VERIFIABLE EARLY-REPLY WITH C++

A Thesis

by

STEPHEN WENDELL COOK

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Computer Science

VERIFIABLE EARLY-REPLY WITH C++

A Thesis

by

STEPHEN WENDELL COOK

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Scott Pike
	Tom Wehrly
Head of Department,	Valerie Taylor

May 2007

Major Subject: Computer Science

ABSTRACT

Verifiable Early-Reply with C++. (May 2007)

Stephen Wendell Cook, B.S., Colorado School of Mines

Chair of Advisory Committee: Dr. Bjarne Stroustrup

Concurrent programming can improve performance. However, it comes with two drawbacks. First, concurrent programs can be more difficult to design and reason about than their sequential counterparts. Second, error conditions that do not exist in sequential programs, such as data race conditions and deadlock, can make concurrent programs more unreliable. To make concurrent programming simpler and more reliable, while still providing sufficient performance gains, we present a concurrency framework based on an existing concurrency initiation mechanism called "Early-Reply".

Early-Reply is based on the idea that some functions can produce final return values long before they terminate. Concurrent execution begins when return value of a function is returned to the caller, allowing the rest of the work of the function to be done on an auxiliary thread. The simpler sequential programming model can be used by the caller, because the concurrency is initiated and hidden within the function body. Pike and Sridhar recognized Early-Reply as a way for sequential programs to get the benefits of concurrent execution. They also discussed using object-oriented programming to serialize access to data that needs synchronization. Our work expands on their approach and provides an actual C++ implementation of an Early-Reply based framework.

Our framework simplifies concurrent programming for both users and implementers by allowing developers to use sequential reasoning, and by providing a minimal framework interface. Concurrent programming is made more reliable by combining the concurrency synchronization and initiation into one mechanism within the framework, which isolates where race conditions and deadlock can occur. Furthermore, this isolation facilitates the development of a simple set of coding guidelines that can be used by developers (through inspection) or static analysis tools (through verification) to eliminate race conditions and deadlocks.

As a motivating example, we parallelize an instructional compiler that processes multiple input source files. For each input file; the parsing and semantic analysis execute on the calling thread, while the code optimization and object code generation execute on an auxiliary thread. Speedups of 1.5 to 1.7 were observed on a dual processor confirming that sufficient performance gains are possible.

DEDICATION

To my wife, Audrey, who had to sacrifice many things so that I could go back to school and change careers at a later stage in our lives. This was particularly difficult because she had to shoulder much of the burden of raising our two twin daughters during their early years.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Stroustrup for all of his guidance throughout the course of my research. It has been a pleasure to work with him and to get the benefits of his infinite wisdom on all the various research topics on which we worked together. I also would like to thank Dr. Pike for introducing me to my thesis topic of "Early-Reply" and for all of his time and effort in reviewing possible approaches and making significant suggestions. Although not on my committee, Dr. Dos Reis was very helpful in attending practice presentations and offering many suggestions. I would also like to thank Dr. Wehrly for his efforts and being a part of my committee.

Peter Pirkelbauer, Damian Dechev, Yuri Solodkyy, and Luke Wagner who are students in Dr. Stroustrup's Programming Tools, Techniques, and Languages group, were also very helpful in refining my research and defense presentation. I would like to thank them also.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

Page

LIST OF TABLES

Page

1 INTRODUCTION

1.1 Motivation

How to make software applications run faster is a popular research area. The main reasons for wanting an application to run faster are:

1) The program's total execution time needs to be improved (throughput).
2) Certain program segments require a faster response (response time).

For example, by improving the execution time of a compiler, the development times of many software projects could be shortened. Furthermore, shorter compilation times are likely to encourage an increase in test runs which could improve the reliability of the applications being developed. Real-time applications can be deemed unsuitable or even life threatening if specific code segments cannot meet required deadlines. Finally, interactive applications that exhibit long wait times between subsequent activities can become so frustrating that the application is no longer used.

Given these compelling reasons for improving the performance of applications, what can be done to make applications faster? From the perspective of a software developer, the most common approaches to improve application performance are:

- Upgrade to faster hardware.
- Design more efficient algorithms.
- Introduce concurrency so that multiple tasks can execute simultaneously.

Upgrading to faster hardware without any other changes is often not a viable option because; the cost exceeds the limit of affordability, the current hardware is already top of the line, or

---

This thesis follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

environmental constraints. Improvements in response time can sometimes require an order of magnitude improvement, something that may not be possible with just faster hardware. Re-engineering a more efficient algorithm can be effective at times, but typically requires considerable skill and domain knowledge, and often does not scale well as a general solution. On the other hand, introducing concurrency into an application can encompass a wide variety of applications, and can be effective in improving both throughput and response time.

Concurrency can take place at the machine instruction level, statement level, subprogram level, and the program level. Machine instruction level concurrency (i.e. pipelining) is handled at the computer hardware level. Statement level concurrency is used in techniques such as loop unrolling and automatic parallelization, and thus more in the domain of the compiler writer. Subprogram level concurrency is when two or more subprogram units from the same program execute concurrently, and is typically controlled by the developer. Program level concurrency is concurrent execution of two or more distinct programs, and is generally controlled by the operating system [1]. Since our concern is what an application developer can do to improve performance, we will from this point on be talking about subprogram level concurrency, which is under the developer's direct control.

There are two categories of subprogram level concurrency; physical and logical. Physical concurrency occurs when multiple processors are available, and at least two subprograms from the same program are executing on different processors simultaneously. Physical concurrency can be used to improve both throughput and response time. Logical concurrency occurs when at least two subprograms from the same program execute in an interleaved fashion on the same processor. Each subprogram executes on the processor until either its time slice is up, makes an I/O request, or terminates, allowing another subprogram to execute on the processor. Logical concurrency is typically used only for improving response time for interactive applications. However, in some rare cases where one of the interleaved subprograms must block for an I/O request and another is CPU bound, logical concurrency could be used to boost throughput. In this case, the I/O bound subprogram makes progress via the DMA (Direct Memory Access) while the CPU bound subprogram executes in parallel on the CPU.

Subprogram level concurrency can be accomplished by creating either additional processes or additional threads. In this thesis, only threads will be considered for three reasons. The first reason is that threads are more lightweight than processes by consuming less overall memory and being faster to create/delete/context switch, making it more likely that better performance gains will be realized. The second is that threads share the same address space as the process that creates them, which makes it easier to share resources and communicate between threads. This brings up the counter argument that it is easier to prevent race conditions from occurring if processes are used over threads, since by default there isn't any shared memory. However, the communication overhead of message passing between processes could be just enough to wipe out any potential performance gains for certain applications. The third and last reason is that it is the more interesting case of the two, especially with respect to data race conditions, and therefore the thread model will be the only one discussed in this work.

Within the shared-memory model, there are two types of operations; SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data). With the SIMD type (a synchronous mode), all processors execute the same program instruction during each time unit, but with different data. Whereas with the MIMD type (an asynchronous mode), each processor may execute a different instruction or operate on data different from those executed by any other processor during any given time unit [2]. Within the context of this work, we can think of SIMD as "data parallelism" and MIMD as "task parallelism". We will focus on task parallelism in this work, because it lends itself better to the approach of Early-Reply.

A recent trend in personal desktop computing is the rise of multi-core microprocessors, where two, four, or more cores are available on a single processor chip. This trend can be attributed to chip makers looking for different ways to maintain the performance gains that have been enjoyed over the last several decades, because the traditional way of simply increasing the clock rate is reaching its limits. Moore's Law that the number of transistors on an integrated circuit will double every 18 months, will continue to hold for the foreseeable future, and will allow clock rates to improve as they have in the past. However, side effects induced by higher clock rates, such as increased temperature, are making the CPUs too unreliable to be cost effective. This has caused the focus to shift to other approaches for improving performance, such as larger caches and multiple computing cores. To take full advantage of the increased processing power of the

multi-core chips, developers must explicitly parallelize their applications, which is new territory for many developers. This makes approaches to make parallel programming easier (as Early-Reply does), all the more important.

Despite the benefits of introducing concurrency into applications there are many reasons to avoid it. Listed below are some reasons why it would be undesirable to introduce concurrency into applications:

- Applications not inherently parallel become difficult to design and reason about, which increases development time and reduces the application's maintainability.
- Debugging is much more difficult.
- Reuse of concurrent components or libraries is more difficult.
- Code is dependent upon the concurrency library used or number of processors available, making the application less portable.
- Safety error conditions which are unique to concurrent programs, such as data race conditions* can occur when shared data is not synchronized for orderly access between concurrently executing subprograms. This leads to programs that produce unreliable results and errors that can go undetected for long periods of time.
- Liveness error conditions such as deadlock can occur, which are also unique to concurrent applications, when two or more concurrently executing subprograms halt prematurely because of the circular dependencies of resources needed and resources held. Other liveness error conditions such as livelock and starvation are also possible.
- Difficulty in predicting the performance gains of introducing concurrency (if any at all).

* - A data race condition signifies that one of the three data hazards, RAW (read after write), WAR (write after read), or WAW (write after write), may occur when there is at least one read and one write access to the same data variable (each belonging to a different thread) that execute in the incorrect order. For example, a RAW data hazard indicates that the intended order of a data variable access is a write followed by a read, but because the two operations are on different threads, the read may occur before the write is finished resulting in the incorrect old value being read. It must be also noted that the word "resource" is more appropriate than "data", where a resource could refer to items other than a data value in memory, such as a thread id or file

descriptor. However, for more fluid reading of this text, data will be used throughout even though they could be used interchangeably.

The benefits and drawbacks of concurrency just discussed lead to a set of design criteria:

- Improved Performance (throughput and/or response time)
- Usability (sequential reasoning)
- Maintainability (easier to debug)
- Portability (from concurrency library, hardware architecture)
- Safety (no race conditions)
- Liveness (no deadlocks or starvation)
- Predictability (is improved performance possible?)

These criteria will provide the motivation for much of the work of this thesis.

1.2 Related Work and Background

When introducing concurrency into an application, there are two primary issues to consider; how to initiate the concurrency, and how to coordinate the concurrency. Early-reply in its raw form is a technique to initiate concurrency. Since there are many ways to introduce concurrency into an application, a brief review for each approach across the spectrum of possibilities (including Early-Reply) will be given, so that Early-Reply can be put in perspective with the others. Although we will not discuss the various synchronization mechanisms in this review, there will be some focus on how complete concurrent tasks can be synchronized, so that we can begin to understand the issues that relate to concurrency coordination. As stated earlier, multiple threads will be considered instead of multiple processes, and shared memory instead of message passing.

1.2.1 Automatic Parallelization

The ultimate way to satisfy the framework design criteria would be automatic parallelization, where a compiler or static analysis transformation tool converts a sequential application to a

parallel one. This way, the implementer of the application gets the benefits of sequential programming such as simplicity, re-usability, and safety while also gaining the benefit of improved performance through concurrency. Automatic parallelization, however, is currently more of a research topic than a practical commercial solution, and has two different areas of focus. The first is scientific computing where arrays are used in computations, and the second is pointer/integer computing where languages that involve pointers and references such as C [3], C++ [4] and Java [5] are used. Then within each of these two areas the research efforts are further divided into automatic parallelization at compile time (using static analysis tools) and run time (speculation based).

Automatic parallelization for scientific computing has shown the most promise so far. This is primarily because arrays are more predictable in dependence analysis and because it has been a research topic for a much longer time. Earlier approaches have focused more on parallelizing (many) fine grained loops that can give a reasonable overall performance benefit when aggregated together. However, there has been some recent success parallelizing (fewer) large grain loops that span many procedures using compile time static analysis [6]. Run time automatic parallelization techniques have also shown promise through speculation, although they require customized support at all sites running the software. Hybrid approaches that involve both compile time and run time efforts have also shown some promise, by extracting information at compile time that can be used as an aid for parallelization at run time [7].

Unfortunately, only limited success has been achieved in automatic parallelization for pointer/integer computing. This is because the pointer-chasing induced by most object-oriented and C code doesn't allow the compiler to detect enough places where the code is free from data and control dependences, so that parallelization can occur [8]. There are ongoing research efforts within this area of automatic parallelization however it is still many years away from being viable for general consumption. Since object-oriented programming languages are now the norm for new development, automatic parallelization will not be considered as a suitable candidate for meeting the properties of this thesis.

1.2.2 Fork/Join

At the other end of the spectrum, where the concurrency is explicit, is the traditional fork/join approach. The idea here is that the fork statement creates a new thread that can execute in parallel with the thread where the fork statement was invoked, as shown in the pseudo-code shown in Figure 1.

```
tid1 := fork( c.task1() )
tid2 := fork( c.task2(&d) )
do_something_else

join(tid2)
c.task3(d)
```

Figure 1. Fork and join.

In this code segment, two additional threads are to be created and run in parallel with the thread from which they were called, which means task1( ), task2( ), and do_something_else can run in parallel. Tasks invoked by forking, are done so asynchronously, and cannot return a result in the traditional way. The return mechanism can be simulated by supplying a reference or pointer variable as a function parameter to the task being forked, as is done with task2( ) with variable ''d'. The actual fork routine does return a thread ID just before invoking a task, but it is not related to the return mechanism of the task itself.

A forked task can mimic a traditional synchronous call (although without the traditional return mechanism) by placing a join statement immediately following the fork of the desired task, and by specifying the thread ID returned by the relevant fork as a join parameter. If the join statement does not immediately follow the fork, then the task can behave both asynchronously (with the code between the fork and join) and synchronously (with code after the join). In this example, the join statement will block the calling thread until the thread in which task2( ) runs has terminated, most likely to make sure that the value of variable ‘d’ has been computed before it is used in task3( ). The join operation could also provide the necessary synchronization on the state of object ‘c’ or any common global data (not evident by code inspection), if accessed between task2( ) and task3( ).

One important drawback of the fork/join approach is that there are still opportunities for race conditions to occur, even with the join statement. The join operation only preserves the desired ordering between task2( ) and task3( ) and nothing else. For example, race conditions are possible if simultaneous access occurs on;

- The state of object 'c' in task1( ), task2( ), and/or do_something_else.
- Variable 'd' in task1( ) and do_something_else.
- Global_data in task1( ), task2( ), and/or do_something else.

Listed below is a summary of the potential problems that can occur when concurrency is introduced into an application using the fork/join approach:

- Race conditions are possible on the following:
    - Returned values (explicit or implicit) used before they have actually been computed. Assumptions cannot be made about a task's relative computation time.
    - Aliased function parameters such as references or pointers.
    - Global data.
    - Public data members in object oriented languages, since they will have the same problem as global data, and should not be allowed.
    - Modification of an object's state. Note that this cannot be determined by examining the member function's interface.
- The join operation cannot be called on the same thread more than once.
- Deadlock and/or starvation are possible.

Surprisingly, even with this very small and simple code segment, there are many things that could go wrong, which means that the developer must be extremely careful when designing, implementing, and modifying parallel code. As the parallel code segment grows in size and additional developers become involved in modifying the code, keeping the code free from race conditions and deadlock becomes a very difficult task. A different approach is needed.

