|  |
| --- |
| ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI |
| Yarn  A Speculative Multithreading System |
| Architecture |
|  |
| **Rémi Attab** |
| **7/30/2011** |

|  |
| --- |
|  |

Table of Contents

[List of Tables 3](#_Toc299762280)

[List of Figures 3](#_Toc299762281)

[List of Code Samples 3](#_Toc299762282)

[1. Introduction 4](#_Toc299762283)

[1.1 Aliases 4](#_Toc299762284)

[2. Yarn System Overview 5](#_Toc299762285)

[3. libyarn Description 5](#_Toc299762286)

[3.1 Coding Conventions 5](#_Toc299762287)

[3.2 Component Overview 6](#_Toc299762288)

[3.2.1 Parallel Program 7](#_Toc299762289)

[3.2.2 libyarn 7](#_Toc299762290)

[3.2.3 Data Tracking 9](#_Toc299762291)

[3.3 yarn\_epoch Description 10](#_Toc299762292)

[3.3.1 Epoch Definition 10](#_Toc299762293)

[3.3.2 yarn\_epoch Interface 10](#_Toc299762294)

[3.3.3 Epoch States Description 11](#_Toc299762295)

[3.4 Task Execution Protocol 13](#_Toc299762296)

[3.4.1 Get Epoch Phase 14](#_Toc299762297)

[3.4.2 Execute Task Phase 14](#_Toc299762298)

[3.4.3 Commit Phase 14](#_Toc299762299)

[3.5 Error Recovery 15](#_Toc299762300)

[4. yarnc Description 16](#_Toc299762301)

[4.1 Environment Overview 16](#_Toc299762302)

[4.1.1 Classical Compiler Architecture 17](#_Toc299762303)

[4.1.2 LLVM Architecture 18](#_Toc299762304)

[4.1.3 IR versus AST 19](#_Toc299762305)

[4.1.4 Coding Conventions 20](#_Toc299762306)

[4.2 yarnc Overview 20](#_Toc299762307)

[4.2.1 Passes Overview 20](#_Toc299762308)

[4.2.2 Passes Dependency 21](#_Toc299762309)

[4.3 Analysis Pass 23](#_Toc299762310)

[4.3.1 Types of Dependencies 23](#_Toc299762311)

[4.3.2 Instrumentation Points 25](#_Toc299762312)

[4.3.3 Array Values 25](#_Toc299762313)

[4.3.4 Loop Compatibility Check 25](#_Toc299762314)

[4.3.5 Loop Heuristic 25](#_Toc299762315)

[4.4 Transformation Pass 26](#_Toc299762316)

[4.4.1 Original Loop 27](#_Toc299762317)

[4.4.2 Instrumented Loop 29](#_Toc299762318)

[4.4.3 Speculative Callback Function 31](#_Toc299762319)

[5. Bibliography 33](#_Toc299762320)

# List of Tables

[Table 1 - Alias to name mapping 4](#_Toc299745874)

# List of Figures

[Figure i - System overview 5](#_Toc299762321)

[Figure ii - libyarn Overview 7](#_Toc299762322)

[Figure iii - Epoch States 11](#_Toc299762323)

[Figure iv - Task execution protocol 13](#_Toc299762324)

[Figure v - AST Example (B) 16](#_Toc299762325)

[Figure vi - Basic Block Exemple (B) 17](#_Toc299762326)

[Figure vii - Basic Compiler Architecture [1] 17](#_Toc299762327)

[Figure viii - LLVM Architecture [1] 18](#_Toc299762328)

[Figure ix - Pass Overview 20](#_Toc299762329)

[Figure x - Pass Dependencies 21](#_Toc299762330)

[Figure xi - Original Loop Structure 27](#_Toc299762331)

[Figure xii - Instrumented Loop Structure 29](#_Toc299762332)

[Figure xiii - Speculative Loop Structure 31](#_Toc299762333)

# List of Code Samples

[Code i - libyarn Error Handling Example 6](#_Toc299762334)

[Code ii - AST Example (A) 16](#_Toc299762335)

[Code iii - Three Address Code Example (A) 16](#_Toc299762336)

[Code iv - Three Address Code Example (B) 16](#_Toc299762337)

[Code v - Basic Block Example (A) 17](#_Toc299762338)

[Code vi - Value Dependency Example 23](#_Toc299762339)

[Code vii - Pointer Alias Example 24](#_Toc299762340)

[Code viii - Pointer Dependency Example 24](#_Toc299762341)

[Code ix - Loop Invariant Example 24](#_Toc299762342)

# 1. Introduction

This document describes the various components that make up the yarn speculative multithreading system. This description will include the composition of the system at various levels, the behavior of certain components and the interaction between these components.

It should be noted that this document will mention critical performance path. Critical performance paths indicate portions of the system that are critical to the overall performances of the system. These sections will therefore be examined more carefully for possible optimizations.

It should also be noted that this document will mention data dependency violations. These occur when a *read* is made before a *write* during the execution of the speculative version but the sequential execution version indicate that the *write* should come before the *read*.

## 1.1 Aliases

It should be noted that in this document we will refer to the system and its components by the names used within the code base. maps these names to the aliases used in the other documents.

|  |  |
| --- | --- |
| Alias | Name |
| Speculative Multithreading System | yarn |
| Runtime component | libyarn |
| Compiler component | yarnc |
| Dependency tracker | yarn\_dep |
| Scheduler | yarn\_epoch |
| Specialized concurrent hash table | yarn\_map |
| Thread pool | yarn\_tpool |
| Thread local storage | yarn\_pstore |
| Pool allocator | yarn\_pmem |

Table 1 - Alias to name mapping

# 2. Yarn System Overview

Figure i gives a general overview of the system and a component by component description is given below.



