|  |
| --- |
| ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI |
| Yarn  A Speculative Multithreading System |
| Algorithms |
|  |
| **Rémi Attab** |
| **7/31/2011** |

|  |
| --- |
|  |

Table of Contents

[List of Figures 3](#_Toc299902604)

[List of Code Samples 3](#_Toc299902605)

[1. Introduction 4](#_Toc299902606)

[2. libyarn 5](#_Toc299902607)

[2.1 yarn\_map 6](#_Toc299902608)

[2.1.1 Probe 7](#_Toc299902609)

[2.1.2 Resize 7](#_Toc299902610)

[2.2 yarn\_epoch 8](#_Toc299902611)

[2.2.1 Data Structure 9](#_Toc299902612)

[2.2.2 Next 10](#_Toc299902613)

[2.2.3 Rollback 10](#_Toc299902614)

[2.2.4 Commit 11](#_Toc299902615)

[2.2.5 Stop 12](#_Toc299902616)

[2.3 yarn\_dep 12](#_Toc299902617)

[2.3.1 Data Structures 12](#_Toc299902618)

[2.3.1 Load 13](#_Toc299902619)

[2.3.2 Store 14](#_Toc299902620)

[2.3.3 Commit 14](#_Toc299902621)

[2.3.4 Rollback 15](#_Toc299902622)

[3. yarnc 16](#_Toc299902623)

[3.1 Analysis Pass 17](#_Toc299902624)

[3.1.1 Value Dependency 17](#_Toc299902625)

[3.1.2 Pointer Dependency 18](#_Toc299902626)

[3.1.3 Invariants 18](#_Toc299902627)

[3.1.4 Instrumentation Points 18](#_Toc299902628)

[3.1.5 Array Values 19](#_Toc299902629)

[3.2 Transformation Pass 19](#_Toc299902630)

[3.2.1 Speculative Callback Function 20](#_Toc299902631)

[3.2.2 Original Function 22](#_Toc299902632)

[5. Bibliography 24](#_Toc299902633)

# List of Figures

[Figure i - Epoch States 9](#_Toc299902634)

[Figure ii - Loop Structure 16](#_Toc299902635)

[Figure iii - Callback Function Instrumentation 20](#_Toc299902636)

[Figure iv - Original Loop Instrumentation 23](#_Toc299902637)

# List of Code Samples

[Code i - Compare and Set Example 5](#_Toc299902638)

[Code ii - CAS-Loop Example 6](#_Toc299902639)

[Code iii - Load Value Original 21](#_Toc299902640)

[Code iv - Load Value Instrumented 21](#_Toc299902641)

[Code v - Store Value Original 21](#_Toc299902642)

[Code vi - Store Value Instrumented 21](#_Toc299902643)

[Code vii - Load Pointer Original 21](#_Toc299902644)

[Code viii - Load Pointer Instrumented 21](#_Toc299902645)

[Code ix - Store Pointer Original 21](#_Toc299902646)

[Code x - Store Pointer Instrumented 21](#_Toc299902647)

# 1. Introduction

This goal of this document is to give a general overview of the critical algorithms used within yarn. It should allow the reader to either reproduce or understand the inner workings of the yarn project without being able to read C and C++ source code. Our description will mainly focus on the data structures used and the important steps of each algorithm. We will also describe the key decisions and tradeoffs that were made along with a rationale that explains the logic behind each decision.

Because the algorithms used by the two main components of the project are of different nature, we will describe them separately of each other in section 2 and 3. Each section will begin with a brief set of definitions that will be used to describe the algorithm. When appropriate, we will also give an overview of the data structures that drive the algorithm. We will then split each component in their individual part in order to describe each algorithm individually.

Note that this document still assumes a basic knowledge of programming languages and multithreading as well as general computer architecture and compilers. This document also makes frequent references to the architectural document to avoid duplicating high level information across multiple documents.

# 2. libyarn

In this section we will describe the algorithms used by the libyarn runtime component of the yarn project. The algorithms described are all meant to be executed in parallel and must therefore be synchronized to preserve the coherence of the calculation.

Below we describe the terminology used in this section to describe the various algorithms.

**Word:** A word is an integer whose size is the register size of the target processor. For example, on a 32 bit processor, a word would be 32 bits while on a 64 bit processor, a word would be 64 bits. Since this type is used in all algorithms that require atomic operands, we must be careful about making any assumptions related to integer size.

**Lock-free Algorithm:** A lock-free algorithm is an algorithm that doesn't employ any locks to synchronize itself. The main advantage of this type of algorithm is that it avoids context switches that are required when contending for a lock. So each algorithm is constantly running and should also guarantee progress.

