf{WaitSignal}$. The index of the $\mathsf{WaitSignal}$ is n and is interpreted including the iteration of all loops it is nested in (even though this is an execution time property).

$$\begin{aligned}
(\mathsf{WaitSignal}\ c\ s\ n)(\hat{\rho}, (\Phi, \sigma)) &= \big(\hat{\rho}, \big(\mathsf{WaitSignal}'\ \hat{\rho}_C(c)\ \hat{\rho}_S(c, s)\ n\big)(\Phi, \sigma)\big) \qquad (6.11) \\
(\mathsf{WaitSignal}'\ c\ s\ n)(\Phi, \sigma) &= \left(\widetilde{\bigvee}(\sigma(c, s), (c, \Phi))\ ,\ \big[(c, s) \to n\big](\sigma)\right)
\end{aligned}$$

We define Loop inductively on the number of iterations performed by the loop with the base case at $k = 0$. Otherwise, we compose all the AVOps of the body of the loop and perform our rewrite at the end of an iteration.

$$(\mathsf{Loop}\ k\ \rho\ \beta)(\hat{\rho}, (\Phi, \sigma)) = \begin{cases} (\hat{\rho}, (\Phi, \sigma)) & \text{If } k \leq 0 \\ \big((\mathsf{Loop}\ (k-1)\ \rho\ \beta) \circ \mathrm{rwr}_\rho \circ \beta\big)(\hat{\rho}, (\Phi, \sigma)) & \text{Otherwise} \end{cases} \qquad (6.12)$$

Where we make use of the body of the loop, β, acting on the state. Here, β_i denotes the i^{th} AVOp of the loop body, thus:

$$\beta(\hat{\rho}, (\Phi, \sigma)) = (\beta_{|\beta-1|} \circ \cdots \circ \beta_1 \circ \beta_0)(\hat{\rho}, (\Phi, \sigma))$$

We also made use of the function rwr which applies the loop rewrite after the body of the loop is executed.

$$\mathrm{rwr}_\rho(\hat{\rho}, (\Phi, \sigma)) = (\rho \circ \hat{\rho}, (\Phi, \sigma))$$

6.4 Induced Rewrites and Rewriting AVOps

In the definitions of Section 6.3 it was clear that only the Loop AVOp had any affect on the global rewrite $\hat{\rho}$. We refer to this as inducing a rewrite and introduce the concept here formally.

Definition 6.4.1 *Induced Rewrite $\tilde{\rho}$ — The rewrite induced by an AVOp a, given $a(\hat{\rho}, (\Phi, \sigma)) = (\hat{\rho}', (\Phi', \sigma'))$, is a rewrite $\tilde{\rho}$ such that $\hat{\rho}' = \tilde{\rho} \circ \hat{\rho}$.*

We introduce the explicit wording of an induced rewrite because the full affect of a loop on the global rewrite is not equivalent to just the loop rewrite. Given a Loop k ρ β that has no nested loops, we have that $\tilde{\rho} = \rho^k$. That is, we perform the rewrite as many times as there are loop iterations and that is the full affect of the loop on $\hat{\rho}$. The rewrite induced by a loop with nested loops is more complicated.

Lemma 6.4.2 *Given Loop k ρ β, where β has m nested loops and the rewrite induced by the i^{th} nested loop is $\tilde{\rho}_i$ then the rewrite induced by Loop is:*

$$(\rho \circ (\tilde{\rho}_{m-1} \circ \tilde{\rho}_{m-2} \circ \cdots \circ \tilde{\rho}_0))^k$$

Although the affect of nested loops on $\hat{\rho}$ can easily be computed by Lemma 6.4 the nested loops also affect all AVOps following them in the body of the loop. To take this into account we introduce the concept of rewriting AVOps themselves.

Definition 6.4.3 *Rewriting an AVOp a by ρ — we define the affect of applying a rewrite on an AVOp. This changes the affect of the AVOp on the state maps Φ and σ but does not modify its affect on $\hat{\rho}$.*

$$\rho(a)(\hat{\rho}, (\Phi, \sigma)) = \big(\,\mathrm{fst}(a(\hat{\rho}, (\Phi, \sigma)))\ ,\ \ \mathrm{snd}(a(\rho \circ \hat{\rho}, (\Phi, \sigma)))\big)$$

Where we make use of the functions fst *and* snd *which return the corresponding element of a tuple, that is,* $\mathrm{fst}(\hat{\rho}, (\Phi, \sigma)) = \hat{\rho}$.