Drawn from the list above and listed below, is a summary of the potential problems that all of the concurrency approaches discussed in this section will share, so that it does not need to be repeated for each approach:

- Race conditions are possible on the following:
    - Aliased function parameters such as references or pointers.
    - Global data.
    - Public data members in object oriented languages, since they will have the same problem as global data, and should not be allowed.
    - Modification of an object's state. Note that this cannot be determined by examining the member function's interface.
- Deadlock and/or starvation are possible.

1.2.3 Co-begin

The co-begin approach exists in languages such as Algol 68 [9] and was first introduced by Dijkstra [10]. Each statement within the co-begin block is created in its own thread and can execute concurrently with others within the block. The co-begin construct behaves as though a fork is invoked on each statement within the block and a join invoked on the thread ID returned from each fork immediately after the block. Therefore a co-begin forces an implicit synchronization on all concurrently executing threads within the block as shown in Figure 2.

```
par begin
  c.task1()
  b := c.task2(&d)
end
do_something_else

c.task3(b,d)
```

Figure 2. Co-begin.

In this code segment, task1( ) and task2( ) would run in parallel until both of them complete, before control would return to the calling thread just after the “end” statement. Since do_something_else and task3( ) appear outside of the block, they cannot start execution until the

other two tasks complete. This is necessary for task3( ) because it cannot use the explicit return value ‘b’ or the implicit return value ‘d’ until they are computed by task2( ). As with the fork/join example, race conditions are still possible on the state of object ‘c’ or common global data between task1( ) and task2( ), or even between do_something_else and task3( ).

The differences over the fork/join approach are mostly cosmetic, since the developer on the client side must still be aware of what can run concurrently and what can’t. For example, if there are two activities that access the same data, they must be separated by placing one inside the co-begin block and the other outside or both can reside in separate co-begin blocks. However, not having to worry about inserting multiple join operations (with appropriate thread ID) for task synchronization is a plus for code maintainability when using the co-begin approach.

1.2.4 Futures

Futures is available as a language construct in Multilisp [11]. The main point of this approach is to have the programming language allow a return variable to be declared with a special “future” type, so that it can be used at any point in the future if its value has already been computed, otherwise all threads that try to access the un-computed future will block. Figure 3 shows a pseudo-code segment that illustrates how Futures could be used.

```
a,b : future int
a := c.task1()  &
b := c.task2(&d)  &
do_something_else

c.task3(a,b,d)
```

Figure 3. Futures.

The & notation is used to indicate that the task1( ) and task2( ) member functions are called asynchronously, each within their own thread. Notice that for variables 'a' and 'b', explicit synchronization such as a join operation is unnecessary, before the values are used in task3( ), since it is implicitly handled by the future variables. However, in order to insure that variable 'd' is not accessed concurrently by task2( ) and task3( ), a join operation would be required before

task3( ). As with the fork/join example, race conditions are still possible on the state of object ‘c’ or common global data between task1( ) and task2( ), or do_something_else.

Like the fork/join and co-begin approaches, the concurrency is explicit on the client side, since a future variable must be declared at the calling site. Moreover, explicit synchronization primitives (e.g. semaphores, joins) may be needed for any race conditions that the Futures approach is unable to handle.

1.2.5 Wait by Necessity

An attempt to resolve the problem of using a returned result of a concurrent task before it is computed is made by the “wait by necessity” approach, which is supported in the concurrent languages, Mentat [12] and Oz [13]. In this approach, asynchronous function calls are allowed to return values, presumably at the end of the function body. Static analysis is then performed to locate where the return value is first used after the function call, and when found, a join equivalent operation is inserted just before (via a transformation).

This approach focuses only on the problem of using an explicitly returned value before it is computed, so all of the other problems that exist with the fork/join approach still exist. Moreover, the approach still has two critical issues with regard to the return value. The first is that in more complex situations, the data dependence analysis can break down, resulting in a more conservative analysis that inserts the join operation just after the function call, essentially making it a traditional synchronous call. In this case, there would be a loss of performance, because of the increased thread overhead. The second issue is that a specialized compiler or static analysis tool would be required to perform the transformations. It should be noted that “wait by necessity” has essentially the same expressive power as Futures, but may be worse in cases where static analysis is unable to resolve the data dependences.

1.2.6 Active Objects

Another approach is that of active objects [14]. The term “active objects” is overloaded in the literature, and in this case it refers to autonomous objects that communicate asynchronously and can be run concurrently with other code. Each object is created within its own thread and is

allowed only to do one thing at a time. Immediately after an active object is created, it behaves much like a server, where it accepts incoming requests. The active object can control which request to act upon based on its current state. Concurrency is achieved by requiring all calls to functions of active objects to be asynchronous. If a return value is desired, then this can be constructed with a pair of asynchronous calls (a request and a reply).

Compared to the other approaches, there would be some additional overhead, because an object must be created for each concurrent function call. This could lead to the creation of many more objects than traditional concurrency approaches. Active objects take a step in the right direction by using objects to encapsulate data that may be accessed simultaneously. However, unless all the data is local to the active objects, data race conditions could still occur. Because function calls to active objects are asynchronous, pairs of request and reply calls would be needed each time a return result is desired, which would make the parallelism explicit on the client side, and create even more overhead. The active object model is better suited for multiprocessor systems, without shared memory where message passing is required for communication.

1.2.7 Early-Reply

The basic idea of Early-Reply, which can also be referred to as “early-return” or “post-processing”, is not new and exists in many forms. For example, in computer architecture instruction set pipelining can be thought of as an Early-Reply mechanism, where the result computed by the ALU for a given instruction can be forwarded early to another later instruction that needs the result as its input, preventing this later instruction from stalling so it can execute in parallel while the first finishes its memory access and register write-back cycle. We will see later that Early-Reply is really the software version of hardware pipelining.

Early-reply in software is based on the idea that functions can produce their final return values well before the function actually terminates. Examples that illustrate this behavior include functions that maintain a balanced binary tree, and functions that sort a collection of elements. In the balanced tree example, an insertion function can return a pointer to the inserted element immediately after it is created, allowing the remainder portion to execute concurrently on another thread to determine where in the tree the new element should be placed and perform any

subsequent tree rebalancing. In the sort example, the sort function can return void almost immediately after it is called, allowing the remainder portion to perform the actual sort on the collection of data concurrently on another thread. In both examples, a performance gain is possible as long as the data processed in the remainder portion is not accessed by the caller immediately after the Early-Reply call. Early-reply is then a mechanism to forward final return values to the caller as soon as they are safely available in a synchronous fashion, allowing the client side (caller) to proceed concurrently with the remainder code of the called function.

Like Futures, Early-Reply is an attempt to get the best of both worlds; concurrency with synchronous-like function calls. However, there are two major differences. The first difference is that at the point where the function returns, the return value of an Early-Reply function call is ready to be used. Whereas in Futures, there is no guarantee that the return value has actually been computed yet, which could force any subsequent statement that uses it to block. The second difference is that the concurrency is not evident at the call site for an Early-Reply function. Whereas with Futures, there must be some language construct to indicate that either the return value or the function itself is a future (i.e. indicating concurrency). These two differences can be illustrated in the example code segment in Figure 4.

```
a := c.task1()
b := c.task2(&d)
do_something_else

c.task3(a)
```

Figure 4. Early-Reply.

As in the previous examples for the other approaches, task1( ) and task2( ) are the functions where the concurrency will be initiated (in this case Early-Reply), but notice there isn't any way to differentiate functions involved in concurrency from those that are not (i.e. task3( )), since everything looks sequential. In addition, the return value of task1( ), stored in variable 'a', can be used in any subsequent statement, since the return value is valid as soon as task1() returns (unlike futures). Because of the absence of explicit concurrency on the caller's side (even though it exists), users of Early-Reply functions can program and reason about their code in a simpler and more maintainable sequential style. We feel that this is what separates this approach

apart from the other concurrency approaches. Pike and Sridhar, in their paper, “Early-Reply Components: Concurrent Execution with Sequential Reasoning” [15], recognized this benefit and discuss why a closer look at Early-Reply as a concurrency mechanism is desirable.

One drawback that Early-Reply has that the other approaches do not have, is that the concurrency is bounded by how soon the Early-Reply function can return relative to remaining code to be executed. Because of the synchronous nature of an Early-Reply call, concurrency cannot be realized until the function returns. Most of the other approaches for initiating concurrency, become concurrent (i.e. asynchronous) either immediately or almost immediately after the function is called.

Race conditions and deadlock are possible with Early-Reply, just as with the other approaches. A special race condition can occur on the return value from the Early-Reply function, if it is modified after it is returned while still in the body of the Early-Reply function. However, we will see later that Early-Reply provides a good basis to make safety and liveness error conditions easier to manage.

Scott and Lea briefly discuss the benefits of Early-Reply in their books, “Programming Language Pragmatics” [16] and “Concurrent Programming in Java” [17] respectively. Scott mentions that much of the motivation for Early-Reply is to satisfy the parent thread that each newly created thread has been initialized properly on behalf of Early-Reply function call before it (the parent) continues execution. Lea points out that Early-Reply is ideal for time consuming functions that do not return anything.

Early-reply does appear as an explicit language construct in several concurrent programming languages such as SR [18] and Lynx [19]. For example with SR, an Early-Reply function definition would have an explicit “reply” statement where it is desired to return early, as shown in Figure 5.

```
int task1()
{
    /*  code to compute return value */
    reply
    /*  code that runs concurrently with caller */
}
```

Figure 5. Early-Reply as a return statement.

At the point of the "reply" statement, the return value is returned, and a new thread is created to run concurrently with caller. The "reply" statement is not visible to the caller as part of the function's interface, since it is in the function's definition. Pike and Sridhar [15] suggested that Early-Reply could eventually become a first class language construct for the more popular object-oriented programming languages, and could be inserted simply as a programming language statement where the function should return early as just shown for SR. In languages where Early-Reply is not available as a language construct, such as C++, Java, or C#, it can be mimicked. A brief example of how it can be mimicked in Java is shown in Lea's text, "Concurrent Programming in Java" [17].

1.3 Contribution

Concurrency can be a very effective technique to boost the throughput or response time of a software application, but unfortunately, it comes with a suite of problems that prevent many developers from using it effectively. That said, the primary objective for this work is to come up with a concurrency approach that can significantly improve an application's performance, while also reducing the complexity of parallel programming so that more developers can enjoy its benefits. As Pike and Sridhar pointed out [15], Early-Reply possesses some special properties that make it possible for the concurrency to be implicit to the caller, which in turn gives developers the ability to design and reason about their programs using the simpler and more familiar sequential model. We feel that this makes Early-Reply well suited to be used as a basis of a more comprehensive framework that we will develop to meet our primary objective.

The important contribution of this thesis is to expand on Pike and Sridhar's discussion on the special properties of Early-Reply, by developing an Early-Reply based framework that imparts a

structure that can be used in various ways to help reduce the complexity of parallel programming. This structure facilitates the development of a simple set of coding guidelines that can be used to prevent safety and liveness error conditions such as race conditions and deadlock respectively from occurring. The developer will use the guidelines when converting original class definitions into Early-Reply class definitions, so that race conditions and deadlock do not occur in the first place. A static analysis tool can use the guidelines to verify the absence of these error conditions. This use is particularly relevant because parallel programs are typically too complex for any meaningful verification on safety and liveness error conditions. The framework's structure can also be used to implement a simulation program that predicts possible performance gains for an arbitrary application, given the overhead of the framework's primitives.

Listed below are the contributions provided by this work that will hopefully make this approach a success:

1) Analyze the raw Early-Reply mechanism in more detail to determine why it is well suited for a more comprehensive framework.
2) Design/implement a minimal framework based on Early-Reply that imparts a structure that will:
    a. Provide a simple way to improve the performance of an arbitrary application through concurrency.
    b. Facilitate the development of some simple guidelines that are to be followed when coding to prevent race conditions and deadlock from occurring.
    c. Make it as easier to implement a simulation program that predicts possible performance gains for an arbitrary application (if any at all), given the overhead of the framework's primitives.
3) Select several real-world sample applications and apply the framework and measure the results, so there will be tangible proof that the Early-Reply mechanism can really improve performance.
4) Illustrate how the coding guidelines can be used by a static analysis tool to verify the absence of race conditions and deadlock.

5) Design/implement a simulation program to predict possible performance benefits of arbitrary applications if they use Early-Reply.

2 ANALYSIS OF THE EARLY-REPLY FRAMEWORK

Early-reply in its raw form has a number of drawbacks, and is not sufficient on its own as a robust solution for concurrency. The most critical drawback to resolve is a lack of a reliable synchronization mechanism that will prevent race conditions from occurring. This situation is common among many of the other mechanisms that focus on initiating concurrency. Many of Early-Reply's drawbacks can be resolved by designing and implementing Early-Reply in the context of a comprehensive framework based on Early-Reply. This section will discuss the design criteria of the framework, analyze in more detail the raw Early-Reply mechanism, and examine the design and implementation of this framework.

2.1 Framework Design Goals

Developing concurrent programs can be very complex and error prone (even for those experienced in concurrency). Consequently, this work will focus on a better way to develop concurrent programs by achieving the following three general goals:

- Provide a simpler way to write concurrent programs.
- Prevent the occurrence of safety and liveness error conditions, such as data race conditions, deadlock, and starvation.
- Exploit enough concurrency so that sufficient performance gains can be realized.

It is important to note, that a "one size fits all" solution to concurrent programming is extremely difficult. As such, this approach focuses only on the class of applications that exhibit "task parallelism". "Task parallelism" involves different independent tasks simultaneously operating on different data. This contrasts with task parallelism's counterpart of "data parallelism", where the same task operates on different parts of the data.

Listed below is the complete set of design criteria to guide the design of the Early-Reply Framework and will be subsequently discussed in more detail:

1. usability
2. maintainability
3. portability
4. safety
5. liveness
6. performance
7. predictability

The first three design criteria of usability, maintainability, and portability are used to provide a simpler way to write concurrent programs. Usability can be improved by allowing the simpler and more familiar sequential programming model to be used to develop programs with concurrent execution. For the sequential model to be possible, all parallel programming constructs such as locks and threads, must be encapsulated away.

Program maintainability can be enhanced by Early-Reply function calls retaining the same behavior and appearance as if they were called in a typical sequential program. Proof obligations of the correctness of client code calling Early-Reply functions would then be no different than for sequential programs. This greatly simplifies any future modifications to client code and may even make debugging easier than with traditional concurrent programs.

Concurrent programs are typically dependent upon a particular concurrency library, operating system, and/or computer architecture. As such, the framework will be constructed with "portability" in mind, so that if any of the before mentioned change, the interface / implementation of the client code along with the framework's interface would not require modification. Moreover, other than for performance reasons, the framework will not be dependent upon the number of processors available on a particular machine.