Figure i - System overview

**Sequential Code:** This is the original sequential code that the user wants to parallelize.

**yarnc:** This is one of the two component that we aim to develop in this project. It instruments the user provided sequential code with calls to the libyarn component. In order to do this, it must first detect any potential sequential bottlenecks and analyze them for dependencies which are then instrumented. This component is described in greater details in section 4.

**libyarn:** This is one of the two component that we aim to develop in this project. It must first ensure that it's provided tasks are run in parallel and it must also ensure that the computations performed by these tasks are coherent. It should be noted that this component operates like any other user libraries. This component is described in greater details in section 3.

**Compiler:** Takes the sequential code of the user as an output and executes the regular compilation steps in addition to the *yarnc* optimization passes. This will generate an executable parallel code that was instrumented to run in parallel. This component is described in greater details in section 4.

**Parallel Program:** This is the final parallel program that can be executed directly by the user.

# 3. libyarn Description

## 3.1 Coding Conventions

The code for the *libyarn* component follows a fairly relax coding convention which is described in this section.

First, any headers that can be included within a client code must adhere to the *Clean C* standard. The rest of the code-base must adhere to the *C99* standard.

Second, every functions, types or defines that are defined as *extern* or are visible to multiple compilation units must be prefixed with *yarn\_*. Individual components should also add an extra prefix to their public interface to make them uniquely identifiable. This effectively creates a namespace that avoids name conflicts between components. Functions, types or defines that are static or are only visible to a single compilation units can use any naming conventions.

Finally, error handling should be handled with *goto*s using the following pattern:

type\_a\* handle\_a = alloc\_res\_a();

if (!handle\_a) goto res\_a\_error;

type\_b\* handle\_b = alloc\_res\_b();

if (!handle\_b) goto res\_b\_error;

//...

return true;

free\_res\_b(handle\_b);

res\_b\_error:

free\_res\_a(handle\_a);

res\_a\_error:

perror(\_\_FUNCTION\_\_);

return false;

Code i - libyarn Error Handling Example

This pattern ensures that error handling is done consistently and cleanly. It will also collect a good amount of information which is useful for diagnosing problems. The call to *perror* should eventually be replaced by a customized function.

## 3.2 Component Overview

gives an overview of the libyarn component along with the interactions between the various interfaces. Each of the components is detailed in one of the 3 sub-sections which each represent one of the highlighted areas in the diagram.



Figure ii - libyarn Overview

### 3.2.1 Parallel Program

The *parallel program* is the output of the *yarnc* component. The instrumentation of the main sequential generates two sub components that interact with different part *libyarn* component.

**User code:** This is the bulk of the original sequential program that contains a call to the *yarn* component to start the parallel execution. This call passes a reference to the *Instrumented code* component which will be used during the execution. If an error is returned by the *yarn* component, the user code will resume executing the original sequential version of the instrumented code. The original sequential code must still recognize any committed changes made by the *libyarn* component. This ensures that the program behaves correctly even when recoverable error occurs.

**Instrumented code:** This component contains a copy of the body of the sequential code to be parallelized that was generated by the *yarnc* component. For example, if a *for*-loop is parallelized then the body of this component will contain the body of the loop. This copy has been instrumented with calls to the *yarn\_dep* component whenever a dependency is read or written.

### 3.2.2 libyarn

The libyarn program is a library that we aim to develop in this project. It is responsible for executing a given task in parallel while making sure that its computation remains coherent. To remain as portable as possible, the interface for this component must adhere to the Clean C standard. The interface is composed of the *yarn* and *yarn\_dep* component. The other components must adhere to the C99 standard.

**yarn:** This is the main public interface for the *libyarn* component and is responsible for performing the execution protocol which is described in the section 3.4. During the execution protocol, it makes calls to the *Instrumented code* component to execute the tasks associated with an epoch. It is also responsible for splitting the computation into separate tasks which are associated with the various epochs. To execute the epoch in parallel, uses the *yarn\_tpool* component to run the execution protocol on multiple thread.

**yarn\_tpool:** This component plays the role of a thread pool that is automatically adjusted to the user environment. Its main responsibility is to execute a given piece of code in parallel and aggregate their return value. Since this component doesn't belong to the critical performance path of the program, it uses standard *pthread* synchronization primitives.

**yarn\_epoch:** This component plays the role of the scheduler for the program and is detailed in section 3.3.

**yarn\_dep:** This component is in charge of tracking all the reads and writes made to dependencies and to make commits or rollback based on the result of the execution protocol. This component is also responsible for issuing rollbacks to the *yarn\_epoch* component when a data dependency violation is detected. Because this component is part of the critical performance path, it requires special optimization attention. This takes the form of lock-free algorithms, pool allocators and the use of bit operators in order to manipulate the components of the *Data Tracking* component.

This component interfaces mainly with the *yarn\_map* component which stores all the data tracking information. Because there's a performance cost associated with accessing elements stored in *yarn\_map* the component also uses the *Buffer index* to quickly access commonly used elements. In order for an element to be represented in this index it must first have been identified by the *yarnc* component and given a unique id. This means that the element can't be a pointer because in most cases it is impossible to determine at compile-time what memory location it is pointing to. It should be noted that components in the index are also indexed by the *yarn\_map* component. This is because they could still be accessed through the *yarn\_map* component if it is used through a pointer.