By taking both Lemma 6.4 and Definition 6.4 into account we can completely take into account the affects of a nested loop, both on following AVOps and on the rewrite induced by the outer loop.

index	AVOp
0	Loop k ρ β = [
$(0, *, 0)$	Loop k' ρ' β',
$(0, *, 1)$	$\tilde{\rho}$(SendSignal c_0 $s \to c_1$),
$(0, *, 2)$	$\tilde{\rho}$(WaitSignal c_1 s)
	]

Figure 6.1: The affects of an inner loop can be taken into account by rewriting following AVOps by the inner loop's induced rewrite $\tilde{\rho}$. The outer loop's induced rewrite can also be calculated by Lemma 6.4.

6.5 Short Circuit Theorem for Rewritable Loops

Verifying rewritable loops is quite complicated unless we first simplify to the case without rewrites. This can be done by unrolling the loop $|\rho|$ times where $|\rho|$ is the order of ρ (that is $\forall x.\rho^{|\rho|}(x) = x$).

Suppose we are given the following program with a rewritable loop and want to verify its correctness.

index	AVOp
0	SendSignal c_2 $s \to c_0$
1	Loop k $\rho_C = (0\ 1\ 2)$ $\beta = [$
$(1, *, 0)$	WaitSignal c_0 s
$(1, *, 1)$	SendSignal c_0 $s \to c_1$
	]

Presented in Figure 6.2 are the effects of unrolling this loop by $|\rho| = 3$ times. The main thing to notice is that there is a correspondence between indices $|\rho|$ apart instead of 1 iteration apart as was the case for loops without rewrites. In terms of the bumping function Λ we have ${\Lambda_l}^{|\rho|}$ instead of Λ_l.

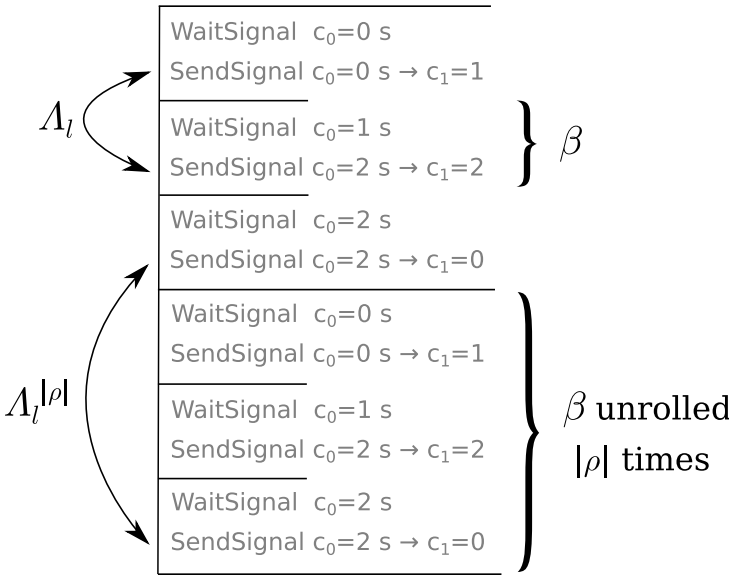


Figure 6.2: Rewritable Loop unrolled into a loop without a rewrite.

We proceed to state the Short Circuit Theorem for Rewritable Loops taking into account this difference in unrolling. Since we do not know that k is a multiple of $|\rho|$ we may also need to perform extra iterations of the loop at the end to compute the full effects of the loop on the state $(\hat{\rho}, (\Phi, \sigma))$.