The framework must prevent certain "safety" and "liveness" error conditions such as race conditions and deadlock respectively from occurring in the first place. These error conditions are unique to concurrent programming, and are the primary reason it is so error prone. The framework's structure can be used to develop a list of coding guidelines that, if followed, will

prevent race conditions and deadlock from occurring. These guidelines are discussed in section 4.2.2. Moreover, this same list can be used by a static analysis tool to guarantee the absence of these error conditions by verifying that these guidelines have been followed.

The last two design criteria of performance and predictability are used to ensure that enough asynchrony is possible so that sufficient performance gains can be realized. Improving program performance is understandably the primary motivation for using concurrency, so the framework must be able to exploit enough concurrency so that a sufficient performance gain can be realized. "Sufficient" is stressed instead of "optimal", because the focus of this thesis is finding a more accessible approach to using concurrency for performance gains. The framework must provide enough asynchrony while also minimizing the concurrency overhead, so that a sufficient performance gain can be observed. Depending upon the application, the desired performance gain could be an improvement in either throughput (faster execution time), or an improvement in response time for interactive programs (shorter wait time over a small segment of code), or both.

Since a concurrent program does not always equate to a better performing program, "predictability" of the potential performance gain should be possible. The framework should provide a simple enough structure so that a simulation program can be developed to determine if a performance gain is possible, through Early-Reply, for a particular application. It should be possible to simulate the calling sequences and their respective execution times along with the framework's concurrency overhead.

2.2 Early-Reply in Its Raw Form

In this section, Early-Reply in its raw form will be analyzed to show why it can be a basis of a more comprehensive framework that uses concurrency to improve a program's performance, while also making concurrency simpler and more reliable. Both its benefits and drawbacks will be discussed, so that it can be better integrated into the framework design. In this section, Early-Reply will be examined as a freestanding function (rather than a member function), so it can be looked at in its purest form.

2.2.1 Definition of Early-Reply

As a freestanding function, Early-Reply can be thought of as a function broken up into two parts:

- A material segment composed of a sequence of statements up to and including the return statement that execute synchronously on the same thread as the caller.
- A residual segment composed of the function's remaining statements that execute asynchronously on a different thread.

Note that the terms "material segment" and "residual segment" could just as easily be replaced by "prefix segment" and "suffix segment", but we choose to be consistent with Pike and Sridhar's terminology [15] since our work is an extension of their efforts.

At the end of the material segment, a final return value can be forwarded to the caller (if safely available) or the function can simply return, and in either case the residual segment will then execute on another thread concurrently with the caller's thread. This is illustrated in Figure 6.

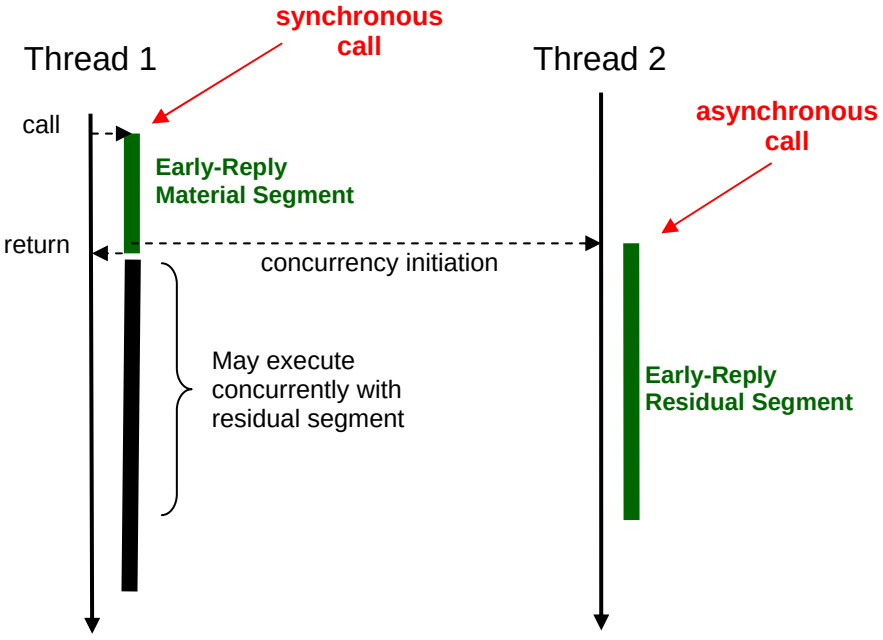

Figure 6. Early-Reply's material and residual segment.

As explained in the “Related Work and Background” section, Early-Reply is based on the idea that functions can produce their final return values well before the function actually terminates. Examples were given such as functions that remove an element from a balanced binary tree or sort data passed to it. These functions are candidates for Early-Reply since there is a sufficient amount of work to do in the residual segment and little or no work done in the material segment, which increases the potential for concurrency by allowing the residual segment to execute in parallel with the code following the Early-Reply call.

To illustrate the basic behavior of Early-Reply, a simple example written in C++ is shown in Figure 7.

```
// Early-Reply function
int foo(float a, int& b)
{
   int c = material_foo(a, b);  // synchronous call
   residual_foo(a, b);          // asynchronous call
   return c;
}

// client code
int main(int argc, char* args[])
{
   for (int i = 0; i < 20; i++) {
      float a = random();
      int c = foo(a, b);
      bar(c);
      foo_bar(b);
   }
}
```

Figure 7. Simple example of Early-Reply code.

In this example, there is one Early-Reply function foo() and three Non-Early-Reply functions that are called within the loop. Notice the client code is completely absent of concurrency specific function calls and/or data types. Moreover, the client code is not dependent upon any specific concurrency library, operating system, or computer architecture. The return value of foo() can be safely used immediately after the call, just as any other synchronous function call. Within the definition of foo(), two threads of execution are used; the primary thread where the material segment of foo() and the other functions execute, and a secondary thread where the

residual segment of foo() executes. The opportunity for concurrency occurs if bar() can safely execute at the same time as the residual segment of foo(). More concurrency can be exploited if the execution time of foo's residual segment is longer than that of bar()'s, and even more if foo_bar() and random() can also safely execute at the same time as foo's residual segment. An optimal performance gain is observed when the sum of the execution times of bar(), foo_bar(), random(), and the material segment of foo() equal the execution time of the foo's residual segment.

To get a better appreciation for possible performance gains with Early-Reply, the execution times for one call of each function in the example are listed in Figure 8. Overhead for the Early-Reply mechanism is ignored.

```
2 units - foo (material segment)
8 units - foo (residual segment)
4 units - bar
2 units - foo_bar
2 units - random
```

Figure 8. Time units of execution in Early-Reply program.

For 20 iterations the total execution time takes 200 time units with Early-Reply, and 400 time units without Early-Reply. Over just this loop segment, this is a speedup of 2 assuming that two unloaded processors are available. Even though this case is not likely to happen, it does illustrate the potential to achieve significant performance gains while keeping the client side unaware of the concurrency.

As with other concurrency mechanisms, there are constraints that must be met in order to realize true performance gains. Early-reply has two lower bound constraints:

- Computation in the residual segment must exceed the concurrency overhead of getting the residual segment to execute on another thread.
- There must be some computation that immediately follows the Early-Reply call on the calling thread that can execute safely and simultaneously with that of the residual segment the another thread. Moreover, this computation must exceed the residual segment overhead.

It should be noted that for the second constraint, average user “idle time” could be substituted for computation when interactive programs are considered. Only if both of these constraints are met, can it be worthwhile to make use of Early-Reply.

However, there are some potential problems. If for example foo_bar() reads the value from the variable 'b' before foo()'s residual segment writes to 'b', a race condition ensues. In order to prevent this race condition from occurring, access to the common data 'b' must be synchronized by some kind of locking mechanism in both foo() and foo_bar(), as shown in Figure 9.

```
Mutex mutex;

// Early-Reply function
int foo(float a, int& b)
{
   lock(&mutex);
   int c = material_foo(a, b);
   residual_foo(a, b);      // asynchronous function call
   return c;
}
void residual_foo(float a, int& b)
{
   /*  body */
   unlock(&mutex);  // since asynchronous, need unlock() here
   return;
}
void foo_bar(int& b)
{
   lock(&mutex);
   /* body * /
   unlock(&mutex);
}
```

Figure 9. Synchronization primitives within implementation of Early-Reply function.

This puts a great burden on the developer to keep track of all the different functions that access variable 'b', so that lock/unlock pairs can be inserted in the appropriate places. In this example, it is foo_bar() that must have synchronized access to variable 'b', but there could be many others as the program is continually modified over time.

An alternative synchronization approach is to insert the locking primitives in the client code instead of inside foo() and foo_bar(), as shown in Figure 10.

```
// driver code
int main(int argc, char* args[])
{
   for (int i = 0; i < 20; i++) {
      float a = random();
      int c = foo(a, b);
      bar(c);
      lock(&mutex);
      foo_bar(b);
      unlock(&mutex);
   }
}
```

Figure 10. Synchronization primitives within client code.

With this approach, the developer must still keep track of all the accesses to variable 'b' (as before), but now has the additional drawback of exposing the concurrency to the caller, which is intrusive. Moreover, for either approach, as the number of inserted lock/unlock pairs grow, so does the likelihood of deadlock occurring.

2.2.2 Benefits of Early-Reply

The key characteristic that differentiates Early-Reply from many other concurrency mechanisms is that it behaves as both a synchronous and asynchronous function call. From the caller's perspective, Early-Reply behaves as a standard synchronous call, allowing the simpler sequential calling model to be used on the caller's side. The caller waits until the Early-Reply function returns before resuming execution, and the return value can be used immediately when it does return. Moreover, no special concurrency types or qualifiers are declared or used, which when combined with the synchronous behavior, keeps the caller oblivious to the concurrency that is occurring.

On the other hand, asynchronous behavior is required to enable the concurrency, and fortunately it can be hidden from the caller inside the implementation of the Early-Reply function. Concurrent execution occurs just after the residual segment is invoked inside the Early-Reply

function. Once invoked, the residual segment executes simultaneously on another thread with the code following the Early-Reply call on the caller's side.

This dual synchronous and asynchronous behavior makes it possible to develop programs using the more familiar and simpler sequential programming model, while also achieving performance gains through concurrent execution. Pike and Sridhar [15] pointed this out and argued that programming properties such as usability and maintainability could be improved by using Early-Reply over other concurrency mechanisms. Listed below is a summary of the benefits that Early-Reply in its raw form provides:

- A return value is usable immediately after an Early-Reply function call.
- Through its residual segment, Early-Reply provides the critical asynchrony that is needed to achieve a performance gain.
- Sequential reasoning can be used to design and implement programs using Early-Reply, because an Early-Reply function call behaves like a typical sequential call. This is because its return value can be used immediately after the function returns, and because there are not any special concurrency data types or qualifiers used in the call.
- Modifications to client code that involve Early-Reply can be done more easily than other concurrency approaches, since proof obligations of the correctness of the client code are no different from a typical sequential program.
- Programs using Early-Reply can be made more portable, since the client code is absent of concurrency specific function calls and/or data types. This by default makes less code dependent upon any specific architecture, operating system, or concurrency library.

2.2.3 Drawbacks of Early-Reply

Despite the benefits that make Early-Reply an attractive approach, Early-Reply in its raw form does come with several drawbacks. The most important drawback is that it does not provide a synchronization mechanism to prevent data race conditions from occurring. This is not unique to Early-Reply, as many other concurrency initiation mechanisms do not provide one either. It is a problem that must be addressed if concurrency is to be used reliably. Race conditions can

occur whenever the residual segment and the caller execute concurrently. Listed below are three general cases where data race conditions are possible:

- Return value is modified within the residual segment before its return value is used in the client code following the Early-Reply call.
- A function parameter is passed by pointer or reference in an Early-Reply call and the value is accessed simultaneously inside the residual segment and by the client after the Early-Reply call.
- Global data is accessed simultaneously inside the residual segment and by the client after the Early-Reply call.

Because of the lack of a synchronization mechanism, users of Early-Reply must think a lot about the concurrency taking place in order to keep their programs safe. This goes against one of our primary goals of making concurrency simpler by minimizing the amount of concurrency reasoning a user of Early-Reply must perform.

Another important restriction of Early-Reply (not necessarily a negative) is that it is focused specifically on programs that can execute independent tasks concurrently. For example, programs that can make use of "data parallelism", where one algorithm operates in lockstep on different parts of data that are split up over many different threads, may not be suitable for Early-Reply.

Since the concurrency begins at the end of the material segment, and not at the beginning of the function call, it could be perceived that the potential "performance" is not as good as some of the other concurrency mechanisms. However, the material segment is allowed to be empty, which means the parallelism can for all practical purposes begin as soon as the Early-Reply function is called. Listed below is a summary of the drawbacks of Early-Reply in its raw form:

- No synchronization mechanism is provided to prevent data race conditions from occurring, making the program's safety tenuous, and forcing the user of Early-Reply to be cognizant of the concurrency taking place.

- Early-reply does not exist yet as a language construct or an as an accepted implementation standard in any of the popular programming languages. This makes it vulnerable to being used incorrectly, leading to a poorly performing or unsafe program.
- In some cases it may be difficult to separate the body of an Early-Reply function into its material and residual segments. Care must be taken so that the function does not return too soon, but the sooner it returns, the better the opportunity for increased performance. This requires separating the data flow from the control flow.
- Early-reply is not a solution for all concurrent programming.
- As with other concurrency mechanisms, it may be difficult to predict potential performance gains. The concurrency overhead and code that could be executed concurrently must be considered somehow.

2.3 Issues to Resolve

Early-reply in its raw form still has a number of drawbacks and is not sufficient as a robust solution for concurrency. It does have the attractive feature of initiating the concurrency without being visible to the caller. Unfortunately, it does not provide any concurrency synchronization to prevent race conditions from occurring. A race condition is the most important error condition to prevent that is caused by concurrency since it can lead to a program that produces incorrect results. Consequently synchronizing the concurrency to prevent race conditions will dominate most of the framework's structure. In this sub-section, we examine what is needed to augment Early-Reply in its raw form so that a comprehensive framework can be constructed that satisfies our framework design goals.

2.3.1 Concurrency Synchronization and Race Conditions

A safety condition is defined as a condition that must hold in every finite prefix of the sequence. Informally, this means that nothing bad has happened [20].

As just discussed, Early-Reply in its raw form does not provide any kind of synchronization capability to control simultaneous access to shared resources. To prevent race conditions from

occurring, raw Early-Reply must be augmented with some sort of synchronization mechanism. However, providing a weak synchronization mechanism that only works in some cases (as do several other concurrency initiation mechanisms) may be worse than not providing one at all. This is because we can now design a better synchronization mechanism from the ground up that is customized to our specific framework.

To set the groundwork, we must first consider the two general synchronization categories; cooperation synchronization and competition synchronization [1]. Cooperation synchronization is needed when one task must wait for another to complete some specific activity before the other can begin its execution. The producer-consumer problem is an example application that needs this type of synchronization, because a producer task must have already produced something before the consumer task can consume it. A consumer task is never allowed to proceed unless there is something waiting for it to consume.