This component also makes use of the *yarn\_pmem* pool allocator because when using the *yarn\_map* component it must often create temporary elements. These elements are often discarded shortly after their creations and the pool allocator allows the component to reuse these temporaries for a very low cost.

**yarn\_map:** This component is a concurrent hash table that is used to map the *Write buffers* data. This table supports an operation name *probe* which atomically looks for an element and adds it if it's not present. Since this component is part of the critical performance path, it uses a custom implementation that takes advantages of relaxed requirements. Since we don't need to support the remove operation on the table we can implement a linear-probing lock-free scheme. We also use a concurrent resizing scheme where every thread that wants to probe for an element will help copy elements.

### 3.2.3 Data Tracking

This section describes the data structures used by the *yarn\_dep* component to track the read and write operations performed on dependencies.

**Write buffers:** This component tracks the reads and writes made to a dependency. In order to do so, it holds a *read* and *write* flag for every active epoch which are set when a read or a write occurs on the dependency. If the *write* flag is set the value of the *write* is stored in a buffer. If a *read* occurs and the *write* flag for an earlier epoch is set, then the content of the buffer for that *write* flag is returned.

The structure also keeps track of the last committed epoch which is used to ensure that the *write* of an epoch doesn't overwrite the *write* of a later epoch during the commit phase. The *Write buffers* for an epoch are also linked together by a link list which facilitates the commit and rollback of an epoch. Note that a *Write buffer* can belong to multiple epochs at once.

**Buffer index:** This component is used to quickly access the *Write buffers* of commonly used dependencies. This component is described in greater details in the description of *yarn\_dep*.

## 3.3 yarn\_epoch Description

This section describes the *yarn\_epoch* component which plays the role of scheduler.

### 3.3.1 Epoch Definition

An epoch is defined as a scheduling unit within which multiple tasks can be executed. An epoch is represented by an id which can be increased indefinitely (cyclic timestamp). An epoch can't be partially committed but can be partially rollback as long as it is no longer executing.

### 3.3.2 yarn\_epoch Interface

This section describes the interface of the component *yarn\_epoch* which will then be used to describe the states of an epoch.

**next():** Returns the next epoch to execute. The returned epoch may require a rollback before any computation can take place. If an epoch is marked by the *stop()* function then this function will no longer return any new epoch until the marked epoch is rolled-back. If the marked epoch is committed then this function indicates to the caller that the computation is done.

To keep the coherence of the returned epoch, this function will block if there's an active call to the *rollback()* function. This was done to simplify the state transitions and because this function is not within the critical performance path.

**stop():** Signals that the computation is completed at indicated epoch. If the indicated epoch is rollback after the call is made then the epoch will no longer mark the termination of the computation. If two epochs are marked then only the earliest epoch is kept.

**done():** Signals that the computation of the tasks for the epoch is completed.

**rollback():** Signals a data dependency violation on a given epoch and marks the epoch and every subsequent active epoch with the rollback status. It should be noted that this function should always be called with an epoch that is later then the epoch of the caller. Meaning that an epoch can't rollback itself and data dependency violation can only occur with epochs that are more speculative then the current epoch.

To keep the coherence of the epochs being rolled-back, this function will block if there's an active call to the *rollback()* function or the *next()* function. This was done to simplify the state transitions and because this function is not within the critical performance path.

**rollback\_done():** Signals that the epoch has been rolled-back and is ready to be re-used.

**get\_next\_commit():** Returns the next epoch that is ready to be committed. If no epochs are ready the caller will be notified.

**commit\_done():** Signals that the epoch has been committed and is ready to be re-allocated.

### 3.3.3 Epoch States Description

Figure iii presents the state diagram for an epoch. Note that epochs are all pre-allocated and to save memory they are continuously re-used. This explains why *Commit* is marked as the initial state and its outgoing link to the *Executing* state.



Figure iii - Epoch States

**Executing:** This state indicates that the epoch is being computed but could be rolled-back at any moment if a data dependency violation is detected. If it is rolled-back, the computation should stop as soon as possible because any modifications it makes will be discarded.

**Done:** This state indicates that the computation is completed and is ready to be committed. The computations done by this state are still subject to rollbacks. It should be noted that if the epoch is returned by the *get\_next\_commit()* function then it can no longer be rolled-back even though it's state hasn't changed. This is because every earlier epoch are in the *Done* or *Commit* state and can therefore no longer generate any rollback.

**Commit:** This state indicates that every *write*s made by the epoch have been committed to memory and the epoch is ready to be re-used.

**Pending Rollback:** This state indicates that a rollback occurred while the epoch was executing. This transitory state is required in order to prevent the *next()* function from prematurely returning the epoch. This is mainly to prevent multiple threads being allocated the same at the same time which could lead to inconsistencies.

**Rollback:** This state indicates that the epoch is ready to be rolled-back and can be assigned to new thread. Note that when calling next, the current status of the epoch will be returned to the caller before it set to executing. The caller can then examine it in order to determine whether it should perform the rollback or not. See section 3.4 for more details.

## 3.4 Task Execution Protocol



Figure iv - Task execution protocol

describes the task execution protocol which is executed by the *yarn* component. The protocol describes how the various component of the system work together to execute the tasks in parallel and ensure the coherence of the computation. It is separated in three phases that are executed sequentially in an infinite loop and are described in the following sub-sections.

### 3.4.1 Get Epoch Phase

The objective of the first phase is to obtain an epoch to execute. This is done by calling the *yarn\_epoch* function *next()* described in 3.3.2. If the return value of the function indicates that the computation is done then the protocol breaks out of the infinite loop and ends there.