Theorem 6.5.1

Given Loop k ρ β *at index* l, *if* $\exists i : \mathbb{N}_0, i > 0, \forall n \in \bigcup_{j=0}^{|\rho|-1} \text{IterIxs}(\beta, i|\rho| + j)$,

$$\Phi_n = {\Lambda_l}^{|\rho|} \circ \Phi_{\text{nchg}_{-|\rho|}(n)}$$
$$\sigma_n = {\Lambda_l}^{|\rho|} \circ \sigma_{\text{nchg}_{-|\rho|}(n)}$$

Then $\forall n' \in \text{IterIxs}(\beta, k-1)$, $\forall n'' \in \text{IterIxs}(\beta, i|\rho|)$ *where* $(*, *, i) = \text{ninr}(n'') = \text{ninr}(n')$, $X = \left\lfloor \frac{(k-1)-i|\rho|}{|\rho|} \right\rfloor$, *and* $x \equiv (k-1) \mod |\rho|$ *where* $0 \leq x < |\rho|$,

$$\Phi_{n'} = \rho^x(\beta) \circ \rho^{x-1}(\beta) \circ \cdots \circ \rho^0(\beta) \circ {\Lambda_l}^{X|\rho|} \circ \Phi_{n''}$$
$$\sigma_{n'} = \rho^x(\beta) \circ \rho^{x-1}(\beta) \circ \cdots \circ \rho^0(\beta) \circ {\Lambda_l}^{X|\rho|} \circ \sigma_{n''}$$

Proof: The proof is by reducing the rewritable loop to a non-rewritable loop and checking some remaining iterations.

For each AVOp a in β we apply the global rewrite $\hat{\rho}$ followed by $\tilde{\rho}_i$ for each of the the n nested loops preceding a in β. That is we convert a to a' by the following equation:

$$a' = (\tilde{\rho}_{n-1} \circ \tilde{\rho}_{n-2} \circ \cdots \circ \tilde{\rho}_0 \circ \hat{\rho})(a)$$

Once all the relevant rewrites have been applied we proceed to unroll $|\rho|$ times. Let m be the total number of nested loops in β. To calculate the AVOp a_j, corresponding to a' at iteration j we apply the following rewrite:

$$a_j = (\tilde{\rho}_{n-1} \circ \tilde{\rho}_1 \circ \cdots \circ \tilde{\rho}_0 \circ \rho \circ \tilde{\rho}_{m-1} \circ \cdots \circ \tilde{\rho}_{n+1} \circ \tilde{\rho}_n)^j(a')$$

Notice that we have to apply the induced rewrites of the nested loops following a before apply the loop rewrite ρ and then applying the induced rewrites preceding a. This is because composition of permutations is not a commutative operation.

At this point we may apply Theorem 5.3.1 to the unrolled loop body. We then proceed to bump as many iterations as possible but we need to take into account the unrolling. As shown in Figure 6.2, we must bump by ${\Lambda_l}^{|\rho|}$ when bumping from one iteration of an unrolled loop to the next.

After this, all that remains is to apply the remaining iterations of β since k is not always a multiple of $|\rho|$.

6.6 Verifier Run Time on Rewritable Loops

We examine the run time of the verification algorithm on rewritable loops. Some results from Section 5.5 are required.

Theorem 6.6.1 *The verification algorithm needs to examine at most* $|\rho| \cdot (|C| + 2)$ *iterations of a rewritable loop in order to verify correctness and compute the state after the loop.*

Proof: In order to verify a rewritable loop, it is unrolled $|\rho|$ times. After unrolling, it is equivalent to a non-rewritable loop, with a $|\rho|$-times bigger body, which requires $|C| + 1$ iterations before establishing the Short Circuit Theorem. After the short circuit is applied, there remain at most another $|\rho|$ iterations that must be checked without unrolling since the iteration count k is not always divisible by the unrolling factor $|\rho|$.

6.7 Memory requirements

In our treatment of Φ slices one may think that the memory requirement for a Φ slice is $\mathcal{O}(|\text{cores}|^2)$. However, in implementation, it is possible to store the slice specific to a given core. That is we fix the core $c : C$ that performed the SendSignal and for all other $c' : C$ we store $\Phi(c', c)$. This results in more efficient memory usage of $\mathcal{O}(|\text{cores}|)$.

In [AK08] the memory requirements for the verification algorithm were incorrectly stated as $\mathcal{O}(|\text{cores}|^2)$. This happened, most likely, because only the memory requirements of Φ were taken into account. However, it gets more complicated as Φ slices may themselves be stored in σ. In the worst case, σ will map all cores and signals to a Φ slice which results in the memory usage of $\mathcal{O}(|\text{cores}|^2 \cdot |\text{signals}|)$. Once we take all AVOps into account (see Section 7) we need to take δ into account which may also store Φ slices or DMA Tags. Thus, the final memory usage is $\mathcal{O}(C^2 \cdot \max(S, B) + C \cdot B \cdot D|)$ where:

- C — is the number of cores.
- S — is the number of signals per core.
- B — is the number of data buffers per core.
- D — is the number of DMA tags.