Competition synchronization, on the other hand, is when two or more tasks require the use of a resource that cannot be simultaneously used, such as a data field. The mutual exclusion problem is an example that needs this type of synchronization, since only one task can be allowed access to the common resource at one time. It has the potential to be a more fine grained synchronization than cooperation synchronization, because tasks can partially execute before being locked out by another earlier task. After the access of the locked resource is complete, the earlier task releases the lock, allowing the other task to resume execution.

The semaphore is a well known synchronization mechanism that can be used for both cooperation and competition synchronization, and involves two primitives “wait” and “signal” [10]. It is typically used for competition synchronization. However, it has proved over time to be very error prone. Forgetting to place a “wait” where it is necessary can cause a race condition to occur, and neglecting a necessary “signal” can lead to deadlock.

Another well known synchronization mechanism that can also be used for both varieties of synchronization is a monitor [21]. A monitor encapsulates the shared resource and forces all accesses to the shared resource (i.e. function calls) to go through the monitor first, so that all function calls that access the shared resource will be serialized. This means that a monitor can

be used simultaneously for both cooperation and competition synchronization. The competition synchronization is intrinsic, and cooperation synchronization is observed since once a function call is allowed access to the monitor it runs until completion before another can start. Below is the definition of a typical monitor [22]:

- Only one thread can be active within a monitor at a time.
- Data local to the monitor can only be accessed through the monitor's procedures.
- Procedures of a monitor can only access data local to the monitor (i.e. cannot access an outside variable).
- Before a thread is active within a monitor it must first issue a wait call to check if another thread is already active. If so, the thread blocks and is put on a wait queue. If not, the thread's procedure begins execution.
- After a thread's procedure is finished executing within the monitor it issues a signal call that activates the procedure of the next thread in the wait queue.
- Wait queues can be implemented as priority queues (although can cause starvation).
- Threads are created and assigned outside of the monitor.

Although not perfect, the monitor is a close match with our design criteria for making the framework's synchronization mechanism simple and well behaved. If object-oriented programming (OOP) is used, the shared resource to protect could be the state of an entire object (i.e. data members), rather than arbitrary data fields. This makes synchronization much simpler and less error prone because synchronization controls are only needed just before and possibly just after each member function call. Any set of functions that may need simultaneous access to the same shared data should be made member functions of the same object. It is important to note that if procedures of a monitor are allowed to access data that is not local to the procedure or the object, race conditions are possible because there would be no control on simultaneous access. This is why any data that can be accessed simultaneously is to be made local to the object and the access to be controlled by the monitor. Moreover, since all the required synchronization is completely encapsulated by the monitor, the usability and maintainability of client side code will be improved. As such, the monitor will be used as a basis for our framework's synchronization mechanism with some modifications.

However, some of the features from the typical monitor definition may adversely affect our design criteria, and they are listed below:

- Creating/assigning threads outside of the monitor makes it more difficult to hide the concurrency from the caller.
- A monitor must be capable of handling multiple simultaneous entry attempts, since thread creation/assignment is outside of the monitor. This means that wait queues are necessary, which could introduce unnecessary performance/space overhead.
- If priority wait queues are used, starvation could occur.
- Only one thread can be active in a monitor at a time, which limits the amount of concurrency that can be exploited, and thus the performance gain.

2.3.2 Concurrency Synchronization and Deadlock/Starvation

An unfortunate side effect of concurrency synchronization is that as soon as it is introduced into a program, liveness error conditions such as deadlock and starvation become possible. Since they do not occur before synchronization is introduced, we would like the framework to make their occurrence less likely, or even impossible to occur.

Deadlock is typically the most common liveness error condition to occur in a concurrent program. Deadlock is a situation where a process or a set of processes is blocked, waiting on an event that will never occur. The following four conditions are all necessary for deadlock to occur [22]:

- Process requests exclusive access of resources.
- Processes hold previously acquired resources while waiting for additional resources.
- A resource cannot be preempted from a process without aborting the process.
- There exists a set of blocked process involved in a circular wait.

Starvation occurs when a process waits for a resource that continually becomes available but is never assigned to the process because other processes with higher priority continually preempt it or there is a flaw in the scheduler [22].

2.3.3 Concurrency Initiation

Early-reply in its raw form provides a good basis for initiating concurrency because it is initiated inside the function implementation and thus hidden from the caller. However, there are still two primary issues of concern with respect to concurrency initiation; enough concurrency must be exploited for sufficient performance gains, and the initiation must not make synchronization of the concurrency more difficult.

The opportunity for concurrency begins as soon as the residual segment is invoked on another thread that has just been created or reused. The faster the residual segment can actually start the better the performance. Creating a new thread each time a residual segment is invoked will incur too much overhead, and thus a thread pool will be used instead. Thread pools create a number of new threads at startup, and then assign an unused existing thread to a residual segment as it is about to start execution, rather than create a new thread each time. If all the pool's threads are in use, there must be some mechanism to block until a thread frees up.

Another possible way to exploit more concurrency with respect to concurrency initiation is by having an Early-Reply call within another Early-Reply call. This is called fan-out and is best when the second Early-Reply call is made with the residual segment of the first Early-Reply call. Early-reply fan-out will be discussed in more detail later.

One of the problems of a traditional monitor approach that was discussed earlier is that the threads are created or assigned outside of the monitor when the concurrency is initiated. This makes the concurrency more difficult to hide from the caller and complicates the synchronization mechanism by allowing an unlimited number of threads to access the monitor simultaneously. The monitor must then support a queuing mechanism to support all of the incoming threads. We will look at a way to initiate the concurrency inside the monitor to help hide the concurrency from the caller and to limit the number of threads that access the monitor simultaneously.

2.3.4 Object-Oriented Paradigm

The earlier work by Pike and Sridhar [15] discussed Early-Reply within the context of an object-oriented approach that encapsulates shared data inside an object and serializes any access to that data. We agree with this approach and will expand on this work with an actual OOP implementation. Consequently, the framework will be written using a general purpose object-oriented programming language so that object-oriented features such as abstraction, encapsulation, object aggregation, and polymorphism can be leveraged to produce a more effective and robust concurrency framework. Listed below are some of the tangible benefits:

- A monitor-like synchronization mechanism can be constructed by encapsulating any data that can be simultaneously accessed by a group of related functions into one object. Encapsulation can then provide controlled access to the shared data through monitor entry checks and by making data member access private. This means that concurrency synchronization is implicit in the client code.
- By also encapsulating the concurrency initiation mechanism into this monitor-like construct, the concurrency will be completely hidden from the client, which enables the use of the sequential programming model for client code.
- Portability can be improved by encapsulating the low-level concurrency primitives that depend upon a specific concurrency library into their own object, and only access them through an interface.
- Object aggregation makes it possible for one Early-Reply object to contain another Early-Reply object so that sub-object Early-Reply fan-out is possible to exploit more concurrency, which in turn can enhance performance.

Polymorphism will be shown later on to benefit the usability of the framework, so that the user can simply indicate the appropriate residual segment to execute within the secondary thread.

2.3.5 Early-Reply as a Freestanding or Member Function

Before going further we feel it is important to look at whether or not it would be appropriate to implement Early-Reply as a freestanding function.

The member function approach encapsulates any data that may be simultaneously accessed by a group of related functions into an object, where each member function can access it in a controlled and synchronized manner. The member functions are finite in number and easily located by a developer because they must be part of the class definition. Furthermore, by locking the entire object for accesses by the member functions, only one lock/unlock pair needs to be used per public member function. It will be shown later that this property makes it possible that all data local to the object (i.e. data members and member function local), do not have to be considered for race conditions, because of framework guarantees. This greatly simplifies safety verification by static analysis.

On the other hand, a group of related freestanding functions can only access the same data through references passed in as function parameters or global variables. The shared data can then be accessed by any number of functions, which could prove to be too numerous for the developer to track all of them down. Moreover, there also may be too many different mutex variables to keep track of since there is no way to perform the equivalent of an "object lock" on a stateless freestanding function. These issues make Early-Reply as a freestanding function too error prone for the developer, and too complicated for verification by static analysis. Consequently, Early-Reply will only be considered for member functions in this thesis.

2.4 How It Will Work

2.4.1 Framework Overview

Listed in Figure 11 are several definitions that are helpful in explaining the workings of the framework.

Early-Reply Function – freestanding early-reply function.
Early-Reply Member Function – member function with a material and residual segment.
Non-Early-Reply Member Function – standard synchronous member function.
Early-Reply Class – class that contains at least one early-reply member function.

Figure 11. Helpful framework defintions.

The Early-Reply framework is a minimal framework that consists of the following five participating classes:

- An Early-Reply class
- ER_Coordinator
- ER_ThreadPool
- ER_ThreadEngine
- ER_Task

The Early-Reply class provides the implementation for the desired client functionality so there can be many different Early-Reply classes. By definition each Early-Reply class has at least one Early-Reply member function to exploit concurrency and contains one ER_Coordinator as a data member. The ER_Coordinator class provides the mechanisms to initiate and synchronize the concurrency, and provides the interface to these services. The ER_ThreadPool class is a singleton (one per program) that creates all of the additional threads at startup, assigns these threads to Early-Reply residual segments as they are invoked, and the handles the destruction of the threads when the program is ready to terminate. Each ER_Coordinator contains a pointer to the singleton ER_ThreadPool. The ER_ThreadEngine class manages the execution of residual segments on the thread for which it is responsible. Each ER_ThreadEngine object (one for each additional thread) is stored in a container that the ER_ThreadPool singleton manages. The ER_Task is a helper nested class that is to be derived from, so that it can pass function parameter values to each residual segment, as well store some task specific state. These classes and their relationships are shown in the class diagram in Figure 12.

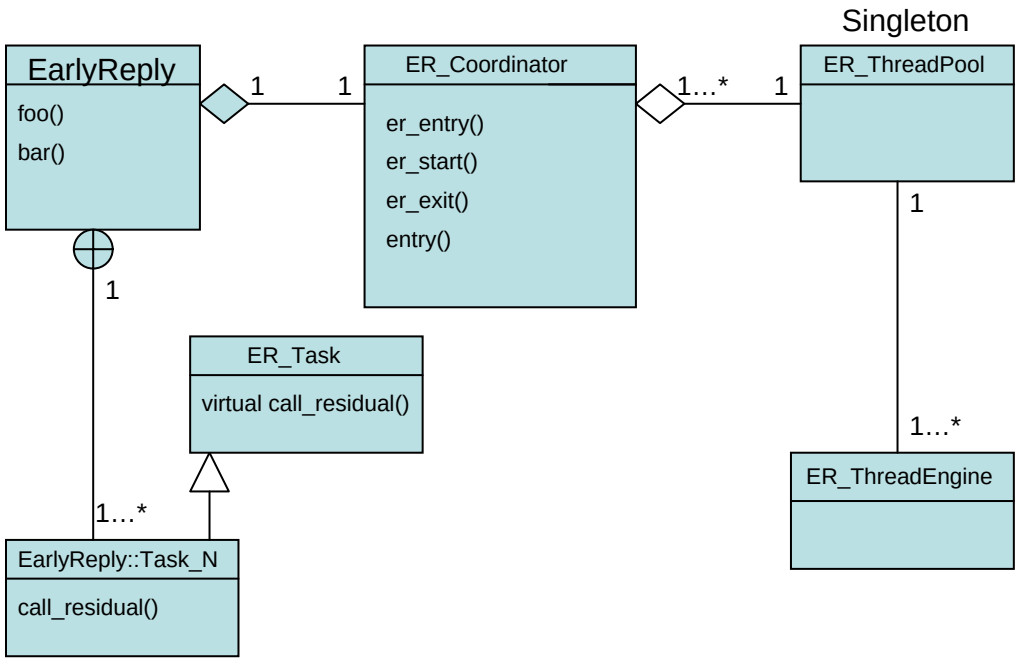

Figure 12. Class diagram of framework.

To simplify further readability, the following aliases will be used: ER_Coordinator as a "coordinator", ER_ThreadPool as a "pool", ER_ThreadEngine as "engine", and ER_Task as a "task". Since the services of the pool and engine objects are not visible to an Early-Reply object, they (along with the task object), will not be discussed further here (see Framework Design and Implementation section).

2.4.2 Combining Concurrency Synchronization and Initiation

As discussed earlier, the monitor will be used as a basis for our framework's synchronization mechanism since it is a close match with our design criteria for making the framework's synchronization mechanism simple and well behaved. However, one important drawback of a monitor is that the concurrency is initiated outside the monitor. This makes the concurrency more difficult to hide from the caller, since an additional thread must be created or assigned before the monitor function call is invoked. As such, our framework instead initiates the concurrency within the monitor, after safely passing through the monitor's synchronization mechanism. This not only helps hide the concurrency from the caller, but also (as we will see

later) limits the number of potential simultaneous accesses on the monitor. This approach is somewhat unconventional, as traditional concurrency mechanisms typically provide one without the other, or a weak version of the other. For example, “fork and join” is an initiation mechanism (fork) with a weak synchronization mechanism (join). Monitors, semaphores, and mutex’s are synchronization mechanisms but do not provide a concurrency initiation mechanism.

To keep things simple, we will follow the traditional monitor definition of allowing only one thread active within the monitor at a time. This makes our synchronization approach that of mutual exclusion, and makes the state of the object the shared data on which to synchronize. OOP’s access privileges can then be used to facilitate controlled access to the object’s state as long as the usual practice of making all data members private or protected is followed. When this practice is followed, only a class’s public member functions are allowed access to an object’s state from outside of the object. This makes it easy to locate all the potential accessors since they must be declared within the class definition. Friend classes and functions are an exception, but they will not be allowed to be declared within an Early-Reply class. Consequently all public member functions of an Early-Reply class will be required to perform some sort of synchronization check as their first step before proceeding.

In our framework, the ER_Coordinator class provides the interface to the services of both concurrency synchronization and initiation. Public member functions of an Early-Reply class will be classified either as Early-Reply or Non-Early-Reply. Early-reply member functions will make use of the coordinator’s er_entry() and er_exit() for synchronization purposes and the coordinator’s er_start() for initiating the concurrency. Since Non-Early-Reply member functions do not initiate concurrency and have less restrictive synchronization requirements, they will only make use of the coordinator’s entry(). An illustrative look of how the concurrency synchronization and initiation is combined and how these services are used by the coordinator class is shown in Figure 13.