**Atomic Operand:** Lock-free algorithms are usually built using atomic operands in order to guarantee that an operation is done atomically without using locks.

**getAndIncrement:** Is an atomic operation that increments a given value and return its old value.

**Compare and Set (CAS):** Is an atomic operation that first checks if a value *X* is equal to the value *A*. If it is, then the value *X* will be set to the value *B* and cas will return the value *A*. If it is not equal then the value *X* will remain unchanged and the value *X* will be returned. Code i gives an example of its usage.

word x = 10;

...

if cas(x, 10, 100) == 10

print Success!

else

print Another thread interfered!

Code - Compare and Set Example

**CAS-Loop:** A cas-loop is used to modify a given value atomically while tolerating interference from other threads. Code ii shows the general format that a CAS-loop will take.

word x = 10;

...

loop

word old\_x = x

word new\_x = f(old\_x)

until cas(x, old\_x, new\_x) == old\_x

Code ii - CAS-Loop Example

We first begin by atomically reading the value of *x*. We then modify the read value using the function *f* and store the result in a new variable. We then attempt the write the modification back to *x* using a CAS instruction. If the write succeeded, we break out of the loop and execute the rest of the code. If the write failed, then we try again.

The main advantage of a CAS-loop is that it guarantees progress because if N > 1 threads are attempting to modify a given value, then 1 thread will always succeeded and N-1 thread will try again. A CAS-loop also allows us to make multiple modifications coherently on a single variable that is contended by multiple threads.

**Bit Packing:** The process of packing multiple integer values within a single word. For example, yarn\_dep uses two bit fields to keep track of the read and write fields. In order to modify these atomically, we must first pack them into a single word which can then be manipulated using atomic operands. Bit unpacking is the inverse of bit packing where we take a packed word and extract the various variables stored within.

## 2.1 yarn\_map

yarn\_map is concurrent open-addressing hash map implementation that uses linear probing. We use a custom implementation because we don't need to support concurrent element removal which greatly simplifies any potential concurrency schemes. It also allows us to obtain better performance while keeping the memory usage down.

The hash-map is composed of an array of nodes. Each node is composed of two elements:

* **key**  which is an address and acts as the key to the table.
* **value** which is a pointer to the user data.

We also keep track of the current capacity and size of the table in order to determine when resizes are required. The hash function implementation is taken from the excellent MurmurHash[[1]](#footnote-1) project.

### 2.1.1 Probe

The probe operation attempts to retrieve the value associated with a given key in the hash-table. If the given key is not present in the table it is added along with the provided value.

The probing algorithm will sequentially check every bucket in the hash table array starting at the hash of the key value until it either finds the desired key or an empty slot to place the key. There are three main contention points between the threads during the probing of the hash table.

The first contention point is when a thread reaches an empty bucket and needs to grab it for its key. Since multiple threads may be probing the same bucket at any given time, we must ensure that only one thread is able to grab the bucket. We do this by using a CAS instruction to convert the key attribute of the bucket from NULL to a value. If the CAS instruction succeeded then the thread successfully added the key to the hash table. If the CAS instruction failed, we have to re-attempt to read the bucket because the key that was added might be the one we are currently looking for.

Once the key is added, we must also set the user provided value into the table. To do this we decided to not use any synchronization primitives and just set the value directly after the key was set (we use a memory barrier to enforce this). Instead of synchronizing during the write on the bucket, we synchronize during the read of the bucket by way of a spinning primitive. Essentially we continuously read the value until it is non-null. We deem this acceptable since the chances of contentions at this point are relatively low and the simple spin primitives keeps the total latency of the operation down to a minimum.

Finally in order to do our capacity checks in order to determine whether to resize or not, we must update the size variable for the hash table. In order to do this atomically we use a get and increment atomic primitive. While this does introduce a shared contention point between all the thread, it hasn't proven to be performance bottleneck.

### 2.1.2 Resize

One of the downside of using a hash table is that it must be resized when its load-factor reaches a certain threshold. We decided early on that supporting concurrent resizes and probing operations would be simply too complex to implement and would not lead to a useful performance improvement since resize operations should be a rare occurrence. This means that when the hash table needs to be resized we block any new probing operations and wait for any ongoing probes to complete before attempting to resize the hash table.

When a probing operation is blocked because of an ongoing resize, we decided that it would be wasteful to have a thread simply wait. Instead we devised a scheme where we have one thread (the master) that systematically transfer the items of the entire table while the other threads (the helpers) waiting to probe the table will pick items to transfer at random. This scheme introduces a number of contention points between the master and helper threads which are discussed below. These contention points rely on a resize status variable which is initiated with the status *nothing*.

First of all, there's a contention point when two threads try attempts to become the master. This contention point is resolved using a CAS instruction where the first thread that is able to change the resize state to *preparing* becomes the default master. The loser of this contention point automatically becomes a helper thread.