Chapter 7

Extension to Full AVOp Set

The AVOp language that we have covered so far is only a subset of the full language which deals directly with synchronization across cores. To be useful, we also need constructs for addressing main memory, sending data between cores and running kernel computations.

7.1 Extension of AVOp State and Hazard Checking

To keep track of data transfers we need to extend the state from $(\hat{\rho}, (\Phi, \sigma))$ to $(\hat{\rho}, (\Phi, \sigma, \Delta))$. Where Δ keeps track of the status of data transfers and makes use of the following types of entries:

- L — is the name of a data buffer, an area of memory local to a given core.
- D — is the set of DMA tags. These reference a specific DMA transfer operation so that we may be notified about transfer completion.

The type of Δ is as follows (where 2^D is the power set of D, and $\oplus$ denotes disjoint union):

$$\Delta_n : C \times L \to \begin{pmatrix} C \times N \times F \\ \oplus \\ N \\ \oplus \\ 2^D \\ \oplus \\ D \end{pmatrix} \tag{7.1}$$

Each part of the disjoint union of Δ may be interpreted as follows (we list the AVOps that modify entries Δ into this state):

1. An incoming transfer from core in C started at instruction in N with known partial order in F is expected (SendData – destination).

2. The data was last used at an instruction in N (WaitData, WaitDma, RunComputation).

3. The data buffer is the source of one or more DMA transfers whose list of (completion) tags which have *not yet* been consumed by a WaitDMA are in 2^D, and the data tag is in T (SendData – source, StoreMemory, WaitDma).

4. An incoming DMA with tag in D was started. Such a transfer can only come from main memory (LoadMemory).

For the initial Δ map, $\forall c \in C, \forall l \in L : \Delta_0(c, l) = -1$ (case 2).

We make use of definitions from [AK08] in this section, however, the data tag type T has been abstracted away in this presentation of the language. In practice, it is an extra boolean tacked onto the memory area of a buffer and is the last thing to be modified by a DMA. By repeatedly checking whether or not this flag is set, it is possible to determine whether or not the DMA has completed on the receiving core.

The non-rewrite part of the state $(\Phi_{n+1}, \sigma_{n+1}, \Delta_{n+1})$ at index $n+1$ is updated from the previous state at index n as follows by each of the following AVOps (including checking for hazards):

SendData $c\ l'\ \lambda\ c''\ l''\ d$

At the Source: We need to ensure that the source data is not about to be rewritten and that the use of d does not conflict with another use. If d appears in $\Delta_n(c, \tilde{l})$ for some $\tilde{l}$, then the DMA tag is already in use, and the completion of this DMA would be indistinguishable from the completion of another DMA already originated from this core, this is an error. If d occurs in $\Delta_n(c, \tilde{l})$ for some $\tilde{l}$, signal an error.

If $\Delta_n(c, l') = (n'')$ then no DMAs (SPU to SPU transfers or transfers from main memory) are pending for this buffer, so record the send by letting $\Delta_{n+1}(c, l') \coloneqq (\{d\})$ (with Δ_{n+1} equal to Δ_n at all other values).

If $\Delta_n(c, l') = (d')$, then on the sending core c an incoming DMA is in progress from main memory with DMA tag d'. Therefore it is not safe to start an outgoing DMA as this data could be overwritten while it is being sent, so this is an error.

If $\Delta_n(c, l') = (D')$ for some subset $D' \subset D$, then there are already outgoing DMAs in progress, which is OK if we do not try to reuse DMA tags.

Otherwise, record its use by $\Delta_{n+1}(c, l') \coloneqq (D' \cup \{d\})$, and copy the other maps from n to $n+1$.

Otherwise, this buffer is the target of a DMA from another core, which is an error.

At the Destination: Between cores, execution order does not have to follow presentation order. So we need to check that all use of the previous contents of the target buffer must have completed before the current instruction.