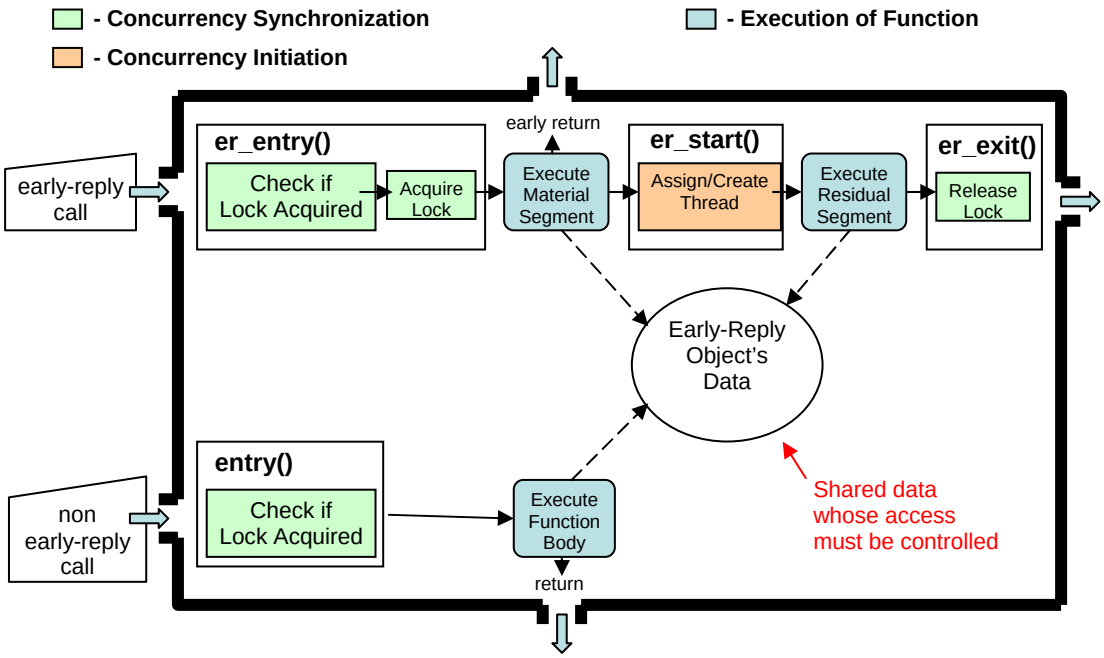

Figure 13. Functionality of ER_Coordinator.

Every Early-Reply member function must first invoke the coordinator's er_entry() within the material segment before proceeding. The er_entry() call checks if the object lock has already been acquired. If so, the Early-Reply function will be blocked until the object lock is released using "suspend and resume" semantics rather than wasting CPU effort with a "busy wait". If the lock has not been acquired, the lock is subsequently acquired and the body of the material segment is executed.

Just before the material segment terminates the coordinator's er_start() is invoked to initiate the concurrency. If available, an auxiliary thread will be assigned by the thread pool on which the residual segment can execute. If one is not available, the Early-Reply call will have to block for a possible second time. It is important to note that up until this point, the Early-Reply call has been executing on the primary thread on which it was originally called. This not only conserves the use of the auxiliary threads, but also releases the caller from any auxiliary thread responsibilities. Once an auxiliary thread has been assigned, the body of the residual segment is executed asynchronously on the auxiliary thread, and thus does not return a value. The last statement in the material segment optionally forwards a return value to the caller (completing the Early-Reply).

At the end of the residual segment execution, the coordinator's er_exit() is invoked to free up the recently used auxiliary thread and to release the object lock. This call is invoked on the auxiliary thread (unlike the other interface calls). Once the object lock has been released, either a blocked Early-Reply or Non-Early-Reply member function will be unblocked and allowed to proceed.

In contrast, a Non-Early-Reply member function invokes only one interface call from the coordinator. The coordinator's entry() is invoked as the first step before proceeding to check if the object lock has already been acquired. If the lock has been acquired, the Non-Early-Reply call will block (as an Early-Reply call) until the object lock is released. If it has not been acquired, the call will simply proceed and its body will be executed. It is important to note that a Non-Early-Reply call does not need to worry about acquiring or releasing the object lock. This is because before its function's body can execute, any previously executing Early-Reply or Non-Early-Reply member call on behalf of the same object would have to be completely finished. A residual segment of a previously executing Early-Reply member call would have to complete and release the object lock. A previously executing Non-Early-Reply member call would prevent the next Early-Reply call from executing because it executes on the same thread. Recall that only residual segments are able to execute on auxiliary threads.

Private and protected member functions of an Early-Reply class do not make use of the coordinator's interface. This is because they cannot be called from outside the Early-Reply object and are always enclosed within a public member function that does make use of the interface.

2.4.3 Self-Contained Concurrency and Thread Scheduling

In order to make use of the services of the coordinator, an Early-Reply class contains a coordinator as a data member. In addition to providing the interface to the concurrency synchronization and initiation, a coordinator stores the state of the object lock. Through the coordinator object, the Early-Reply class (in effect) becomes a monitor that additionally initiates concurrency within the monitor. This makes the concurrency synchronization of an Early-Reply object "self-contained". By this we mean that synchronization on the data of an Early-Reply

object is not affected by nor does it interfere with any other objects of the same class or of different client classes. For example, two Early-Reply member functions can safely execute simultaneously as long as each belongs to a different Early-Reply object.

The framework will not provide any scheduling of the tasks that execute on the auxiliary threads (i.e. residual segments). All residual segments will be invoked in the sequential order of the Early-Reply calls to which they belong are invoked. However, once invoked, the operating system will be free to schedule the execution of each residual segment as it sees fit, since each is executing on a different thread. The framework's blocking mechanism will prevent any computation from commencing that accesses the state of the same object of a currently executing residual segment preventing possible race conditions. If there is no conflict, the residual computation can interleave in any order with any other computation.

2.5 Benefits of the Framework

2.5.1 How the Framework Simplifies Concurrent Programming

Developing concurrent programs that both run reliably and attain reasonable performance gains is difficult even for those who are very proficient in concurrency. Consequently, one of our primary design goals is to simplify concurrent programming for all levels of expertise (including novices). The Early-Reply framework aims to make concurrent programming simpler for both users and implementers of Early-Reply. We consider users of Early-Reply those who make calls to previously implemented member functions of Early-Reply objects. Implementers of Early-Reply are those who actually provide the implementation of early and Non-Early-Reply member functions.

One of the nicest perks for users of Early-Reply is the ability to write programs using the sequential programming model while getting the benefits of parallel execution. This allows simpler sequential design methodologies such as Design by Contract [23] and sequential proof obligations to be used when designing and implementing concurrent programs. Moreover, the sequential programming model implies that concurrency constructs are not visible in client-side code. By providing these two benefits, we are then able to satisfy our goal of keeping the

existing client-side code unchanged from its original form before Early-Reply has been introduced.

The framework is able to support the sequential programming model because the concurrency synchronization and initiation is self-contained within the Early-Reply object, and because all concurrency constructs have been encapsulated away from the user. Self-contained concurrency releases users of Early-Reply from the concerns of race conditions, deadlock, and starvation. It also releases them from concerns of getting an auxiliary thread to execute their concurrent function. The framework takes care of these issues in an automated fashion, and is illustrated with the following code segment shown in Figure 14 and Figure 15.

```
X x();
Y y();

int a, b;
a = x.foo();
b = x.bar(a); // Non-Early-Reply call on object ‘x’ (must block)


// or could have

a = x.foo();
b = y.bar(a);    // call on object ‘y’ (does not have to block)
```

Figure 14. Example code to illustrate automated blocking.

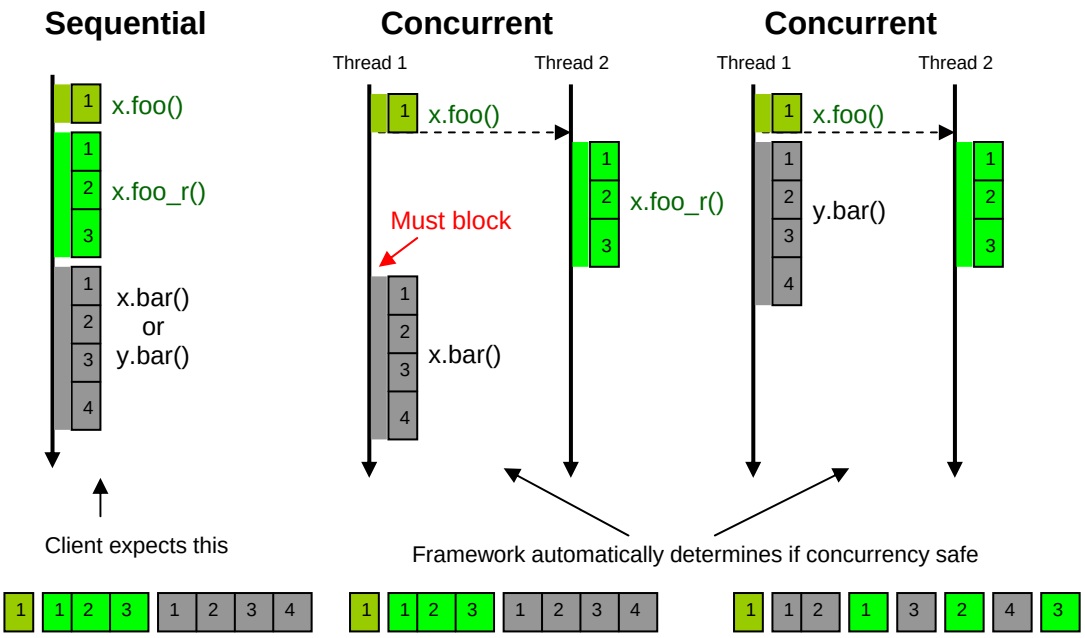

Figure 15. Comparison of execution units between sequential and Early-Reply versions.

To ensure that the concurrency is not visible in client-side code, the framework encapsulates the actual concurrency primitives (i.e. mutex's, thread spawning) within the implementations of the framework classes, and encapsulates the framework interface calls (e.g. er_entry( ), er_start( )) within the implementations of the Early and Non-Early-Reply member functions. This greatly enhances portability and helps facilitate the sequential programming model.

To make concurrent programming simpler for implementers of Early-Reply we realize that the public member functions of an Early-Reply object must be modified somewhat to incorporate the framework. However, we also realize that it is beneficial to minimize this effort. As such, the framework's interface is isolated to one class, the coordinator class, and requiring a minimal set of interface functions (er_entry(), er_start(), er_exit(), and entry()). The implementer must follow a simple rewrite procedure where the Early-Reply member function is split into its material and residual segment and the interface functions are inserted into the appropriate places as illustrated in Figure 16.

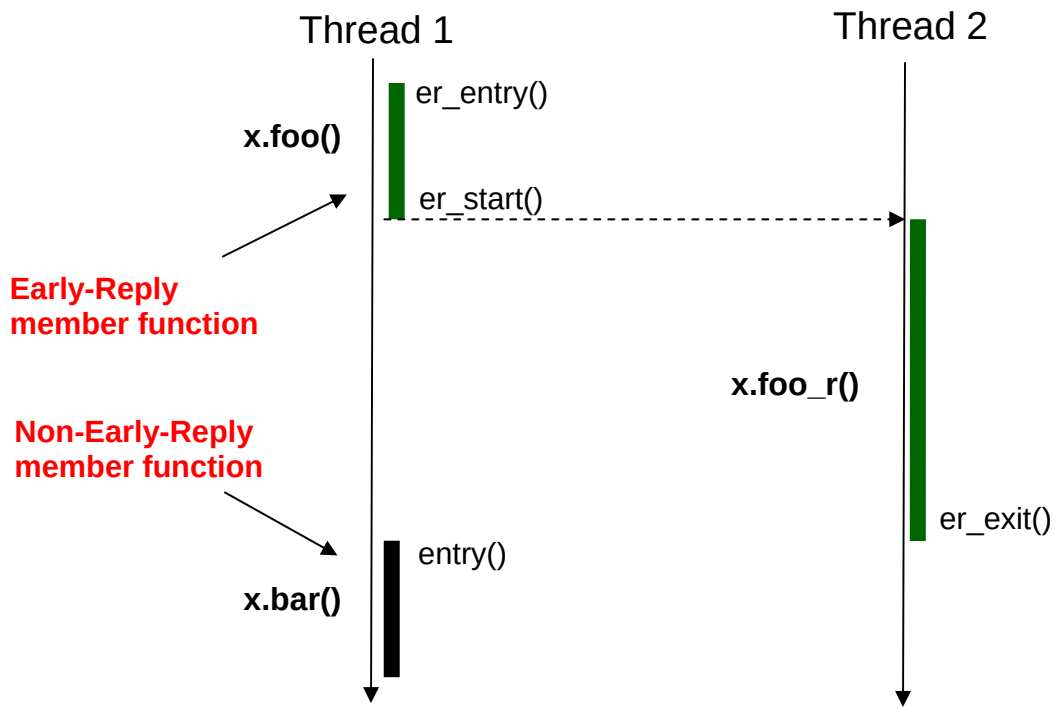

Figure 16. Pictoral view of where framework's interface is used.

As with all concurrency mechanisms, race conditions and deadlock are still possible even with our framework's best intentions. However, assuming the rewrite procedure is followed, the places where they are possible have been isolated by the framework's structure (which will be explained in more detail in the next sub-section). As such the framework is then able to provide a simple set of coding guidelines that, if followed, will eliminate the possibility of race conditions and deadlock. This makes it possible not to require concurrency expertise even for the implementer of Early-Reply, which is not so with traditional concurrency approaches.

These rewrite steps represent the interface for using the Early-Reply framework. The intention here is to make the rewrite steps as simple as possible but not too simple. There is no doubt that this interface could be made even simpler in a number of ways by hiding some of these details. However, we feel it is instructive to keep these steps visible, because they represent the canonical issues that must be taken care of for the concurrency to take place.

2.5.2 How the Framework Makes Concurrent Programs More Reliable

The most critical part of the Early-Reply framework is to make concurrent programming more reliable. By more reliable we mean preventing race conditions, deadlock, and starvation from occurring. These are the primary error conditions associated with concurrency.

Race conditions are possible as soon as the concurrency is initiated, because from that point on multiple threads can access the same variable simultaneously (assuming no synchronization). Traditional synchronization approaches such as monitors or mutex's are typically used after the concurrency has been initiated, which can leave an unprotected gap for race conditions. This is not the case with the Early-Reply framework, since the concurrency is not initiated until after the synchronization says it is safe to do so. In fact, the only place where concurrency is initiated is within the residual segment. As such, the object's data members and any data local to the member functions of an Early-Reply class will be immune from race conditions, assuming the framework's rewrite procedure is followed. Consequently, we only need to be concerned with the function bodies of the residual segments and make sure that they do not access global variables, aliased function parameters, and do not modify the function's return value.

Deadlock can occur with any kind of concurrency approach as soon as concurrency synchronization is introduced into the program. However, the Early-Reply framework is able to limit deadlock occurrences to just the case when there are nested Early-Reply or Non-Early-Reply calls on behalf of the same object. This includes both recursive and non-recursive nested Early-Reply and Non-Early-Reply member function calls.

Like deadlock, starvation is a side effect of synchronization. However, it is typically not as likely to occur as do race conditions or deadlock. Starvation occurs only if different priorities are assigned to the current set of running threads and/or if some sort of scheduler is rearranging the execution order of the threads.

