|  |
| --- |
| ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI |
| A Speculative Multithreading System |
| Architecture |
|  |
| **Rémi Attab** |
| **6/22/2011** |

|  |
| --- |
|  |

Table of Contents

[List of Figures 2](#_Toc296536510)

[List of Tables 2](#_Toc296536511)

[1. Introduction 3](#_Toc296536512)

[1.1 Aliases 3](#_Toc296536513)

[2. Yarn System Overview 4](#_Toc296536514)

[3. libyarn Description 5](#_Toc296536515)

[3.1 Coding Conventions 5](#_Toc296536516)

[3.2 Component Overview 5](#_Toc296536517)

[3.2.1 Parallel Program 6](#_Toc296536518)

[3.2.2 libyarn 6](#_Toc296536519)

[3.2.3 Data Tracking 8](#_Toc296536520)

[3.3 yarn\_epoch Description 9](#_Toc296536521)

[3.3.1 Epoch Definition 9](#_Toc296536522)

[3.3.2 yarn\_epoch Interface 9](#_Toc296536523)

[3.3.3 Epoch States Description 10](#_Toc296536524)

[3.4 Task Execution Protocol 12](#_Toc296536525)

[3.4.1 Get Epoch Phase 13](#_Toc296536526)

[3.4.2 Execute Task Phase 13](#_Toc296536527)

[3.4.3 Commit Phase 13](#_Toc296536528)

[3.5 Error Recovery 14](#_Toc296536529)

[4. yarnc Description 15](#_Toc296536530)

# List of Figures

[Figure i - System overview 4](#_Toc296536481)

[Figure ii - libyarn Overview 6](#_Toc296536482)

[Figure iii - Epoch States 10](#_Toc296536483)

[Figure iv - Task execution protocol 12](#_Toc296536484)

# List of Tables

[Table 1 - Alias to name mapping 3](#_Toc296536485)

# 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 - 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 - 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.

**Parallel Code:** This is the output of the yarnc component. It's essentially the original user provided sequential code that has been instrumented to run in parallel.

**User Libraries:** This represents any extra libraries that the original sequential code requires to run.

**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 output of the yarnc component, the libyarn library and any other required user libraries in order to build the final executable. It should be noted that this component could be any currently available standard compliant compilers like *clan*g, *gcc*, *Visual Studio*, etc.

**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;

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 - 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 right 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 - 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 - 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 will be completed once more information has been gathered about the *yarnc* component.