If $\Delta_n(c'', l'') = (n'')$ and $n'' \leq \Phi_n(c'', c)$, then the destination buffer has no pending DMAs after instruction n'' and the current instruction is known to complete $(\Phi_n(c'', c))$ after the last instruction (n'') to have used the target buffer, so it is safe

thus far. Because we assume an implementation of SendData and WaitData with a hardware based signal mechanism for each data tag, we cannot perform a SendData unless there is no incoming SendData transfer on the destination SPU with the same data tag (because signal usage cannot overlap). So if we have that for any l''', $\Delta_n(c'', l''') \coloneqq (c''', n''', \Phi_n)$ where $c''' \in C$ and $n''' \in N$, then we return an error because this data tag signal may not be consumed by the time this signal arrives.

Then we record that $\Delta_n(c'', l'') \coloneqq (c, n, \Phi_n)$. Otherwise, report the error that multiple incompatible DMAs with source and target (c'', l'') were detected, or the data tag is illegally reused.

WaitData $c\ l'$

If $\Delta_n(c, l') = (c'', n', \Phi')$, then we are waiting for this data, and have stored the follows map at the send instruction, which we can combine with the map at the current instruction: $\Phi_{n+1} \coloneqq \widetilde{\vee}((c'', n', \Phi'), (c, \Phi_n))$. Let $\sigma_{n+1} \coloneqq \sigma_n$, and $\Delta_{n+1} \coloneqq \Delta_n$ except $\Delta_{n+1}(c, l') \coloneqq (n)$, which indicates that this buffer is safe to use after the current instruction.

Otherwise, no incoming DMA from another core preceded this instruction, so the program is not locally sequential, or this DMA overlaps DMAs initiated locally, which is unsafe.

LoadMemory $c\ l'\ \lambda\ g'\ d$

Check that d does not appear in the image of $\Delta_n(\{c\} \times L)$, otherwise the uses of this tag overlap, which is an error.

If $\Delta_n(c, l') = (n')$, then the buffer has no pending IO, and we can initiate an incoming DMA by setting $\Delta_{n+1} \coloneqq \Delta_n$ except $\Delta_{n+1}(c, l') \coloneqq (d)$, $\sigma_{n+1} \coloneqq \sigma_n$ and $\Phi_{n+1} \coloneqq \Phi_n$.

Otherwise, the target buffer is the source or target for transfer which may not have completed, and this will produce an indeterminate result.

StoreMemory $c\ l'\ \lambda\ g'\ d$

Check that d does not appear in the image of $\Delta_n(\{c\} \times L)$, otherwise the uses of this tag overlap, which is an error.

If $\Delta_n(c, l') = (n')$, then no other IO is pending, and the store is safe. Set $\Delta_{n+1} \coloneqq \Delta_n$ except $\Delta_{n+1}(c, l') \coloneqq (\{d\})$ $\sigma_{n+1} \coloneqq \sigma_n$ and $\Phi_{n+1} \coloneqq \Phi_n$.

If $\Delta_n(c, l') = (D')$, other outgoing DMAs are pending, which is allowed. So set $\Delta_{n+1} \coloneqq \Delta_n$ except $\Delta_{n+1}(c, l') \coloneqq (D' \cup \{d\})$ (to record the new DMA tag we are waiting for), $\sigma_{n+1} \coloneqq \sigma_n$ and $\Phi_{n+1} \coloneqq \Phi_n$.

Otherwise, the source buffer is the target for a pending transfer, and this will produce an indeterminate result.

WaitDMA $c\ d$

The DMA tag could be associated with incoming or outgoing IO. In either case $\exists l$ such that $\Delta_n(c, l) = (d)$ (incoming), or $\Delta_n(c, l) = (D')$ and $d \in D'$ (outgoing), and l is unique. Otherwise, there is an error, because no DMAs are pending using this tag. This either indicates an unsound program or a non-locally sequential presentation.

If $\Delta_n(c, l') = (\{d\})$ then $\Delta_{n+1}(c, l') \coloneqq (n)$.

If $\Delta_n(c, l') = (D')$ then $\Delta_{n+1}(c, l') \coloneqq (D' \setminus \{d\})$

If $\Delta_n(c, l') = (d)$ then $\Delta_{n+1}(c, l') \coloneqq (n)$.

RunComputation c x $(l', l'', ...)$ $(\tilde{l}', \tilde{l}'', ...)$ $(p', p'', ...)$

Inputs: If for every $l \in \{l', l'', ...\}$, $\Delta_n(c, l) = (n')$ or (D') then $\Delta_{n+1}(c, l) \coloneqq (n)$ or (D') (respectively), otherwise, the buffer has pending incoming IO, which makes this computation unsound.