The master then proceeds to setup the new array where the elements of the old table must be transferred to. It then waits for the user count variable to indicate that no threads are currently probing the table. This is done by spinning since the latency of a single probe operation should be fairly as small as possible. Once all the users have drained-out, the resizing process is initiated by setting the resize state to *resizing*. At this point the master goes through every element of the array and transfers them sequentially. Once all the items have been transferred, it must wait for the helpers to also complete in order to avoid allowing incoming probes on half transferred values. The wait is executed by first setting the resize status to *waiting* and by then spinning on a helper count variable which keeps track of the number of helpers. Once the helpers are done, the master cleans up the old table and sets the resize status to *nothing* which allows new probing operations to happen.

When transferring an item, there may be multiple threads attempting to transfer it. While the helper threads try to keep this kind of contention to a minimum, it is still a possibility and we must be able to handle it. Therefore, a thread must first indicate ownership of the item by using a CAS instruction to set the key of the item to NULL. If the CAS instruction fails, then another thread has gained ownership and we must pick a new value to transfer. If the CAS instruction succeeds, we then linearly probe the target hash table in order to transfer the key and value there. The probing scheme here is essentially the same as the one describe in section 2.1.1.

The first thing the helpers must do is increment the helper count variable and make sure that there is effectively an ongoing resize. If there are no ongoing resizes then it decrements the helpers count variable and returns. It then waits for the resize status to be *resizing* by spinning and will then proceed to transfer items until the status changes. The item transfer is executed by picking a random bucket that hasn't been process by the master thread. In order to avoid contending with the master thread, the range of available buckets starts a few elements ahead of the current element treated by the master thread. Once the helper is done resizing, it decrements the helper count and waits for the resize status to be *nothing.* Note that all the waiting operations executed by the helper are implemented so that a helper can show up at any point in the resizing process without negatively affecting it.

## 2.2 yarn\_epoch

yarn\_epoch is responsible for scheduling the execution of the epochs as well as tracking the status of the various epochs. The various status of the epochs as well as their transitions are detailed in Figure i which will be referred to often in the description of the various algorithms



Figure - Epoch States

### 2.2.1 Data Structure

The scheduler operates on a fixed size array whose size is a function of the size of a word. The size requirement is due to an implementation detail in yarn\_dep and is explained in greater detail in section 2.3. We use a strict array in order to avoid having to allocate memory during computation which greatly improves performances as well as data locality. It nevertheless restricts the number of active epochs that can be computed at any given moment.

Note that we handle overflows in the array by simply looping back to the beginning. This introduces tail-eating problems where we have to be careful of not assigning to active epochs to the same array index. In order to avoid these problems and to keep track of the assigned epochs, we use three cursors into the array:

* **first:** This cursor indicates the earliest epoch that was scheduled and essentially provides the lower bound of the active epochs.
* **next\_commit:** This cursor indicates the next epoch that needs to start committing.
* **next:** This cursor indicates the next epoch that needs to be scheduled. By ensuring that this cursor never overtakes the first cursor we can ensure that no tail-eating problems occur.

Each nodes of the epoch array contains a status and a task pointer. The task pointer is not used in this current version. The epoch statuses are the main contention points for the scheduler. Contention occurs because the rollback function can modify the status of all epochs being executed at any time. It is therefore important to always synchronize modifications of an epoch's status.

### 2.2.2 Next

The next function is used by speculative threads in order to get the next epoch that needs to be computed. This function is the most important bottleneck of the epoch functions because it is depended on by all the other epoch functions. It is also one of the most difficult to implement has been the source of many contention issues. Because of this, the current solution is temporary and it doesn't scale to a higher number of cores. In this section we will not describe the current solution but we will instead describe an older solution that also showed severe scaling issues but performed better with fewer cores.

The goal of the next function is to atomically change the status of the epoch at the next cursor and increment the next cursor. Since these two values can't be incremented atomically, we decided to simply avoid having concurrent calls to next and epoch by adding a read-lock around the next function and a write-lock around the rollback function. This allows any number of concurrent calls to next to happen while there are no calls to rollback. As soon as a rollback is initiated, calls to next are put on hold until the rollback is completed. We consider this a good trade-off because the lock-free scheme used to permit concurrent calls to next and rollback are complex. Rollback is also a costly and infrequent operation so the costs of having next calls put on hold are amortized in the long run. Finally, there is no real reason to continue allocating epochs by there's an ongoing rollback since the allocated epoch would usually lead to more rollbacks.