One property of the Early-Reply framework that leads to the elimination of starvation is that there can be at most one task that is blocked on behalf of a particular Early-Reply object. This is true because once an Early-Reply member function has acquired the object lock through its material segment call any subsequent call on behalf of the same object must occur on the

primary thread and will be blocked. As soon as a task is blocked, no other statements can execute on the primary thread until the waiting task is unblocked, thereby not allowing any further er_entry() or entry() calls to be attempted. The only exception to this is a nested Early-Reply call on behalf of the same object in either the material or residual segment. However, this would result in immediate deadlock before starvation would even be considered, and thus should not be allowed. Maintaining wait queues or prioritized threads serves no purpose if there is at most one waiting task, and so will not be done. Therefore, as long as there are not any nested Early-Reply calls on behalf of the same object, the framework does not maintain any prioritized wait queues, and there is no rearranging of the residual segment execution order (i.e. explicit scheduling), starvation can be eliminated.

As we have just discovered, a big benefit of how the framework is structured is that the places within our program where race conditions and deadlock can occur has been isolated to just a few places. For example, with race conditions only the residual segments are of concern. For deadlock, we only need to check for nested Early-Reply calls on behalf of the same object in the material and residual segments. This facilitates the development of a set of coding guidelines that, if followed, will eliminate the possibility of race conditions, deadlock, and starvation from occurring. The coding guidelines are listed below to put them in perspective with the framework analysis and will be discussed further in section 4.2.2:

- No public data members.
- No friend functions/classes involving the Early-Reply class.
- Early-reply return value cannot be modified in the residual segment.
- Data of any enclosing scope is not allowed in the residual segment.
- Residual segment cannot take aliased function parameters.
- Residual segment cannot access data members that have references that exist outside the object.
- Nested Early-Reply calls are not allowed.

It is important to note that the coding guidelines are only needed for implementations of Early-Reply classes. They can be ignored in all other parts of the program. Without the framework, the coding guidelines would be much more complex and more difficult for developers to follow.

2.5.3 How the Framework Exploits Enough Concurrency

It is not the focus of this thesis to optimize the performance gains of concurrent programs. Rather instead we are optimizing for simplicity and reliability, while keeping performance gains at acceptable level. However, a reasonable effort is still made to keep the concurrency overhead to a minimum and to find ways to exploit more concurrency, as long as it does not make the framework more complex or less reliable.

In trying to keep the concurrency overhead to a minimum, there are a number of features that the framework provides.

- Blocking mechanism is “suspend and resume” rather than wasting CPU effort with a “busy wait”.
- A thread pool is used so threads are created (expensive operation) only once and recycled thereafter.
- Initiating the concurrency only after passing the synchronization step may reduce the number of total threads that need to be created for efficient performance.
- Non-Early-Reply member functions are extremely efficient when they don’t have to be blocked since they only check a boolean flag before proceeding and don’t have to release the lock when they complete. In a traditional approach, there would be a costly semaphore or mutex call on entry and one on exit.
- Early-reply member functions also use a boolean flag to check if the lock has been acquired, thus avoiding expensive primitive synchronization calls when possible.

Performance benefits due to concurrency can only be realized when there are two or more threads simultaneously executing. The more concurrency that can be exploited the greater the potential for performance gains. Since our primary focus is to make concurrency simpler and more reliable, we will only embrace an approach to exploit more concurrency if it does not make the framework more complex or less reliable.

One such approach that meets this litmus is sub-object fan-out. Sub-object fan-out involves Early-Reply classes having data members that are themselves Early-Reply classes. More concurrency can be exploited if Early-Reply calls are invoked inside a material or residual segment of the host Early-Reply object. This is illustrated in Figure 17.

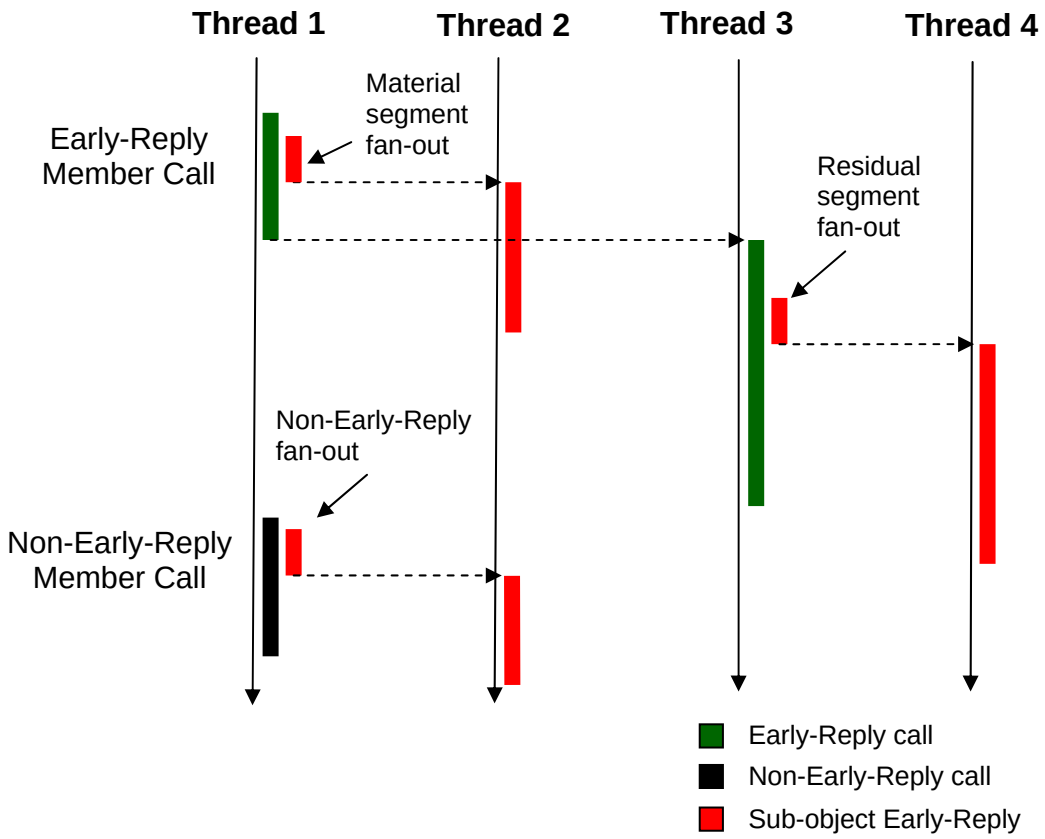

Figure 17. Sub-object fan-out of Early-Reply.

The fan-out is possible because each object's concurrency is completely self-contained. However, one restriction is that the sub-object is created within the host Early-Reply object and there are no references to that sub-object outside of the host. This is inline with the coding guidelines discussed earlier.

Another approach that can potentially exploit more concurrency is to make the synchronization less restrictive. By less restrictive in this case we mean allowing multiple simultaneous "read-

only” access to the object’s state, or allow some nested Early-Reply calls on behalf of the same object that can be done safely. This less restrictive synchronization approach can be done without affecting reliability, but will make the framework a little more complex to use. As such, we will discuss it in a later section, so as to not detract from our main effort.

3 DESIGN AND IMPLEMENTATION OF FRAMEWORK

In this section we will first look at some general design decisions and then follow with a design and implementation level discussion about each of the different functional areas that make up the framework. The functional areas are bootstrapping the framework, synchronizing the concurrency, initiating the concurrency, and terminating the framework.

3.1 General Decisions

3.1.1 Programming Language

As discussed in the Related Work section, Early-Reply exists as an actual language construct in several concurrent languages. For example, in SR [18], Early-Reply can be implemented simply be inserting a “reply” statement inside the body of a function where it is desired to return early. This statement will forward the return value to the caller and also start a new thread in which to execute the remaining code concurrently with the caller's code. Unfortunately, the languages that include the “reply” statement do not provide any inherent synchronization on shared data when using the “reply” statement. Moreover, these languages are not widely used which limits the potential for Early-Reply as a language construct.

In order to make Early-Reply more accessible and provide the necessary synchronization on shared data, our Early-Reply framework will be implemented in C++. This decision is based on the fact that C++ is a widely used general purpose language available on many different platforms, it supports the object-oriented paradigm that will facilitate the concurrency synchronization, and it is also the target language supported by our research group's Static Analysis and Program Transformation project called “The Pivot”. Potential follow up work involves performing static analysis on programs that have been rewritten for Early-Reply to demonstrate that it is possible to verify that a program is free from race conditions and deadlock.

3.1.2 Concurrency Library

In order to simplify the task of implementing the framework's concurrency initiation and synchronization, we will make use of an existing trusted concurrency library to provide the necessary concurrency primitives. Similar to the choice of programming language, the selected library should help make our framework more accessible. Consequently, the POSIX standard API "pthread" library will be used. Library implementations of pthread are freely available on many different platforms. The library provides primitives for spawning threads, conditional variables used for blocking, mutex's for mutual exclusion, and joins for cleanly terminating threads.

3.2 Bootstrapping the Framework

Bootstrapping the framework is what is needed to get the framework up and running. At least one Early-Reply class needs to exist within a program, and each Early-Reply class needs to be modified appropriately. As the first object of an Early-Reply class is created, so will all the threads that will be used to execute the residual segments for the entire program. The following two sections take a look at these steps in more detail.

3.2.1 Setting Up the Early-Reply Class

The Early-Reply framework is introduced into a program by the creation of at least one Early-Reply object that is to implement client specific functionality. An Early-Reply class definition must be modified somewhat so it can make use of the concurrency framework. Listed below are the modifications:

- Declare (by containment) one ER_Coordinator as a private data member. ER_Coordinator's constructor can optionally indicate the desired number of threads in the thread pool.
- For each Early-Reply member function:
    - Declare a new private member function representing the residual segment.
    - Move the residual computation from the original function into the definition of the residual segment. The original function now becomes the material segment.

  - Declare and define a private nested class that derives from ER_Task so it can be used to call the appropriate residual segment and pass any function parameters to the residual segment.
  - The first statement in the material segment is the creation of the nested class derived from ER_Task on the heap. The constructor parameters must be pointers to the host object (this) and the ER_Coordinator. Any function parameter values needed for the residual segment should be included also. The creation statement must be immediately followed by calling the coordinator's er_entry() that takes a pointer to the ER_Task derived object just created.
  - The last statement before the return statement in the material segment must be a call to the coordinator's er_start() that also takes a pointer to the ER_Task derived object.
- The first statement of each public Non-Early-Reply member function must be a call to the coordinator's entry().

The nested class that derives from ER_Task is primarily an implementation detail to accommodate two deficiencies of the pthread_create() function in the pthread library. First, a pointer to a non-static member function cannot be used to invoke the function that starts the execution in the auxiliary thread. Indirection is used to overcome this deficiency by using a pointer to a static member function of ER_Task that in turn calls the polymorphic virtual function call_residual(), so that the residual segment of the Early-Reply class (i.e. non-static member function) can eventually be invoked. Second, only one void pointer can be used to pass function parameters to the pthread_create() function. To allow multiple function parameters to be passed to the residual segment, a nested class derived from ER_Task is used to pack and unpack function parameters. Moreover, the base ER_Task class is used to handle calling the coordinator's er_exit() via call_exit(), so that the implementer of Early-Reply does not have to worry about it.

To better illustrate these issues, an example is shown in Figure 18, Figure 19, and Figure 20 that involves an Early-Reply class X with an Early-Reply member function f(), and a Non-Early-Reply member function g().

```
class X {
public:
    X() : erc(2) {}           // create 2 threads in thread pool
    int  f(short x, double y);           // Early-Reply
    void  g(int x);                      // Non-Early-Reply

private:
    ER_Coordinator erc;

    // One for each Early-Reply member function
    struct  F_Task :  public ER_Task<X>  {
        F_Task(X* er, ER_Coordinator* coord,
                      short a, double b)
              :  ER_Task<X>(er, coord), x(a), y(b)
        {}

        // To know which residual segment to call in aux thread
        void  call_residual()
        {  erptr->resid_f(this);  }

        // following for packing f()'s resid segment params
        short x;
        double y;
    };
    void  resid_f(F_Task* ft);         // f()'s residual segment
};

int X::f(short x, double y)
{
    // indicate host, coordinator along w/ function params
    F_Task* ft = new F_Task(this, erc, x, y);
    erc.er_entry(ft);

    int z;
    /* material computation */

    erc.er_start(ft);    // invoke residual of f() on aux thread
    return z;            // reply early
}

void  X::f_resid(F_Task* ft)
{
    short x = ft->x;               // unpack function params
    double y = ft->y;
    /* residual computation */
}

void X::g(int x)
{
    erc.entry();
    /* non-early-reply computation */
}
```

Figure 18. Early-Reply class definition and its implementation.

```
template<class ERC>
struct  ER_Task  {
    ERC* erptr;
    ER_Coordinator* coordptr;
    int engine_index;
    bool inQ;          // indicates if waiting in thread pool queue

    ER_Task(ERC* er, ER_Coordinator* coord)
           :  erobj(er), coordptr(coord), inQ(false)  {}
    virtual  ~ER_Task()  {}

    // engine calls these to invoke appropriate resid seg and er_exit()
    virtual  void  call_residual() = 0;
    void  call_exit()
    { coordptr->er_exit(this);  }
};
```

Figure 19. ER_Task class defintion.

```
int  main()
{
    X x = X(2);
    Y y = Y();
    int val  =  x.f(2, 6.75);   // Early-Reply call
    y.foo(val);           // unrelated, can execute concurrently w/ f()
    x.g(5);               // Non-Early-Reply, blocks if f()'s residual
                          //    not finished
}
```

Figure 20. Driver code for example.

3.2.2 Starting Up the Thread Pool

Once an Early-Reply object is created, the framework bootstrapping process begins by first creating an ER_Coordinator. The coordinator object in turn gets a pointer to a thread pool object that is a singleton (i.e. first coordinator object created will create the thread pool object). This is shown in the code in Figure 21.

```
ER_Coordinator::ER_Coordinator(int num_threads)  :  er_active(false),

entry(false),

term_coordinator(false)
{
    ertp = ER_Thread_Pool::singleton(num_threads);
    pthread_mutex_init(&early_reply_mutex, NULL);
    pthread_cond_init(&early_reply, NULL);
}

// this is a static member function
ER_Thread_Pool*  ER_Thread_Pool::singleton(int num_threads)
{
    if (sm_ertp == 0)  {
        sm_etrp = new ER_Thread_Pool(num_threads);
        pthread_mutex_init(&ref_count_mutex, NULL);
    }
    pthread_mutex_lock(&ref_count_mutex);
        ref_count++;
    pthread_mutex_unlock(&ref_count_mutex);
    return sm_ertp;
}
```

Figure 21. ER_Coordinator constructor and ER_Thread_Pool singleton.