If the *next()* function returns an epoch, it may require that it be rolled-back. If that is the case then the *yarn\_dep* function *rollback()* should be called to clean up the epoch. Once this is done, we signal that the epoch is rolled-back() and we can then proceed to the next phase.

It should be noted that once the function *next()* returns an epoch, then this thread is the only thread that owns this epoch. The ownership of the epoch remains until the end of the second phase. This means that if the epoch is to be re-executed after a rollback or be re-used to execute another epoch, then the thread asking for an epoch will block until it is made available again.

### 3.4.2 Execute Task Phase

In this phase we call the *Instrumented code* component so that it can execute a given task. If multiple tasks have been assigned to an epoch then they are executed here consecutively. If the *Instrumented code* component indicated that the we should stop the computation, then we signal this by calling the *yarn\_epoch* function *stop().* Once the tasks have been completed, we call the *yarn\_epoch* function done*()* to release ownership of the epoch.

During the execution of the task, the *Instrumented code* component will make calls to the *yarn\_dep* functions *load()* and *store()* for every dependencies that are being read or written. If the *store()* function detects a data dependency violation, it will trigger a rollback by calling the *yarn\_*epoch function *set\_rollback()* on the appropriate epoch.

Note that this section of the code is highlighted in red in the protocol which indicates that it is a critical performance path. This is because the performance of the speculative execution will be a function of the number of load and stores executed. The longer the load and stores take to execute, the lesser we can expect the resulting speed-up to be. The rest of the protocol is mostly a constant factor that is added as an overhead. Note that it is possible that performance tests indicate that the constant overhead has much more of an impact then the load and store operations. In this case we should revise our optimization approach.

### 3.4.3 Commit Phase

The objective of this phase is to commit any and every epoch that are currently ready to be committed. This phase is not limited to only the current epoch because it could needlessly force a faster thread to wait for a slower thread. It also allows multiple threads to commit multiple epochs in parallel regardless of which thread executed which epoch.

To obtain an epoch to commit, we must first call the *yarn\_epoch* function *get\_next\_commit()*. This function will either return an epoch to commit or indicate that no epochs are ready to commit. This phase will be executed repetitively until the function returns the later. The returned epoch is also guaranteed to be only returned once which gives ownership of the epoch to the thread. The commit is accomplished by calling the *yarn\_dep* function *commit()* which ensures that the write buffer for that epoch is committed to memory only if no later epochs have committed theirs first. We then release ownership of the epoch by calling the *yarn\_epoch* function *commit\_done()* which changes the state of the epoch.

Note that this scheme guarantees that all the epochs will be committed because:

* The last thread to be executing will commit itself because it is the last epoch to be executed which means that no rollback can occur.
* The last thread will commit all the older epochs that are ready because it must first commit all previous epochs in order to commit itself.
* The last thread will commit the successor epoch because if it was able to commit itself then it will be able his successor as well.
* If a rollback occurs during the third phase, then the thread is not the last thread to be executing.

## 3.5 Error Recovery

This section describes the error recovery scheme employed to ensure that the computation remains coherent even when an error occurs. It should be noted that in order for the *libyarn* component to recover from an error, the error must first be recoverable. For example, an error indicating a failure to open a can be easily recovered from while a signal that indicates a segmentation fault should not be recovered from. Also, the coding convention to handle errors is described in section 3.1.

When an error is detected, the library should clean-up any allocated memory and return to the *User code* component with an error code. That component should then execute the original sequential program. It should be noted that by the point where an error is detected, the *libyarn* component may have already committed several tasks. In this scenario, the original sequential program should take into account these modifications and continue executing accordingly. This means that the *yarnc* component should modify the original sequential program in order to allow for this.

The *libyarn* component should also keep track of the error count and simply stop executing any speculative code if this count exceeds a certain threshold. This will avoid needlessly slowing down the program by failing on the same problem over and over again.

Finally, all detected errors should be logged to a file to facilitate diagnosing of problems. This logging can be disabled by the user during the compilation process or at runtime.

# 4. yarnc Description

This section describes the *yarnc* component which is used to automate the parallelization of a sample program. Since this automation will take place as part of the compilation process, we will first briefly describe the LLVM compiler which we chose for our project. We will then describe how the *yarnc* component is implemented.

## 4.1 Environment Overview

In this section we will give a brief overview of how a compiler operates and how LLVM implements these ideas. A compiler usually operates on two major types of data structures:

**Abstract Syntax Tree (AST):** This data structure is used while parsing the original source code and gives a tree representation of each operation. A node within that tree represents and operator and its branches the operands. This representation can be used to check for syntax or semantic errors in the original program and for translation into IR. The types of node and operand are specific to a given programming language.

The following is a short example of how the instruction presented in Code ii is represented by the AST in Figure v.

a = b + 10

Code ii - AST Example (A)



Figure v - AST Example (B)

**Itermediate Representation (IR):** This data structure is used during the optimization and the machine code generation process and usually takes the *three address code* (TAD) form. All TAD instructions are composed of 3 addresses and an operand. The following is an example of the equivalent TAD in Code iv of the C instruction in Code iii :

a = (b-100) \* (c+10);

Code iii - Three Address Code Example (A)

t1 = c + 10

t2 = b - 100

a = t1 \* t2

Code iv - Three Address Code Example (B)

TAD instructions are grouped within a data structure called *basic blocks* (BB). These basic blocks are then linked together to form the control flow of the program. For example, the *if* statement illustrated in Code ii would be represented by the basic blocks illustrated in Figure v.

a = i;

if (a > 10) {

b = 10;

}

else {

b = 100;

}

c = b;

Code v - Basic Block Example (A)



Figure vi - Basic Block Exemple (B)

IR is useful because it can be easily manipulated and analyzed the optimizer. It's also a representation that is very close to assembly and can therefore be translated much more easily. IR is also independent of the original programming language.