Next begins by taking ownership of the read lock. It then attempts to increment the next cursor using a CAS-Loop. To increment the cursor it must first ensure that it doesn't overtake the first cursor and that the next epoch is in a state that can be allocated. It must also ensure that no stop epochs have been set. If any of these cases are true then it must spin until all these conditions are false. If all the conditions have been met, it then attempts to increment the next cursor using the CAS instruction of the CAS-Loop. Note that if the first cursor is equals to stop epoch, then the next function immediately returns and indicates to the caller that the speculative execution is done.

Once an epoch as been allocated, the thread then has to take ownership by changing the status from *commit* or *rollback* to *executing.* This status change doesn't need to be synchronized because the next function is the only one that can move the epoch out of these two states. The next function then releases the lock and returns the epoch as well as its previous status to the caller.

### 2.2.3 Rollback

The rollback function is responsible of updating the status of all epochs that are affected by a rollback as well as updating the next cursor in order to indicate that the rolled-back epochs need to be re-allocated. This function is the source of the contention issues of the next function and therefore suffered from the same design complications. The solution described in this section will correspond to the one describe for the next function.

The rollback functions starts by taking ownership of the write lock which prevents any epoch from being allocated until we're done. It then iterates over every epoch starting at the first provided epoch and changes the statuses of these epochs according to the state diagram in Figure i. Since the affected epochs might be active, it is therefore important to make the status changes in a CAS-Loop. Note that if no modifications have to be made to an epoch's status then we simply break out of the CAS-Loop and continue with the next epoch.

Once all the epochs have been rolled-back, we use a CAS-Loop to reset the next cursor to the provided epoch. The CAS-Loop in this case is superfluous because the next function is the only other function that can modify this cursor and it is prevented from doing so because of the lock. We still keep it for safety reason and to prevent errors in future versions of the code.

Once the cursor has been updated, we release the lock which allows epochs from being allocated again.

The rollback functions also keep track of an extra data structure used by yarn\_dep to determine whether it is safe to read the buffer of a given epoch or not. The structure is a bit field that keeps track of every epoch that is currently rolled-back. The bits are set when an epoch is first rolled-back and cleared once the data of the epoch has been rolled-back.

### 2.2.4 Commit

Because epoch doesn't actually commit any data, the commit operation is done in two phases and uses two cursors to keep track of these. The first is the allocation of a function to be committed to a thread and the second is to mark the epoch as being committed. This two phase approach allows a great amount of flexibility in our scheduling because an epoch can be executed and committed by two different threads. A description of this behaviour is given in detail in section 3.4 of the architectural document.

The first commit phase begins by incrementing the next commit cursor in a CAS-Loop because there may be multiple concurrent ongoing calls to the first commit phase. Before increment the cursor it must first ensure that there's at least one epoch that is ready to be committed (see state diagram) and that, if the stop epoch has been set, no epochs are committed beyond the stop epoch. If any of these conditions aren't met, we indicate to the caller that no epochs are available for commits. Notes that since we can't read the next and next commit cursor atomically, it is therefore possible to get false positives (return indicating no available epochs while there is one available). This will not prevent all the epochs from being committed (for a detailed proof, see section 3.4.3 of the architectural document). Once the next commit cursor has been incremented, we return the epoch to the user along with the task pointer. After this phase, the thread is said to own the epoch and no other thread can affect the epoch.

The second phase of the commit takes place after all the data for the epoch has been committed to memory. Our goal here is to update the status of the epoch and increment the first cursor. The update of the epoch can happen directly without CAS instruction because the thread owns the epoch. The increment of the first cursor is a bit tricky because we shouldn't increase it past an epoch that hasn't finished committing. We must therefore be able to increment it multiple times or not at all. It can be a source of contention so we must synchronize its execution using a CAS instruction placed within a loop. This doesn't mean that it is executed as a CAS-Loop because before executing the CAS instruction we make sure that the first cursor doesn't overtake the next commit cursor or any epochs that aren't in the *commit* state. If these conditions aren't met, then we break out of the loop. This means that whether the CAS instruction succeeded or not, we simply try again but with the new first cursor.

### 2.2.5 Stop

This section describes how we determine when to stop the computation. When a thread detects that the computation is completed it calls the stop function which updates the stop state to keep it coherent with another thread that may also have detected a stop point.

When calling the stop function, the stop epoch is updated using a CAS-Loop but only if no other stop epoch has been set or if the set stop epoch is greater than the current epoch. This ensures that we break out of the speculative loop as soon as possible. Setting a stop epoch will also cause the next function to stop allocating new epoch because all the epochs required to finish the computation have already been allocated so there's no need to continue computation. If the stop epoch is to be rolled-back then there's still no need to allocate new epochs because they would be after the stop epoch and would therefore be rolled-back as well.