The singleton thread pool constructor then creates a number of thread engine objects equal to the desired number of threads for the pool, and puts them into a vector named engines. Before returning back to the Early-Reply object constructor, synch_engine() is called on each thread to make sure all threads have been created and are waiting to start executing residual segments. Definitions for the thread pool and thread engine are shown in Figure 22.

It is important to note that a race condition is not possible when creating the singleton ER_Thread_Pool, since no auxiliary threads have been created at this point. Therefore no synchronization is necessary.

```
ER_Thread_Pool::ER_Thread_Pool(int nt) :
  num_threads(nt), num_using_pool(0), wait_called(false), pool_full(false)
{
    pthread_mutex_init(&pool_mutex, NULL);
    pthread_cond_init(&pool, NULL);
    for (int i = 0;  i < num_threads;  i++)  {
        engines.push_back( new ER_Thread_Engine(i) );
        engines[i]->synch_engine();  // all threads waiting at same place
    }
}

ER_Thread_Engine::ER_Thread_Engine(int id) : booked(false), engine_id(id),
   engine_started(false), resid_running(false), term_engine(false),
sec_thread(0)
{
    pthread_mutex_init(&residual_mutex, NULL);
    pthread_cond_init(&residual, NULL);
    sec_thread = new pthread_t();
    pthread_create(sec_thread, NULL, &ER_Thread_Engine::start_engine,
this);
}
```

Figure 22. ER_Thread_Pool and ER_Thread_Engine constructors.

The pthread API for pthread_create() specifies a function pointer to indicate which function should be called to execute in the new thread. Consequently, the static member function start_engine() is used to start an infinite loop that waits for a signal to start executing a given residual segment. How a thread engine is started up is shown in Figure 23.

```
// this is a static member function
void*  ER_Thread_engine::start_engine(void* obj)
{
    static_cast<ER_Thread_Engine*>(obj)->thread_loop();
    return 0;
}

void  ER_Thread_engine::synch_engine()
{
    pthread_mutex_lock(&residual_mutex);
        while (! engine_started)
            pthread_cond_wait(&residual, &residual_mutex);
    pthread_mutex_unlock(&residual_mutex);
}
```

Figure 23. Static member function start_engine() and non-static synch_engine().

After all of the desired thread engines have reached the first conditional wait (just before the infinite loop), the Early-Reply framework is considered to be bootstrapped, and ready to execute residual segments as shown in Figure 24.

```
void  ER_Thread_engine::thread_loop()
{
    pthread_mutex_lock(&residual_mutex);
       engine_started = true;
       pthread_cond_signal(&residual);   // for synch_engine()s cond wait
       // synch_engine() gets each thread to this point
       pthread_cond_wait(&residual, &residual_mutex);
    pthread_mutex_unlock(&residual_mutex);

    while(1)  {
       if (term_engine && ! resid_running)  break;
       task->call_residual();       // virtual call to approp res segment

       pthread_mutex_lock(&residual_mutex);
           task->call_exit();        // virtual call to coord's er_exit()
           resid_running = false;
           if (! term_engine)
               // wait until another res segment comes
               pthread_cond_wait(&residual, &residual_mutex);
          delete task;  task = 0;
       pthread_mutex_unlock(&residual_mutex);
    }
}
```

Figure 24. ER_Thread_Engine's thread loop.

3.3 Synchronizing the Concurrency

The interface for serializing access to the Early-Reply object's state is handled by ER_Coordinator's er_entry(), entry(), and er_exit(). The lock used to synchronize each early-object is encapsulated within the ER_Coordinator, which is represented by the boolean data member er_active. Recall that each Early-Reply object contains its own ER_Coordinator.

Before the material segment of an Early-Reply call can begin, it must first call ER_Coordinator's er_entry(), which checks if the lock has already been acquired. If it has, the call will block on the primary thread until the previous Early-Reply call releases the lock upon exit. If the lock has not yet been acquired, or a signal has been sent indicating that the lock has been released, the

er_entry() call will first acquire the lock, and then call thread pool's pool_entry() to book an unused thread in the thread pool. Figure 25 shows er_entry()'s definition.

```
void  ER_Coordinator::er_entry(ER_Task* task)
{
   pthread_mutex_lock(&early_reply_mutex);
      while ( er_active ) {
         entry = true;
         // block if lock already acquired
         pthread_cond_wait(&early_reply, &early_reply_mutex);
      }
      entry = false;
      er_active = true;                       // acquire the lock
   pthread_mutex_unlock(&early_reply_mutex);

   ertp->pool_entry(task);      // try to book an unused thread in pool
}
```

Figure 25. ER_Coordinator's er_entry().

If the thread pool is full (i.e. all threads being used), the task object corresponding to the Early-Reply call will be put on a waiting queue, and the Early-Reply call will block for a possible second time until an unused thread becomes available. If a thread is already available, the task object will be assigned an index corresponding to the available thread in the thread pool, which is a thread running in a thread engine object. Figure 26 shows how a thread is reserved and assigned from the thread pool.

```
void  ER_Thread_Pool::pool_entry(ER_Task* task)
{
   pthread_mutex_lock(&pool_mutex);
      wait_called = false;
      if ( pool_full )  {         // if full, put task on waiting queue
         waiting_tasks.push_back(task);
         task->inQ = true;
         while ( task->inQ)  {
            wait_called = true;
            // block until pool has an unused thread
            pthread_cond_wait(&pool, &pool_mutex);         }
      }
      // unused thread available, assign it to task object
      else  {
         assign_engine(task);
      }
      if ( ++num_using_pool >= engines.size() )
         pool_full = true;
   pthread_mutex_unlock(&pool_mutex);
}

void  ER_Thread_Pool::assign_engine(ER_Task* task)
{
   size_t  i;
   for (i = 0;  i < engines.size();  i++)
      if ( ! (engines[i]->booked) )  break;
   task->engine_index = i;                 // assign first free thread
   engines[i]->booked = true;              // engine is now booked
}
```

Figure 26. ER_Thread_Pool's pool_entry() and assign_engine().

Before a Non-Early-Reply call can proceed, it must first call ER_Coordinator's entry(), whose only job is to check if the lock has already been acquired. If it has, the call will block on the primary thread until the earlier Early-Reply call releases the lock upon exit. If the lock has not yet been acquired, or a signal has been sent indicating the lock has just been released, the entry() call will return allowing the Non-Early-Reply call to proceed on the primary thread. The entry() definition for Non-Early-Reply member functions is shown in Figure 27.

```
void  ER_Coordinator::entry()
{
   pthread_mutex_lock(&early_reply_mutex);
      if ( er_active ) {              // checks if lock already acquired
         entry = true;
         pthread_cond_wait(&early_reply, &early_reply_mutex);
         entry = false;
      }
   pthread_mutex_unlock(&early_reply_mutex);
}
```

Figure 27. ER_Coordinator's entry() for Non-Early-Reply member functions.

After a residual segment has completed its computation, the last thing it must do is call ER_Coordinator's er_exit(). The details of the completion of an Early-Reply member function are shown in Figure 28. Initially, the thread pool object is notified through pool_exit() that a thread has just become available. Then the lock is released, and if there is a waiting Early-Reply or Non-Early-Reply call, a signal will be sent to unblock the waiting call.

```
void  ER_Coordinator::er_exit(ER_Task* task)
{
   ertp->pool_exit(task);
// notify that a thread is ready for reuse

   pthread_mutex_lock(&early_reply_mutex);
      er_active = false;                        // release the lock
      if ( entry || term_coordinator )
         pthread_cond_signal(&early_reply);    // only if a call waiting
   pthread_mutex_unlock(&early_reply_mutex);
}

void  ER_Thread_Pool::pool_exit(ER_Task* task)
{
   pthread_mutex_lock(&pool_mutex);
      engines[task->engine_index]->booked = false;    // no longer used

      if ( --num_using_pool  <  engines.size() )
         pool_full = false;

      if ( wait_called )  {                        // if a task is waiting
         ER_Task*  imminent_task  =  waiting_tasks.front();
         waiting_tasks.pop_front();
         imminent_task->inQ = false;
         // assign index to task at queue front
         imminent_task->engine_index  =  task->engine_index;
         engines[task->engine_index]->booked = true;
         if ( ++num_using_pool  >=  engines.size() )
            pool_full  = true;
         wait_called = false;
         // signal all waiting tasks, but only one whose inQ false proceeds
         pthread_cond_broadcast(&pool);
      }
   pthread_mutex_unlock(&pool_mutex);
}
```

Figure 28. ER_Coordinator's er_exit() and ER_Thread_Pool's pool_exit().

3.4 Initiating the Concurrency

The interface for initiating concurrency is handled by ER_Coordinator's er_start(). The nested class derived from ER_Task keeps track of the appropriate thread engine used for executing the residual segment and supplies call_exit() to call ER_Coordinator's er_exit(). It also supplies the virtual call_residual() to call the appropriate residual segment. As soon as er_start() is invoked, the derived ER_Task object is passed to the thread pool to indicate which thread engine should be used as shown in Figure 29.

```
void  ER_Coordinator::er_start(ER_Task* task)
{
   ertp->set_task(task);   // calls thread pool since no access to engine
}

void  ER_Thread_Pool::set_task(ER_Task* task)
{
   // calls appropriate engine so task ptr can be set
   engines[task->engine_index]->set_task(task);
}

void  ER_Thread_Engine::set_task(ER_Task* t)
{
   task = t;        // for virtual call_residual() & call_exit() in loop
   pthread_mutex_lock(&residual_mutex);
      resid_running = true;           // residual segment will be running
      pthread_cond_signal(&residual); // wake up infinite loop in engine
   pthread_mutex_unlock(&residual_mutex);
}
```

Figure 29. ER_Coordinator's er_start().

Notice in ER_Thread_Engine's thread_loop() in Figure 24, that once the signal has been sent from set_task(), the loop will wake up and begin executing the appropriate residual segment through the virtual call_residual(). After the residual segment is executed, er_exit() is invoked through call_exit() to release the lock.

A benefit of invoking the residual segment within the wrapper call_residual() is that the coordinator's er_exit() will be called appropriately even when there are multiple return statements within the residual segment. Another benefit is that the body of call_residual() is a good place to catch any exceptions that might be thrown from the residual segment.

3.5 Terminating the Framework

When the last Early-Reply object is destroyed, the Early-Reply framework will begin to perform some clean-up before destroying all of the required objects, which is shown in Figure 30. As the destructor for each Early-Reply object is called, so will the destructor of the contained ER_Coordinator. The ER_Coordinator destructor in turn calls the static member function destroy() on behalf of ER_Thread_Pool so that a count of existing ER_Coordinators can be maintained. When this count goes to zero, the ER_Thread_Pool singleton considers itself no

longer useful, and calls a non-static destroy(). This destroy() sets the term_engine flag in each ER_Thread_Engine so that each will break out of its infinite loop and finally invoke pthread's join() for a graceful termination of the auxiliary thread.

```
ER_Coordinator::~ER_Coordinator()
{
   pthread_mutex_lock(&early_reply_mutex);
      if ( er_active ) {              // if lock acquired, wait for resid
         term_coordinator = true;
         pthread_cond_wait(&early_reply, &early_reply_mutex);
      }
   pthread_mutex_unlock(&early_reply_mutex);

   ER_Thread_Pool::destroy(ertp);          // call static member function
   pthread_mutex_destroy(&early_reply_mutex);
   pthread_cond_destroy(&early_reply);
}

// static member function
void  ER_Thread_Pool::destroy(ER_Thread_Pool*  ertp_self)
{
   pthread_mutex_lock(&ref_count_mutex);
      int temp_rc  =  --ref_count;
   pthread_mutex_unlock(&ref_count_mutex);

   if (tmp_rc == 0)  {
      ertp_self->destroy();  // dtor not called so thread pool can restart
      delete  ertp_self;
      ertp_self  =  0;
   }
}

void  ER_Thread_Pool::destroy()
{
   for (size_t  i = 0;  i < engines.size();  i++)  {
      engines[i]->stop_engine();             // get thread_loop() to break
      // join when thread_loop() terminates
      pthread_join( *(engines[i]->get_thread() ), NULL);
      delete engines[i];
   }
   pthread_mutex_destroy(&pool_mutex);
   pthread_cond_destroy(&pool);
}

void  ER_Thread_Engine::stop_engine()
{
   pthread_mutex_lock(&residual_mutex);
      term_engine = true;
      if ( ! resid_running )
         pthread_cond_signal(&residual);         // signal loop to break
   pthread_mutex_unlock(&residual_mutex);
}
```

Figure 30. Clean-up and destructors.

4 HOW TO USE THE FRAMEWORK

4.1 Candidate Applications

The Early-Reply framework presented here is just one of many approaches to concurrent programming, and is not intended to be the solution for all applications in need of concurrency. Rather it is intended for a specific class of applications that exhibit Early-Reply behavior and are able to achieve a performance gain in spite of the concurrency overhead. In this section, we will try to categorize the different software applications that may be able to take advantage of Early-Reply. We will also try to define the applications that would not benefit from Early-Reply and may need to use another approach.

4.1.1 Favorable to Early-Reply

As discussed earlier, of the two categories of sub-program concurrency, Early-Reply is better suited for task parallelism than it is for data parallelism. This means that for a performance gain to be possible, candidate applications must have two or more independent tasks that can execute concurrently and are able to operate on different data.

In terms of Early-Reply, listed below are two minimal constraints that must be met for a performance gain to be possible.

- Computation in the residual segment must exceed the Early-Reply overhead.
- There must also be some computation that immediately follows the Early-Reply call on the primary thread that can execute safely and simultaneously with the residual segment on the auxiliary thread. Moreover, this computation must also exceed the Early-Reply overhead.

Only if both of these constraints are met, is it possible for the Early-Reply framework to improve a program's performance. As a general guideline, for the computation to exceed the Early-Reply

overhead something on the order of hundreds or thousands of statements would need to be executed.

Even though the class of applications that can benefit from the Early-Reply framework is limited, there still are several different forms of computation that could prove to be useful. Below is an initial list of some possible different uses for Early-Reply. This list is just a first effort at discovering the different ways the Early-Reply framework might be useful, and could be expanded as the Early-Reply approach evolves.

- Pipelining
- Prefetching
- Initialization

When we refer to pipelining in this context we are referring to something closer to instruction set pipelining in computer architecture. In instruction set pipelining, the result computed by the ALU for a given instruction can forwarded early to another later instruction that needs the result as its input, allowing two or more instructions to proceed in parallel. The earlier instruction can finish its memory access and register write-back cycle, while the later instruction uses the forwarded result as input to an ALU operation. We can think of the initial instruction up to computing the forwarded result by the ALU as the material segment of an Early-Reply call, and the memory access and register write-back as the residual segment. The later instruction is analogous to the other computation discussed above that follows the Early-Reply call. This correspondence is illustrated in Figure 31.