Outputs: If for every $\tilde{l} \in \{\tilde{l}', \tilde{l}'', ...\}$, $\Delta_n(c, \tilde{l}) = (n')$ then $\Delta_{n+1}(c, \tilde{l}) \coloneqq (n)$, otherwise, there is pending IO and this modification makes the program unsound.

7.2 Extension of the Rewrite Map

The rewrite maps need to be extended to take into account some of these new datatypes. The type of a rewrite map thus becomes:

$$\begin{aligned} &\rho : R \\ &\hat{\rho} : R \\ &R = C \times S \times L \times D \to C \times S \times L \times D \end{aligned}$$

We define two additional projected rewrite maps, on top of the extension of ρ_C and ρ_S from Section 6.2.

$$\begin{aligned} \rho_C &: C \to C \\ \rho_S &: C \times S \to S \\ \rho_L &: C \times L \to L \\ \rho_D &: C \times D \to D \end{aligned}$$

To define these two projected versions precisely, we first introduce the projections themselves. Each one selects the relevant item of the tuple.

$$\begin{aligned} \pi_C(c, s, l, d) &= c \\ \pi_S(c, s, l, d) &= s \\ \pi_L(c, s, l, d) &= l \\ \pi_D(c, s, l, d) &= d \end{aligned}$$

Because of the extension of the domain and codomain, the definition of all projected rewrite maps takes on the following form in a commutative diagram:

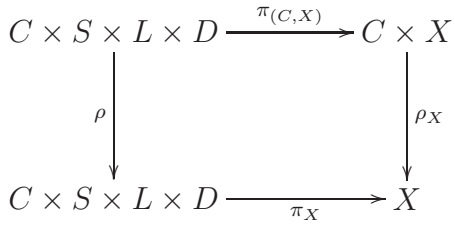


Figure 7.1: Defining diagram for projected rewrite ρ_X where X may be S, L or D. We made use of a helper function $\pi_{(C,X)}(y) = (\pi_C(y), \pi_X(y))$.

The commutative diagram in Figure 7.1 enforces the following equation for all $X = S$:

$$\forall c \in C \;\; \forall l, l' \in L \;\; \forall d, d' \in D \;\; \forall s \in S \;\; \pi_S(\rho(c, s, l, d)) = \pi_S(\rho(c, s, l', d'))$$

A similar equation holds for each of the other cases.

7.3 Extension of Loops for Accessing Global Memory

In addition to the changes required to the rewrite map of a loop, there is also a larger problem with how to handle global memory access. If the same locations in memory were to be accessed with every iteration of the loop, the set of useful programs that could be expressed would be limited. To this end we introduce a list of global memory buffers γ, where from each list we assign a memory location to every instance of a symbol g, in unrolled presentation order. To be precise, for a single global buffer g in γ we need $|\gamma(g)| \geq \kappa$ where κ is the number of times g is referenced in the unrolled program. Notice that κ can be much larger than the number of iteration counts of a single loop, especially with nested loops.

The global memory access list is further broken down into γ_r for memory locations that can only be read from and γ_w for memory locations that can be written to only once. This is a key property which allows arbitrary reads and no data hazards such as read-on-write to occur.

In an implementation, it would not be practical to have a list with one entry per execution of the loop body. More efficient approaches such as functional lazy lists or generators are envisioned instead.

In Figure 7.2 we present an example of computing the sum of two vectors and storing the result in a third vector.

It should be noted that all verification burden is on the user. She needs to prove before hand that all of the global addresses listed in any γ_w are used only once and are not listed in any γ_r. Among the issues that need to be addressed is the limitation of the buffer size parameter λ so that overlap checks can be performed (if an arbitrary and varying λ was allowed the user could not check that any given global location does not extend and overlap a latter global location).