### 4.1.1 Classical Compiler Architecture

Figure v shows the most widely used compiler architecture that promotes modularity as well as code re-use. Note that the described architecture is ideal and is not necessarily perfectly implemented in all compilers.



Figure vii - Basic Compiler Architecture [[1](#Bro11)]

**Front-End:** This component is responsible for taking in an input source code and constructing and AST representation. This AST is then checked for syntax errors or semantic errors. This component then translates the AST into IR code which is then passed to the optimizer.

Since this component primarily works on the AST it is therefore independent to a given programming language. Since the output IR is language agnostic, it is possible to have multiple front-ends for multiple languages without having to modify the optimizer or back-end.

**Optimizer:** This component is responsible for optimizing the output of the front-end. It is usually organized as a series of *passes* that are run in a certain order to make progressive improvements to the program. The improvements will usually aim to either reduce the size of the program or improve the performances of the program.

Since this component manipulates only the IR code, it is independent of the language used by the original source code and can therefore be re-used for any language as long as we have the proper front-end. It is also independent of the target system and can also be re-used for any target system architecture as long as we have the proper back-end.

**Back-End:** This component is used to transform the IR code generated by either the front-end or the optimizer and transform it into a bit representation that is directly executable by the processor. Since different processors have different instruction sets (or language), it is possible for a compiler to have multiple back-ends; one per supported instruction sets. The back-end may also apply a number of optimization passes that are specific to a type of processor architecture.

Since the Back-End uses IR code as an input, it is therefore independent of the optimizer and the front-end.

### 4.1.2 LLVM Architecture

In this section we detail how LLVM implements the classical architecture detailed in the section 4.1.1. Figure vii illustrates the various components that make up the LLVM compiler.

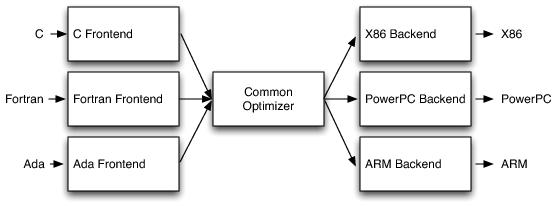


Figure viii - LLVM Architecture [[1](#Bro11)]

On the left we can some of the front-ends that exist for LLVM. More specifically, the C front-end which is implemented by the *clang* project, also support the C++ and Objective-C language. Each of the front-ends, implement their own lexer, parser and AST to best suite the language being processed.

The optimizer contains large number of passes which are categorized as either analysis passes or transformation passes. An analysis pass is used to obtain information about a given piece of code while transformation passes are used to apply an optimization. One pass will usually depend on other passes having being executed in order to run properly. LLVM provides a pass manager which can be used to register dependency between passes. The manager will then be responsible for scheduling the execution of the passes in the most efficient way possible. All the passes are built to be as independent from each other as possible.

The LLVM back-ends are implemented as another series of optimization passes. Since machine code generation is out of scope for this document, the LLVM back-ends will not be discussed in further details.

LLVM is also provided as an extensive set of libraries that can be used to construct other compiler related utilities. This feature is used by yarnc to easily build our own compiler tools and re-use LLVM's powerful IR code.

Also of note, LLVM's IR code is a first-class language. This means that it can easily be written by hand, and compiled without ever needing a higher level language like C. It can also be easily converted between its string, serialized binary (named byte code) and programmable interface. This makes it easy to manipulate and read a program in its IR form. It also simplifies other tasks like writing test cases for an optimization pass.

### 4.1.3 IR versus AST

To implement yarnc, we can go with two different routes. We can either implement it as an extra front-end which will generate an instrumented C file or as an optimization pass which will generate LLVM bytecode. This boils down to whether we want to manipulate an AST or the IR representation of the code. In this section we will examine both approaches.

**AST:** This approach involves using the clang front-end to obtain the program's AST. yarnc would then analyze the program and modify the AST in order to add the loop-instrumentation. It would finally output a C file which would look largely like the original but with the extra instrumentation.

This approach has the advantage of allowing the user to see the result of the instrumentation process and edit it if needed. It also allows us to create a standalone tool that has no dependency on LLVM which allows the user to use the output with any other C compiler.

The disadvantages are that analyzing the code in its AST form is very difficult. We would therefore have to convert it to its IR form and do the analysis with that. This approach is risky because too much information may be lost during the translation process making it difficult to apply the analysis result to the original AST. This also makes yarnc dependent on the clang supported language. Finally, we would also have to instrument a code that has not been optimized. This means that our tool could instrument a loop that would have later been optimized-out by the optimizer.

**IR:** This approach involves using one of the LLVM front-ends to generate the IR form of the program. The IR would then be fed to the optimizer and the yarnc optimization pass would then analyze and transform the code. In this case yarnc would be part of the regular compilation process.

The advantage of this approach is that it is very easy to analyze and manipulate the program in this form. We can also make sure that the appropriate LLVM optimizations have been executed to reduce the number of false-positive. Finally, this also makes yarnc compatible with any language that has an LLVM front-end.

The disadvantages of this approaches is that yarnc is entirely dependent on LLVM which also forces the user to use LLVM to compile his code. It's also very difficult for the user to inspect or modify the result of the instrumentation process.