During a rollback, if the stop epoch is one of the epochs to be rolled-back, the stop epoch is unset using a CAS-Loop. This is correct because if any other stop epochs have been detected then they were greater to the current stop epoch and are also to be rolled-back. Un-setting the stop epoch can be done at any point during the rollback as long as it is currently holding the write lock. It will also allow next to continue allocating new epochs.

## 2.3 yarn\_dep

yarn\_dep is the component responsible to tracking all the loads and stores made to memory locations. It also triggers rollbacks when a dependency violation is detected and commits speculative epochs back to memory.

Note that this component uses many bit manipulation techniques in order to speed-up several operations. Both the log2 and the trailling\_zeros functions were taken from the Stanford bit twiddling hacks[[2]](#footnote-2) page and are used to find the most significant and least significant bit set in a word.

### 2.3.1 Data Structures

yarn\_dep uses the hash map described in 2.1 to store it's dependency tracking node for a given memory address. The node contains a word value which is composed of two packed bit fields of equal size used to keep track of the reads and writes per epoch. These bit fields are responsible for the epoch size limitation because we must provide one flag per epoch and since we must be able to pack both bit fields with a single word. This effectively limits our maximum epochs to the size of a word divided by two (16 epochs on 32 bit processors and 32 epochs on 64 bit processors). The read and write flags are represented as bit fields because they can be more easily and rapidly manipulated. It also allows us to pack them in a single word which makes it possible to atomically read and write them. This is essential in order to provide a lock-free algorithm to the load and store functions.

The node contains a write buffer for each epoch that contains the values to be committed. The buffer should only be read if its corresponding write flag has been set. Otherwise it is considered to hold garbage. The node also contains a word which represents the last committed epoch for this dependency. This is essential to keep the commits done to epochs coherent and to ensure that our lock-free loading algorithms are coherent with the commits.

Because the hash map requires that a value is provided when probing and since the nodes are dynamically allocated, we employ a simple thread-safe memory pool to do our allocations. If the value provided to the hash map was not required then the memory pool simply remembers that value which will avoid an extra allocation on the next hash map probing.

We also provide an extra fast index that allows us to skip the hash map if the value has already been seen for a known index. Note that we still must add the indexed values to the hash map when they are first used in order to be able to detect any modifications made by an unknown pointer that aliases the memory location.

We also keep a linked list of all the dependencies associated with an epoch. This allows us to quickly process all the dependencies of an epoch during a rollback or commit operations at the cost of a non-trivial amount of memory. This trade-off may not be worthwhile because the extra memory may needlessly pollute the cache and increase the number of page faults. More testing is required to fully understand the effects. Note that removing this linked list would require that we augment the hash map in order to be able to iterate over all the elements within the hash table. This may not be feasible to do in a coherent manner.

### 2.3.1 Load

The load function is responsible for loading the value of a provided address for a given epoch. In order to minimize the chance of rollbacks, we implement Read and Write dependencies by looking for any earlier writes in the buffer and load that value instead.

The load function begins by obtaining the node for the dependency from the index or the hash table. It then atomically read the flags and set the read flag of the corresponding epoch. The read flags are then unpacked and kept for later use. We then obtain the rollback flags as well as the first cursor from epoch. The first cursor along with the current epoch forms the range of epoch where we want to look for a write flag. To find the write flags we first apply a mask formed by the range previously described and the rollback flags. If the resulting bit field is null we atomically load the value directly from memory because no write occurred on the address before or on our current epoch. If the mask value is non-null we obtain the most recent epoch by using the log2 utility. We then atomically load the buffered value of this epoch. Before returning to the user, we check the last committed epoch for this address. If it is greater than the loaded epoch, this means that the value we read was committed and could have been overwritten by another store operation. We must therefore discard the loaded value and atomically load the committed value directly from memory. Note that if the value read from memory is from an epoch older than the current epoch, then our current epoch is in a rollback state and the incoherent read doesn't matter.

### 2.3.2 Store

The store function is responsible for setting the write flag on the current epoch and check whether any are any more recent read flags for the address. If there are, then store will trigger a rollback on that epoch.

The store function begins by obtaining the node for the dependency from the index or the hash table. It then atomically stores the new value into the write buffer for this epoch. It is correct to do this before reading or setting the flags because if the write flag is already set then any premature reads be detected as a violation later in the algorithm. We then atomically read the flags and set the write flag for the appropriate epoch using a CAS-Loop.

The read flags are then examined for any dependency violation. To do this we first obtain the next cursor from epoch in order to form a search range for the read flags. We then mask the read flags with the range obtained previously and the rollback flags. If resulting bit field is null then no dependency violations are detected. Otherwise we use the trailling\_zeros utility in order to obtain the earliest epoch. We use that epoch to trigger a rollback in epoch module. Note that no data is actually rolled-back here because our load and store algorithm were not designed to support concurrent rollbacks call. We prefer to delay the data rollback until the epoch is next allocated. This ensures that only a single thread owns the epoch which negates any need for synchronization.