index	AVOp
-	$\gamma_r(g_a) = [0, 1, 2, 3, 4]$
-	$\gamma_r(g_b) = [5, 6, 7, 8, 9]$
-	$\gamma_w(g_c) = [10, 11, 12, 13, 14]$
0	Loop 5 $\rho_C = (0\ 1\ 2\ 3\ 4)$ $\beta = [$
(0,*,0)	LoadMemory $c_0\ l\ \lambda\ g_a\ d$
(0,*,1)	LoadMemory $c_0\ l'\ \lambda\ g_b\ d'$
(0,*,2)	WaitDma $c_0\ d$
(0,*,3)	WaitDma $c_0\ d'$
(0,*,4)	RunComputation$_{VectAdd}$ $c_0\ (l, l')\ (l''')$
(0,*,5)	StoreMemory $c_0\ l'''\ \lambda\ g_c\ d'''$
(0,*,6)	WaitDma $c_0\ d'''$
-	]

Figure 7.2: Accessing global memory to perform a vector addition $A + B = C$ that is parallelized across 5 cores. The read-only global memory buffer lists γ_r and write-once buffer list γ_w are specified at the beginning, $\lambda = 1$ denotes buffer size, l denotes a location in local memory, d denotes a DMA tag.

7.4 Extension of Verification Algorithm

There are no further extensions necessary to be able to apply the Short Circuit Theorem in the presence of this extended AVOp set. Just as SendSignal and WaitSignal cause a dependency between cores so does SendData and WaitData. In fact, in many ways they are interchangeable.

Chapter 8

Future Work

We cover extensions of results presented in this thesis to stronger postulated results. Also covered are highly relevant, but different, research directions required to make our approach practical on large scale super computers.

8.1 Restricted Global Memory Access for Verification

We can extend γ_w to memory locations which can be read or written to from only a single core over the whole program. We make this single core restriction because the AVOps (thus the reads and writes) of a single core are totally ordered, hence none of the data hazards: read-after-write, write-after-read, write-after-write, can occur. However, we would have to verify that each memory location is only accessed from only that core. Allowing read and write access from arbitrary cores would introduce indirect communication via global memory which is inefficient due to synchronization overhead and difficult to verify.

Another possible approach would be to introduce barriers that seperate the program into phases with writing to a given global memory location being allowed only during a single phase. This write would then be necessarily ordered to finish before any further reads in the next phase.

Finally, the global memory access patterns themselves could be abstracted over in our AVOp language into an algebra of strided memory access. Access patterns could then be combined to generate more intricate patterns and so on while the verifier could inspect the access pattern to ensure correctness.

8.2 Stronger Short Circuit Theorem for Rewritable Loops

We postulate that the effect of executing the loop body on the state $(\hat{\rho}, (\Phi, \sigma))$ is equivalent to applying the rewrite and bumping to the next iteration.

$$\rho \circ \Lambda = \beta \tag{8.1}$$

In fact, it was during the analysis of a weaker (unrolled) property that we came up with the current formulation of Theorem 6.5.1. We can take advantage of the commutativity of Λ and ρ to unroll the loop body by the order of ρ.

$$\begin{aligned}
(\rho \circ \Lambda)^{|\rho|} &= \rho^{|\rho|-1}(\beta) \circ \cdots \circ \rho(\beta) \circ \beta && \text{Unrolling (8.1) } |\rho| \text{ times} \\
\rho^{|\rho|} \circ \Lambda^{|\rho|} &= \rho^{|\rho|-1}(\beta) \circ \cdots \circ \rho(\beta) \circ \beta && \text{By the commutativity of } \Lambda \text{ and } \rho \\
\Lambda^{|\rho|} &= \rho^{|\rho|-1}(\beta) \circ \cdots \circ \rho(\beta) \circ \beta && \text{Since } \rho^{|\rho|} \text{ is the identity}
\end{aligned}$$

Thus, we can see that by working with a loop body unrolled $|\rho|$ times the rewrite term is eliminated.

Using (8.1) would save work when verifying the later loop iterations. However, since we already need to check at least $2 * |\rho|$ iterations at the beginning of the loop to stabilize the loop, an additional $|\rho|$ iterations at the end, in the worst case, does not impact the asymptotic run time. For this reason we did not pursue this idea but practical implementations may be interested in this development to further optimize the actual run time and simplify the implementation.

8.3 Breaking up AVOp Streams to Increase AVOp Feed Rate

Some super computer architectures provide a tree structure to broadcast and potentially collection information back up the tree, for example the IBM Blue Gene super computer provides this type of communication pattern [TDD+02] for both broadcast and collection of information. One well known example of just the collection is the reduce part in the map/reduce method that powers the Google search engine [DGI04]. We are especially interested in the other direction of tree communication – broadcast.