To reduce the complexity of the yarnc component we decided to implement it as two LLVM optimization passes. This effectively makes the user dependent on LLVM but deem this acceptable because LLVM is becoming increasingly more popular and mature. We also don't think it's a good idea for the user to be able to inspect and instrument the generated code. The yarn project will instead provide tools in a later version which will allow the user to guide the instrumentation process. Note that it's still possible for the user to inspect the instrumented code using the LLVM dis-assembler.

### 4.1.4 Coding Conventions

Since yarnc will constructed as two LLVM passes, we will adhere to the LLVM coding conventions[[1]](#footnote-1). This will ensure that our code is consistent with the other LLVM passes. Note that this will note conflict with the coding conventions for libyarn specified in section 3.1 because the two code base should not intersect.

## 4.2 yarnc Overview

This section gives a brief overview of how yarnc interacts and depends on the various LLVM components.

### 4.2.1 Passes Overview



Figure ix - Pass Overview

Figure ix describes the compilation process that the user must use in order to properly instrument his program. We begin by passing the source through the any LLVM front-end (in this case clang) to get the IR representation for the program. We then pass it to the *opt* program which is the LLVM optimizer. The *-O2* option represent the usual collection of optimizations. We do this step before the instrumentation process because these optimizations will greatly simplify the code as well as inlining certain function and removing dead code. These transformations greatly simplify our analysis task as well as removing false instrumentation targets.

After the first round of optimization, we call opt again but using three new optimization options. The *-loop-simplify* and the *-lcssa* options are provided by LLVM and ensures that the loops within the program are formatted in a standardized way which greatly simplifies our analysis. The *-yarn-loop* options invokes our instrumentation process. Note that the *-yarn-loopinfo* option will also invoke our analysis pass but since it's a dependency of the instrumentation pass, the user doesn't need to specify it. Note that call to *opt* should eventually be replaced by a standalone program which should ensure that all the appropriate dependencies are invoked.

Once the second optimization pass has been completed, it is currently a good idea to call *opt* with the *-O2* option to ensure that any code that has been added is fully optimized but this is optional. In any case, the user then invokes *llc* to generate the machine code and link the various translation units in order to generate the final executable.

### 4.2.2 Passes Dependency

Figure x describes the various passes needed by the instrumentation process as well at their dependencies. Note that the description of these passes include many technical terminology used by compilers. It's not our goal here to describe these terms and the reader should therefore refer to the LLVM documentation or any compiler reference manual in order to get a more detailed definition.



Figure x - Pass Dependencies

**Loop Info:** This analysis pass is used to find and gather information about loops. The information gathered is fairly extensive and is one of the most important analysis pass that yarn uses. Note that executing the *Simplify Loop* pass greatly simplifies a number of the data gathered by this pass.

**Alias Analysis:** This analysis pass is used to determine whether two pointers are aliases of each other. For two pointers to be aliases, they must potentially read or write to the same address. For two pointers to be strict aliases, they must be equal. In the yarn passes, it is used to analyse the pointer dependency and determine whether they should be instrumented or not.

**Dominator Tree and Post Dominator Tree:** These analyses pass build a data structure known as a dominator tree. These trees are used to determine if a basic block dominates another basic block[[2]](#footnote-2). In the yarn passes, they are used to determine where a call to yarn\_load and yarn\_store should be added in order to cover all the load or store operations for a given dependency.

**Simplify Loop:** This transformation pass ensures that a loop respects a number of properties:

1. There must be only one edge leading into the loop. This edge will originate from the loop's predecessor and lead into its header.
2. There must only be one back-edge in the loop. This back-edge will originate from the loop's latch and lead into its header.
3. All the edges exiting the loop, will lead to the unique successor block of the loop.

More information about these properties is detailed within the LLVM documentation. These properties are used extensively by the yarn passes.

Note that this pass preserves the dominator trees and the alias analysis data so they don't need to be recalculated. The data within the *Loop Info* pass are invalidated by this pass. Because this analysis pass is used by later passes, it must be recalculated.

**Loop Closed Single Static Assignment (LCSSA):** This transformation pass ensures that *phi* nodes are added to the successor of the loop for every value defined within the loop that are used outside the loop. This is used by the yarn passes in order to identify exit dependencies that correspond to the entry dependencies. It is also used to identify exit-only dependencies.

Note that this pass preserves the data of all the previous analysis passes.

**Yarn Loop Info:** This is the yarn analysis pass that gathers a number of information about a given loop. It is described in greater detail in the section 4.3.

**Yarn Instrument Loop:** This is the yarn transformation pass that injects the calls to the libyarn component. It is described in greater detail in the section 4.4.

## 4.3 Analysis Pass

This section describes the *Yarn Loop Info* analysis pass which is part of the yarnc component.

### 4.3.1 Types of Dependencies

The analysis pass is mainly used to identify dependencies within a loop that must be instrumented. This section describes the there are three kinds of dependencies that the analysis pass must detect.

#### 4.3.1.1 Value Dependency

Value dependencies are essentially variable that are used inside and outside the loop. Code vi shows three types of value dependencies.

int x = 0;

int y;

for (int i = 0; i < 10; ++i) {

int t = x + i;

x = t;

}

return y;

Code vi - Value Dependency Example

**Entry Dependencies:** This type is represented by the *x* and *i* variable and represent a value is used before the loop and within the loop. These are identified using the predecessor value of the header *phi*. Note that an entry dependency can also be an exit dependency. In this case both cases are linked together using the back-edge value of the header *phi* nodes.