### 2.3.3 Commit

The commit operation is responsible for committing on the latest dependency buffer to memory which means that it uses the last commit word within the node in order to keep the commits consistent. Unfortunately, because we can't manipulate this word and commit the write buffer to memory, we have no choice but to lock the dependency while performing a commit. Note that this lock is only required to prevent concurrent commits on the same address and it is not required in the load and store functions. This means that even if we hold the lock we must still ensure that our modifications to the last commit word are done atomically and coherently.

The commit function traverses all the dependencies indexed by the epoch linked-list. For each dependency in list, it first locks the dependency and then checks whether the write flag has been set. If it has, it stores the write buffer into memory and sets the last commit value to the current epoch. It finally clears both the read and write flag for that epoch before releasing the lock. Note that the order of these operations is important and enforced using memory barriers because we must remain coherent with any concurrent load function calls. The read flag can be safely cleared at any time because if we are committing the epoch then it can no longer be implicated in any dependency violations.

### 2.3.4 Rollback

The rollback operation is responsible for only clearing the flags of each dependencies associated with the given epoch. Since the flags can be cleared atomically, no synchronization is required.

The rollback functions traverses all the dependencies indexed by the epoch linked-list. For each dependency in the list, it atomically clears the flag. That's it.

# 3. yarnc

In this section we will describe the yarnc compilation tool which is implemented as two optimizations passes within the LLVM compiler. For more details about the passes, see section 4 of the architectural document. The algorithms described in this section are mostly about data-flow analysis which is used to determine how values are used within a section of code.

Note that the IR representation used by the LLVM compiler enforces an SSA representation (see below for a definition). We will therefore assume in this section that all code manipulated is in the SSA form. Furthermore, we will also assume in this section that all the code manipulated is in the LCSSA form (see below for a definition). The "why" is described in section 4 of the architectural document.

Note that this section will also frequently refer to the basic composition of a loop which is illustrated in Figure ii. See the architectural document for a thorough description of this structure.



Figure - Loop Structure

Below we describe the terminology used in this section to describe the various algorithms.

**Live Value:** A value is live over a section of code if it is either read or written within that section. For example, a variable is live within a loop if it is either written to or read from within the basic blocks that constitute the loop.

**Loop-Carried Dependency:** Is a dependency that is live for multiple iterations of the loops. In this document, it will be represented as a variable that was live before or after the loop and is either read or written to within the loop.

**Loop Invariants:** Is a value live within a loop that is never written to.

**Three Address Code (TAC):** A simplified code representation used by compilers to do data-flow analysis. A TAC instruction is composed of 3 operands or addresses: The first operand receives the result of the operator applied to the last two operands.

**Single Static Assignment (SSA) form:** Is a more strict form of TAC code representation which only allows variables to be written to once. A code in this form is more easily analyzed since we can easily create a use graph of each variable. Note that there is a proof [[1](#01)] which states that any TAC code can be transformed into SSA code using *PHI* nodes which are described below.

**Loop Closed Single Static Assignment (LCSSA) form:** A stricter form of SSA code which states that immediately after any loop, there must be a PHI node for every value live within the loop that is also live after the loop.

**Basic Block:** Basic blocks are used to represent the general control flow of a given function. Each basic block is composed of a sequence of instruction that ends with a single terminator instruction. Note that terminator instructions can only appear at the end of a basic block.

**Terminator Instruction:** Any instruction that affects the flow of the program. This is usually composed of but not limited to any types of branching or return instructions.

**PHI (ɸ) node:** A PHI node is used to aggregate two or more values that belong to different control flow branches within a single value. A PHI node can only appear at the beginning of a basic block. They are necessary because otherwise it would be impossible to represent branches in the SSA form.

## 3.1 Analysis Pass

The analysis pass is responsible of gathering the information required about a loop in order to instrument it. It must also verify the loop to ensure that it is compatible with libyarn. Note that since the architectural documents already gives a good overview of most the concepts of this section, we will instead focus on how to identify and construct the various analysis data.

### 3.1.1 Value Dependency

A value dependency is a loop-carried dependency. See the section 4.3.1.1 of the architectural document for a more detailed. Value dependencies are primarily detected using the PHI nodes in the loop header and exit block or successor.

We detect entry dependencies using the PHI nodes in the loop's header. Each of these PHI nodes should have exactly two incoming values: one from the predecessor and one from the latch. The value coming from the predecessor is marked as the entry value. The value coming from the latch is marked as the end iteration value. The PHI node itself is marked as the start iteration value.