This type of tree structured communication pattern is important in feeding potentially many thousands of cores with AVOps. Since the entire program, split across all cores, is presented as a single listing, this listing must then be forwarded to the relevant cores for computation. Having a single core feed all the other cores, the approach envisioned for the Cell/B.E. Processor, does not scale up to larger sizes. Many cores would be waiting for the main core, starved of AVOps to execute. What needs to be done is for the program to be progressively split into only the AVOps that are relevant to that subset of cores.

Loops are important for scaling because they allow computation across many cores to be highly compressed into a single loop body. Future work will have to examine efficient

methods of progressively breaking up the loops as the program is transferred down the tree. With each step down the tree there are fewer relevant cores which restricts the original program to the sub-program relevant to only those cores.

8.4 Scheduling

We have not addressed the issue of scheduling AVOps with loops for optimal performance. This is a key part of the out-of-order execution on CPUs that our approach mimics. The idea is to rearrange a program's AVOps to hide communication latency while preserving order Independence (that given the same memory contents at input locations, memory will be the same at output locations). Scheduling AVOp streams without loops is covered in [AK08].

8.5 Fault Tolerance

Hardware failure is unavoidable on supercomputers or large clusters due to the sheer amount of hardware involved. No fault tolerance results have been presented so far (by us or anyone else) for the AVOp language. This is an important area for future research in order to make the AVOp language applicable in real settings.

Chapter 9

Conclusion

We have demonstrated how to extended AVOp streams with rewritable loops. Such a construct enables us to express complex communication patterns and hide communication latencies in an approach similar to software pipelining of loops. Rewritable loops also allow the AVOp streams to be highly compressed. This has especially important implications for the future of distributing AVOp stream computations across a large collection of cores.

Furthermore, the verification procedure for AVOp streams was extended to the case with rewritable loops. The new verification procedure runs independently of the run time of the program being verified (for programs with loops repeating many times). This is a direct result of compressing the program text. Without loops, the program text is proportional to program execution. We need to check $\mathcal{O}(|\text{cores}| \cdot |\text{signals}| \cdot |\text{buffers}| \cdot |\text{tags}|)$ iterations of a rewritable loop body in order to verify it. In practice, this may be much better depending on the program text.

We have also refined the memory requirements needed for the verification algorithm to take into account two things. First, the fact that some state entities store other state entities and this needs to be taken into account for an accurate memory bound. Second, the fact that we can use a more restrictive version of some maps to minimize the memory used in the actual implementation. The memory bound for our verification procedure is $\mathcal{O}(|\text{cores}|^2 \cdot \max(|\text{signals}|, |\text{buffers}|) + |\text{cores}| \cdot |\text{buffers}| \cdot |\text{dma tags}|)$.

AVOp streams themselves only use non-blocking communication which allows our approach to scale to large systems and this approach is maintained with our extension to the language.

Bibliography

[AK08] Christopher K. Anand and Wolfram Kahl. Synthesising and verifying multi-core parallelism in categories of nested code graphs. In Michael Alexander and William Gardner, editors, *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[CIC] International Business Machines Corporation, Sony Computer Entertainment Incorporated, and Toshiba Corporation. *Cell Broadband Engine Programming Handbook*. IBM Systems and Technology Group, Hopewell Junction, NY, 1.0 edition.

[Cor10] Intel Corporation. Intel itanium architecture software developers manual, May 2010.

[DGI04] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[FG] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. pages 110–121. ACM.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI 88)*, July 1988.

[Maz] A. Mazurkiewicz. Trace theory. pages 279–324.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report*. Cambridge University Press, 2003. Also on `http://haskell.org/`.

[TDD+02] The Blue Gene/L Team, T Domany, Mb Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, J Gagliano, A Gara, R Garg, R Germain, Me Giampapa, B Gopalsamy, J Gunnels, B Rubin, A Ruehli, S Rus, Rk Sahoo, A Sanomiya, E Schenfeld, M Sharma, S Singh, P Song, V Srinivasan, Bd Steinmacher-burow, K Strauss, C Surovic, Tjc Ward, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae (IBM Server Group, K Dockser (IBM Microelectronics, and L Kissel)). An overview of the blue gene/l supercomputer, 2002.

[WG94] David L. Weaver and Tom Germond. The sparc architecture manual version 9, 1994.