**Induction Variables:** This is a special case of the entry dependencies and is represented by the *i* variable. Although we currently don't support it, later versions of yarn could eventually eliminate instrumentation made to these variable because of their predictable behaviour. To identify an induction variable, LLVM provides the *-indvar* transformation pass which identifies induction variables and if necessary modifies to loop to standardize it (starts at 0 and increments by 1).

**Exit Dependencies:** This type is represented by the *y* variable and represents a value that is used within the loop and after the loop. These are identified using the *phi* nodes generated by the *-lcssa* transformation pass.

#### 4.3.1.2 Pointer Dependency

Pointer dependencies are any memory addresses that are accessed using pointers. In order to understand how pointers are treated, it important to first understands how they may alias each other. Code vii shows several types of pointer aliasing. We will then describe the types of dependencies that can be generated by pointers.

const int\* pArr = {1,2,3};

const int\* pEq = pArr;

const int\* pI = pArr + i;

const char\* pStr = "abc";

Code vii - Pointer Alias Example

**Strict Alias:** This alias type is represented by the *pArr* and the *pEq* pointers. They are strict aliases of each other because it can be proven that both these variables are equal to each other. Yarn treats values that are strict aliases of each other as being the exact same value. So any property that applies to one will apply to all.

**May Alias:** This alias type is represented by the *pArr* and the *pI* pointers. They may alias because *pI* may refer to the same memory location as *pArr* if the value of *i* is *0*. yarn treats these aliases conservatively and if a pointer will be instrumented if any of the values it may alias are instrumented.

**No Alias:** This alias type is represented by the *pArr* and the *pStr* pointers. Since it can be proven that neither of these two pointers will never point to the same memory location they will therefore never alias. Yarn doesn't transfer any properties from pointer to another in this case.

pArr[0] = 1; // 1

t = \*pI; // 2

return pStr; // 3

Code viii - Pointer Dependency Example

**Write Dependency:** This type of dependency describes any memory location that is written by a pointer. In this case any the pointers used to do the writing and the pointers that read from the memory location must all be instrumented. In the example, this includes both the instruction 1 and 2. Even though the second instruction only reads from the pointer, it must still be instrumented because *pI* may alias the *pArr* pointer.

**Read-Only Dependency:** This type of dependency describes a memory location that is only ever read from. In this case no instrumentation needs to take place. In the example, this includes the instruction 3 because *pStr* is only read from and it doesn't alias with any other pointers.

#### 4.3.1.3 Loop Invariant

A loop invariant is any value that is only read within the loop.

int x = 100;

int\* p = ...;

for (int i = 0; i < 10; ++i) {

\*p += x + \*p;

}

Code ix - Loop Invariant Example

Code ix contains two loop invariants: the variable *x* and the pointer *p*. Note that in the example, the pointer is the invariant and not the memory location it points to. It is important to distinguish between the pointer and the pointee.

Loop invariants don't need to be instrumented but they must still be detected because their values will have to be communicated to the speculative method. More details are given in section 4.4.

### 4.3.2 Instrumentation Points

Instrumentation points are generated by the analysis pass and they indicate to the transformation pass what instrumentation that needs to take place and where. This list is compiled using the value and pointer dependencies described in the previous section. To determine where a load instrumentation should be placed, it calculates the nearest common dominator block for all the users of that value. To determine where a store instrumentation should be placed, it calculates the nearest common pos-dominator block for all the writes on that value (this might involve exploring the origins of *phi* nodes).

This list is compiled during analysis because it greatly simplifies the complicated instrumentation process. Note that this is a concept that is independent of the other analysis functions. In the future, it could be refactored into its own pass.

### 4.3.3 Array Values

An array value represents a value that need to be passed to the speculative function. The list of all array values is compiled using the value and pointer dependencies as well as the invariants. This list is compiled during analysis because it greatly simplifies the header creation during the instrumentation process.

### 4.3.4 Loop Compatibility Check

This is a check that is performed before a loop is analysed to ensure that the loop can be safely instrumented. Because there are a number of elements that are currently not supported by the runtime component, we must make sure that we don't attempt to instrument these.

Currently, the yarn analysis pass only checks for function calls which may modify the program's memory. This means that only functions that reads its arguments are accepted within a loop. We disallow all other function calls because they may have side-effect which we can't track using our run-time component. Note that in later versions it would be possible to aggressively inline these functions or instrument them directly in order to allow for a greater range of functions to be processed.

In later versions, the analysis pass should also check the types of the various dependencies to ensure that they fit our alignment and size requirements. We should also look into providing support for the various LLVM intrinsic.

### 4.3.5 Loop Heuristic

The analysis pass must also include a heuristic in order to determine whether a loop is worth instrumenting or not.

Our current heuristic uses the ratio of instrumentation points versus the number of instructions. If that ratio is below a certain threshold, the loop is instrumented. This is a very weak heuristic that is bound to be overly permissive. It also doesn't take into account the data that can be gathered using the benchmark program described in the test document.

In a future version, the heuristic should be improved to look for more code structures within the loop. Special attention should be given to nested loops. We should also take into account data reported from various other sources like a yarn specific profiler and the yarnb benchmarking program. We should also consider providing a mean by which the user could guide the instrumentation process.

## 4.4 Transformation Pass

In this section we will examine how a loop is instrumented and how the information provided by the loop analysis is used to effect the instrumentation. The instrumentation process is divided in two phases described below.

**Instrumented Loop:** In order to initiate the speculative execution, we must also instrument the original loop. This step is described in section 4.4.2.

**Speculative Function Callback:** Because the runtime component requires that a callback is created for each speculative task to execute, we must extract the loop and place it in its own function. This step is described in section 4.4.3.