We detect exit dependencies using the LCSSA PHI nodes in the loop's footer. This node may have multiple incoming values which are all marked as being exiting values. If one of these exiting values corresponds to one of the entry dependencies' end iteration value, then both these dependencies are linked together and will be treated as one. The PHI node itself is marked as the exit value.

### 3.1.2 Pointer Dependency

A pointer dependency is any memory location that is accessed using a pointer. See the section 4.3.1.2 of the architectural document for a more detailed description. Pointer dependencies are detected using all the store and load instructions present within the loop.

We begin by processing every store instructions. If the pointer operand of the store instruction is a strict alias of another store instruction's pointer operand, then these two store instructions are linked together and will be treated as one. Otherwise the store instruction is added to the list of pointer dependencies.

We then process all the load instructions. If the pointer operand of the load instruction is a strict alias of another instruction's pointer operand, then these two instructions are linked together and will be treated as one. If the pointer operand of the load instruction can alias another instruction's pointer operand, then load instruction is added to the list of pointer dependencies. If the pointer operand of the load instruction is a loop invariant, then it is added to the list of loop invariants. Note that any duplicates in the invariants list are automatically discarded.

### 3.1.3 Invariants

Loop invariants that are live before the loop are important to detect because we need to copy over their values for use in the speculative callback function. See the section 4.3.1.3 of the architectural document for a more detailed description.

To detect invariants, we iterate through the operands of every instructions within the loop. If the operand is an instruction or an argument and if it is an loop invariant, then it is added to the list of invariants.

### 3.1.4 Instrumentation Points

The Instrumentation point lists are constructed during analysis in order to simplify the transformation pass. See section 4.3.2 for a more detailed description of instrumentation points. We construct a separate list for the pointers and the value dependencies because the instrumentation process for both of these is significantly different.

The construction of the value dependency instrumentation point relies on two utility functions call *findLoadPos* and *findStorePos*. *findLoadPos* is responsible for using the dominator tree of the loop to get the nearest common dominator block. This gives us the point in the basic block where we can put a load that will dominate all other loads of that value. Note that if the basic block found contains one or load of the value, then the position returned is the instruction itself. *findStorePos* is essentially the same but uses the post-dominator tree in order to find the position that is dominated by all the store instructions. This ensures that our store will happen after every other possible store of the value. Note that because in SSA form a value can only be stored to once, we use look at the incoming values of the PHI nodes in order to find all the possible store operations.

The instrumentation points for each value is built by passing the all the users of each start iteration values to the *findLoadPos* utility. A load instrumentation point is created using the returned position and the start iteration value. We then pass each end iteration values and each exiting values to the *findStorePos* utility. A store instrumentation point is created using the returned position and the value.

For pointer, an instrumentation point is created for every user of the pointers previously discovered. If the instrumentation isn't within the loop, it is discarded. If the target user is a store instruction then a store instrumentation point is created. If the target user is a load instruction then a load instrumentation point is created. Note that we can't regroup pointer instrumentation points like we do for value instrumentation points because each pointer may alias any of the other pointers.

### 3.1.5 Array Values

Array value list aggregate all the values that need to be passed to the speculative callback function in order to properly operate. See section 4.3.3 for a more detailed description. Each element of this list also includes a mark which indicates this value may be modified by the loop.

The list is simply an aggregation of the value and pointer dependency as well as the invariants. Only the value dependencies will be marked as being modified. Note that only the memory location pointed to by the pointers are modified. The pointers remain untouched and don't need to receive the mark.

## 3.2 Transformation Pass

This section describes the process used to instrument a loop that was identified and analyzed by the analysis pass. The process is split into two sections: the creation of the speculative callback function and the instrumentation of the original function. It is important to only modify the original function after we have finished creating the speculative callback function because otherwise we risk invalidating our analysis data.

Note that when generating instructions, it is usually important to perform a number of cast operations on values before using them as operands. The addition of these cast instructions have been elided from this document in order to simplify the algorithm descriptions.

### 3.2.1 Speculative Callback Function

The goal of this phase is to make a copy of the loop into its own callback function and apply the instrumentation points on its body. Since there's no safe way to copy only a group of basic block into their own functions, we decided to instead copy the entire function and prune the excessive blocks once we're done instrumenting. Unfortunately, when copying a function, it is not possible to change its type. This means that we can't adjust its arguments and return value to make it match our runtime library requirements. To get around this problem, we first clone the entire function without adding it to the module. We then make all our instrumentations on this temporary function and once we're done we copy its body into our final function. The result of the operations is illustrated in Figure iii.