### 4.4.1 Original Loop

Before we examine how the original loop is transformed, we must first understand how a loop is structured. The loop structure shown in Figure xiis enforced by the Simplify Loop pass that is executed before our transformation pass.



Figure - Original Loop Structure

**Predecessor:** This block contains all the code that is executed before the loop. Note that this is usually composed of many blocks and can contain complicated control flow. Regardless of its complexity, it must still contain a single block that has an edge leading into the loops header which will dominate all the blocks within the loop.

**Header:** This block contains the header *phi* nodes that are used in some of the analysis passes. It is also the target of the only back edge of the loop that originates from the latch. This block must dominate all the blocks contained within the loop.

**Body:** This block contains the computation of the loop. If in the loop there's an instruction that stops the computation, its edge will lead to the successor. Note that this block may be composed of multiple blocks and can contain complicated control flow logic. Regardless of its complexity, all the control paths must eventually lead to the latch block or the successor block. Note that if this block doesn't contain any control flow logic then the header, the body and the latch will essentially be the same block.

**Latch:** This block is the only block that can contain a back edge for the loop. It will also usually contain the *for*-loop check statement which determines whether to continue executing the loop or not.

**Successor:** This block contains all the code that must be executed after the loop terminates and all control paths that pass through the loop must eventually reach this block. The phi node added by the LCSSA transformation pass will also be added at the beginning of this block.

### 4.4.2 Instrumented Loop

This section describes how the original loop is instrumented in order to initiate the call to the speculative function. Note that in reality this is done after the creation of the callback function in order to not prematurely invalidate the analysis data needed to create the speculative function. Figure xii shows the result of the instrumentation process. The dotted red edges are edges that were deleted while everything in green were elements that were added.



Figure - Instrumented Loop Structure

In order to initiate the speculative execution we create a new block that is added between the loop header and the loop's predecessor. The predecessor is then redirected to our newly added block. Finally the LCSSA *phi* nodes in the successor are modified to accept the values passed by the *success* edge. Also, the values that contain the result of the instrumentation are propagated to the loop itself in case that the *error* edge is executed. This means that if an error occurs during the speculative execution, the loop will be able to continue executing at the exact spot where the speculative execution stopped. This fulfills our runtime requirements stated in section 3.5

The instrumentation header is composed of 4 sections:

**Packing the Data:** This section uses the array value list created by the analysis pass (section XXX) to create a data structure has an element for each dependency identified by the analysis pass. It then fills this data structure with the entry value dependencies and the invariants. This structure allows us to pass all the data for the speculative execution to the speculative callback and it also gives a memory address to all the value dependencies. Having an address is important because the runtime component only accepts addresses for the *yarn\_load* and *yarn\_store* calls.

**Initiate the Speculative Execution:** This section uses the data structure constructed in the previous section to invoke the *yarn\_exec\_xxx* function. It then saves the result in a value that will be needed during the *Check the Result* phase.

**Unpacking the Results:** In this section, all the value dependencies are extracted from data structure and assigned to a value. These values will replace the values from predecessor in the loop and in the successor block. This phase also uses the array value list as well as their associated value dependency data generated by the analysis pass.

**Check the Result:** Checks the result of the *yarn\_exec\_xxx* call. If no errors were detected, it takes the *success* edge to the successor block. If an error was detected, it takes the *error* edge and executes the loop with the last known good state.

### 4.4.3 Speculative Callback Function

This section describes how the speculative callback function is created and instrumented.



Figure - Speculative Loop Structure

Figure xiii illustrates the result of the process. The elements that were removed are shown in red while the elements that were added are shown in green. The callback function is created in four phases which are described below.

**Callback Function Creation:** The first step is to clone the loop from the original function. Because there is no safe way to only clone the loop, we must therefore clone the entire function. Once the function is cloned, we create a new instrumentation header block that is placed between the loop header and the loop's predecessor.

The responsibility of the instrumentation header is to unpack the data structures created in the *packing the data* phase of section 4.4.3. Similar to that phase we use the array value list created by the analysis pass in order to determine how to unpack each element of the structure. If the element is an invariant, then the unpacked value is propagated to the loop.

**Body Instrumentation:** In this phase we use the instrumentation point list generated by the analysis phase to instrument the body of the loop. Since most of the heavy lifting was accomplish during analysis, this phase simply concentrates on creating the correct instruction sequence at the indicated position. If it's a *load* instrumentation it must also propagate the loaded value to its users.

**Exit Conditions:** The goal of to replace the loop by the execution of a single iteration of the loop. We begin by creating two blocks after the latch which will both contain a single return instruction. The *continue* block will indicate to the runtime component that the loop should continue to execute while the *break* block will indicate to the runtime component that the loop should stop executing. We then redirect the single continue edge in the latch to the *continue* block and redirect all the break edges in the body and the latch to the *break* block.

**Cleanup:** The previous phases should have ensured that no values in the predecessor blocks are in use in the loop's body and that no control flow originating from the loop should lead to the successor block. This allows us to safely erase all the successor and predecessor blocks. We must also erase the *phi* nodes that are present in the header of the loop because they are no longer needed.

# 5. Bibliography

x

|  |  |
| --- | --- |
| 1 | Brown, Amy, Wilson, Greg, and Lattner, Chris. *The Architecture of Open Source Applications*. 20011. |

x

1. http://llvm.org/docs/CodingStandards.html [↑](#footnote-ref-1)
2. A basic block A dominates a basic block B if all possible control flow of the program passes through A in order to get to B. [↑](#footnote-ref-2)