Figure - Callback Function Instrumentation

Once the first clone of the original function has been made, we create a new instrumentation header that is placed before the header of the loop. We then create two placeholder values which represents the arguments of the final function. One of these arguments is the task pointer which contains all the values required to execute our loop. In order to use these values, we must first calculate a pointer to these memory locations. If the array value associated with the entry was marked as not being modified during the loop, we immediately load the value and replace every usage of the target value within the loop by the loaded value. We then allocate a buffer word that will be used as temporary storage for the instrumentation point.

To instrument the body of the function, we iterate through each of the instrumentation points generated during analysis and effect the modifications described in the following code samples.

%3 = %1 op %2

Code - Load Value Original

yarn\_load(%b from %p1)

%1 = load %b

...

%3 = %1 op %2

Code - Load Value Instrumented

%1 = %2 op %3

Code - Store Value Original

%1 = %2 op %3

...

store %1 into %b

yarn\_store(%b into %p1)

Code - Store Value Instrumented

%1 = load %2

Code - Load Pointer Original

yarn\_load(%b from %2)

%1 = load %b

Code - Load Pointer Instrumented

store %1 into %2

Code - Store Pointer Original

store %1 into %b

yarn\_store(%b into %2)

Code - Store Pointer Instrumented

There are few things worth nothing about these transformations:

* The value instrumentations can take place away from the instructions that are being instrumented. The pointer instrumentations must always take place directly before or after the load/store instruction.
* The %b value is the buffer that was allocated in the header. We need to use it because the yarn runtime function requires that the result of the load/store a memory location. The buffer provides this temporary memory address.
* The %p1 variable represents the pointer for a given array value that was calculated in the header.

After the body has been instrumented, our goal is to transform the loop into the execution of a single iteration. We do this by creating two new blocks that are placed after the loop latch. These two blocks contain a single return instruction which indicates to the caller whether the loop execution should be stopped or whether we should continue executing the next iteration. We then look for every edge leading into the header and redirect them to the continue block. We also look for every edge leading to the successor block and redirect them to the break block. We must also delete the phi nodes from the loop header because they are simply not needed anymore and they create usage dependencies with the predecessor blocks that we want to delete.

At this point, every value in the predecessor block should no longer have any users within the loop and there should also no longer be any edges from within the loop that leads into the successor block. We can therefore delete the successor block and the predecessor block. Unfortunately, because instructions within the predecessor block may still have uses within the successor block and because the successor blocks may have edges leading into it from the predecessor block, we can't just delete the blocks. In order to properly delete these blocks we must first delete every instruction starting from the last instruction to the first instruction of the function while ignoring everything that is in the loop or the instrumentation header. This ensures that every uses of an instruction have been deleted before we delete the instruction itself. We can then start deleting every basic block that isn't part of the loop. This is safe to do in any order because we already deleted all the edges between the basic blocks when we deleted all the instructions.

Once the function is cleaned up, we create the speculative callback function which is added to the module. We then copy the body of the temporary function into the callback function. We then replace every uses of the two placeholder variables that was previously added at the top of the instrumentation header, by the function arguments. Since the two placeholder values are no longer needed, we can simply erase them.

### 3.2.2 Original Function

The goal of this instrumentation phase to create a header to the original loop that contains the calls to the yarn runtime library. It also gracefully handles any error conditions indicated by the runtime library. This phase must be executed after the speculative callback has been created because we need to make a reference to it. We also make some light modification to the loop which could invalidate some of the analysis data. The result of the instrumentation is illustrated in Figure iv which is described in details in section 4.4.2 of the architectural document.



Figure - Original Loop Instrumentation

We begin this phase by creating a new basic block for our instrumentation header and insert it before the loop header. We then allocate a region of memory big enough to hold all the data that needs to be passed to the speculative function. The array value list constructed during the analysis pass is then used to load all the required values into the memory region. We then insert a call to the *yarn\_exec\_simple* function with the memory region and the speculative function callback as an argument. After the function call, we then load all the array values that were marked as being modifiable back into local values. We then replace all the users of the original value with the newly loaded value which ensures that our modified values are propagated to the loop. We then check whether the call to the runtime component triggered an error. If it is the case then we branch to the loop. Otherwise we branch to the loop successor.

In order to propagate any modifications made by our speculative function call to the rest of the successor of the function, we then modify the LCSSA PHI nodes so that for each loop exiting value we have a corresponding incoming value from the instrumentation header.

# 5. Bibliography

x

|  |  |
| --- | --- |
| 1 | Aho, Alfred V., Lam, Modica S., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles Techniques, & Tools*. Addison Wesley, Boston, 2007. |

x

1. http://sites.google.com/site/murmurhash/ [↑](#footnote-ref-1)
2. http://www-graphics.stanford.edu/~seander/bithacks.html [↑](#footnote-ref-2)