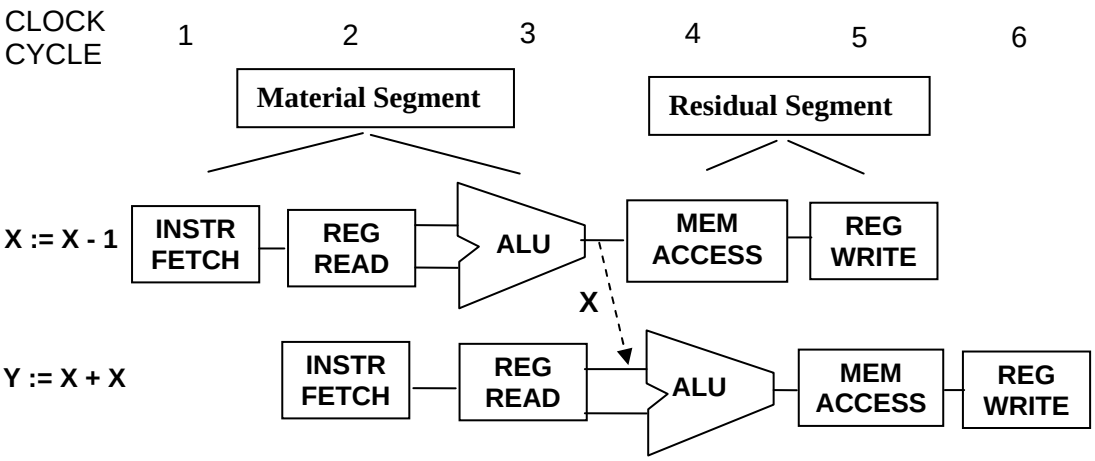

Figure 31. Similarities between instruction pipelining and Early-Reply.

Although not a perfect match, the Decaf compiler application discussed in the Experiments and Results section is a good example application to illustrate the framework being used for pipelining. We can think of each input file as being analogous to a pipelined instruction. The parsing and semantic analysis operations are analogous to the instruction fetch, data fetch, and ALU operations, and the optimization and code generation being analogous to the memory and register write. Each input source file is analogous to an instruction, and like pipelined architectures, the more source files that can be compiled simultaneously the better the potential speedup.

Pre-fetching (or pre-computing) is another form of computation that is well suited for Early-Reply. Pre-fetching involves getting or computing a result well before it is actually needed. With Early-Reply, if there is a particular result that needs to be located or computed over and over in a loop, we can compute the first result sequentially, define a material segment that simply returns the computed result, and then use the residual segment to pre-compute the next result asynchronously on an auxiliary thread so that the next time the material segment is called, the result can be returned immediately after the call. This is shown in Figure 32, where foo() from object ‘x’ is an Early-Reply member function that performs pre-computation, and bar() from object ‘y’ uses the pre-computed value returned early from x.foo(). Performance can be improved because y.bar() and the residual segment of x.foo() can execute simultaneously. Note

that to get the pre-fetching started out correctly, the residual segment of foo() must execute first without the material segment. Both the search for a word in a directory of files and the search for a word in one file discussed in the Experiments and Results section are good example applications for pre-fetching.

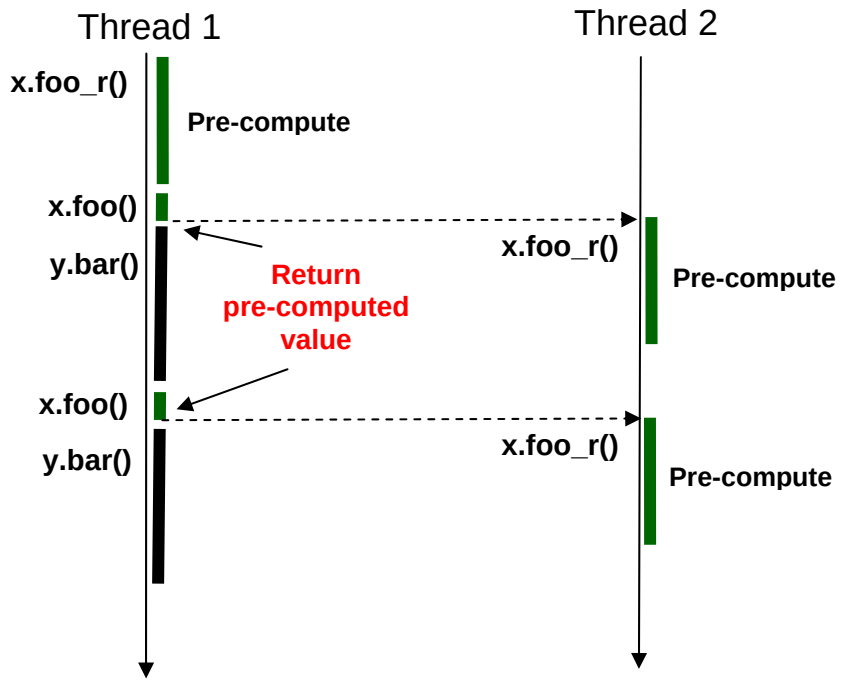

Figure 32. Pictoral view of Early-Reply making use of pre-fetching.

Early-reply can also effectively used to initialize the activities for the auxiliary thread before the primary thread continues [16]. For example, in a web browser, the client code responsible for formatting a web page can use an Early-Reply function for each in-line image. Each material segment can contact the appropriate server to get the image's size and return this size early so the client can place text and other images properly on the page. However, just before the material segment returns, it can begin transferring the actual data from the server concurrently in the residual segment.

When discussing applications favorable for Early-Reply, we must also consider the two primary types of performance gains desired when using concurrency:

- Improved Throughput (or total execution time)
- Response Time

All three of the forms of using Early-Reply discussed above can be used to improve throughput or response time.

Using Early-Reply to improve throughput or total execution time simply means that applications will take less time to complete. This can improve the user's productivity and/or the productivity of the machine since more programs can run within the same unit time. Both the Decaf compiler and the word search within a directory of files were good examples of improved throughput.

Improved response time means to shorten the time interval that a human must wait for an application to respond or that a thread must wait for another thread to respond over an interval before continuing. The word search in a single file is a good example of improved response time, since the human waits for less time between subsequent "find" commands allowing them to be more productive. At the beginning of this section we mentioned that in order for a performance gain to be realized, there must be enough computation following the Early-Reply call on the primary thread that can safely and simultaneously execute with the residual segment executing on an auxiliary thread. When considering interactive programs for improving response time, we can substitute average human processing time for the computation following the Early-Reply call. Humans typically take several seconds to respond to an event produced by a program, and this time can be thought of as some form of computation executing concurrently with the residual segment of an Early-Reply function.

4.1.2 Un-Favorable for Early-Reply

Since Early-Reply is not yet a well understood mature concurrency approach, it is too early to predict with certainty in which cases it will be successful and in which cases it will not. Nevertheless, we will try to identify some uses where we are fairly confident that it will not be successful so that developers can better focus their attention on other more lucrative concurrency approaches. Listed below are three uses that would be highly suspect as the right approach to improving the performance of a program:

- Data parallelism
- Real-time applications
- Low amortized computation in residual segments.

For applications in need of data parallelism, the overhead of an Early-Reply call may overwhelm any performance benefit derived from the concurrency. Data parallelism typically involves performing the same operation on different parts of the same data. Many applications in this area involve heavy number crunching on large amounts of data and use a large number of processors. The primary problem in data parallelism is typically how to do the number crunching efficiently on the given set of processors, rather than avoiding race conditions or deadlock. It also might be the only thing the program does during its entire execution, so there is no need to worry about other conflicting tasks executing simultaneously with the primary task.

In many cases, real-time applications have hard constraints where one or more tasks must execute within a specific time interval or failure will result. In these cases it may be necessary for certain tasks to be given higher priorities than other tasks or a scheduler may rearrange the execution order of the tasks so that the critical tasks are able to complete within the desired interval. This may be possible for future incarnations of Early-Reply, but our current focus has been to make the framework simple and well behaved. Consequently, prioritized tasks or rearrangement of task execution order by a scheduler is currently not supported by this framework.

Another situation that may not produce satisfactory performance results is when the residual segment doesn't have enough consistent computation to overcome the concurrency overhead. This can happen when a sequential algorithm is rewritten into an Early-Reply function with a false impression that the residual segment will perform a fair amount of computation because the algorithm states O(lg n) or O(n) for the residual segment. The trouble with this is that the limits are upper limits or worst case scenarios. A good example of this is the disjoint sets Union/Find where the union operation is O(n) [24]. However, in many cases the performance of the union is constant time because there is only one element to union with the set. Only rarely does the worst case even come close and that is when two equal size sets are union'ed together.

4.2 Applying the Framework to the Application

This section discusses the mechanics of applying the Early-Reply framework to a given program so that performance gains can be achieved without sacrificing reliability or simplicity. The application of the framework involves both a rewrite procedure and a set of coding guidelines that need to be observed by the implementer of each Early-Reply class. The rewrite procedure specifies how the framework interface is to be used, while the coding guidelines specify things that should be avoided. Recall that once an Early-Reply class has been implemented correctly, the user of Early-Reply member functions can design and reason about them using the sequential programming model.

4.2.1 Rewrite Procedure

Although the rewrite procedure has already been discussed, it will be shown again to put it in context with the coding guidelines. One very nice property of the framework is that the only place where modification is necessary to the program is within the implementation of each Early-Reply class. Consequently, the entire effort of applying the framework only involves the Early-Reply classes. Listed below is the Early-Reply rewrite procedure that is to be followed for each Early-Reply class:

- Declare (by containment) one ER_Coordinator as a private data member. ER_Coordinator's constructor can optionally indicate the desired number of threads in the thread pool.
- For each Early-Reply member function:
    - Declare a new private member function representing the residual segment.
    - Move the residual computation from the original function into the definition of the residual segment. The original function now becomes the material segment.
    - Declare and define a private nested class that derives from ER_Task so it can be used to call the appropriate residual segment and pass any function parameters to the residual segment.

  - The first statement in the material segment is the creation of the nested class derived from ER_Task on the heap. The constructor parameters must be pointers to the host object (this) and the ER_Coordinator. Any function parameter values needed for the residual segment should be included also. The creation statement must be immediately followed by calling the coordinator's er_entry() that takes a pointer to the ER_Task derived object just created.
  - The last statement before the return statement in the material segment must be a call to the coordinator's er_start() that also takes a pointer to the ER_Task derived object.
- The first statement of each public Non-Early-Reply member function must be a call to the coordinator's entry().

4.2.2 Coding Guidelines to Follow

By strictly following the rewrite procedure, the framework's structure is now able to isolate to just a few places within the program where race conditions and deadlock can occur. Recall that starvation has been eliminated. To prevent race conditions, only the residual segments need to be checked. For deadlock, nested Early-Reply calls cannot be made on behalf of the same object in either a material or residual segment. This facilitates the development of a set of coding guidelines that, if followed by the implementer of the Early-Reply classes, will eliminate the possibility of race conditions and deadlock from occurring. As is the case with the rewrite procedure, the coding guidelines are only needed for Early-Reply classes and are listed below:

- No public data members.
- No friend functions/classes involving the Early-Reply class.
- Early-reply return value cannot be modified in the residual segment.
- Data of any enclosing scope is not allowed in the residual segment.
- Residual segment cannot take aliased function parameters.
- Residual segment cannot access data members that have references that exist outside the object.
- Nested Early-Reply calls are not allowed.

The first two guidelines are ones most developers would follow anyway to ensure good object encapsulation. The restriction of not being able to use aliased function parameters may be too severe, and may be relaxed in the future. Without the structure of the framework, the coding guidelines would be much more complex and extremely difficult for a developer to follow.

Although it is possible for the guidelines to be followed through extreme diligence, it may be more reassuring if the guidelines could be verified in an automated fashion. We feel that static analysis would be the appropriate tool to perform the automated verification. The coding guidelines listed above for the implementer of Early-Reply would be the same ones used by static analysis, and will be discussed in the Verification section.

5 VERIFICATION ON THE ABSENCE OF CONCURRENCY ERROR CONDITIONS

One of the primary design goals of the Early-Reply framework is to improve the reliability of concurrent programs, by preventing the occurrence of race conditions, deadlock, and starvation. All three error conditions are by-products of concurrent programming. The structure of the framework has been designed to isolate or eliminate where these error conditions can occur. This in turn simplifies the coding guidelines that programmers must follow to develop reliable concurrent programs and simplifies the static analysis process that can be used to automatically verify the absence of these error conditions. In this section we will informally prove how the structure of the Early-Reply framework, rewrite procedure, and coding guidelines are used together to prevent the occurrence of race conditions, deadlock, and starvation. We will also discuss how static analysis can be used to verify the absence of these concurrency error conditions in an Early-Reply program.

5.1 Informal Proofs

One assumption that must be made is that the Early-Reply framework is the only concurrency approach that is used within the program. If other concurrency approaches are allowed, especially those that initiate concurrency, the programmer's coding guidelines and verification process would become too complex. Consequently, in order to keep the verification process by static analysis tractable, this will be the only coding guideline that will not be verified.

Two of the framework's most important properties are that concurrency is initiated in the residual segment only after acquiring the object lock, and the lock is checked and acquired only in the material segment on the primary thread. This means that the residual segment is the only place where computation can take place on an auxiliary thread, and the only place where the object lock can be checked and acquired is on the primary thread. Limiting auxiliary thread computation to the residual segments helps isolate where race conditions can occur. Restricting the checking and acquiring of an object lock to the primary thread reduces the maximum number

of member functions of the same Early-Reply object to one that can be blocked waiting to gain access to the object. This helps prevent and isolate potential race conditions and deadlock, and also eliminates the possibility of starvation.

5.1.1 Race Conditions

A race condition is a situation where multiple processes (or threads) are allowed to access and manipulate the same data simultaneously, and the outcome of the execution depends on the particular order in which the access takes place [25]. To guard against race conditions, we need to ensure that only one thread at a time can access the same shared data. The framework facilitates this by restricting execution on auxiliary threads to the residual segments of Early-Reply member functions. This property means that only the residual segments need to be examined for potentially unsafe accesses to show that the program is free from race conditions. The potential unsafe accesses within the residual segment are listed below:

- Accessing data members of the host object (i.e. this).
- Accessing function parameters that are references or pointers to objects that exist outside the host object.
- Modifying the value returned by the material segment.
- Accessing data of scope outside of the class. Class scope includes scope of any base classes.

The framework synchronization along with coding guidelines provide mutual exclusion for data members of the Early-Reply host object so that they cannot be involved in race conditions. The framework does not allow more than one member function of the same Early-Reply object to execute at the same time. Moreover, the material segment and residual segment of the same Early-Reply member function cannot execute simultaneously since the residual segment is invoked as the last statement before the return statement. Verification of the rewrite procedure confirms that these two statements hold. Coding guidelines will not allow access to data members from outside the object through public data members and friend functions/classes of Early-Reply classes. This can be confirmed by verification of the coding guidelines.

The residual segment does not take function parameters in a traditional sense since it is invoked in a special way to execute on an auxiliary thread. However, it can be passed function parameters via the ER_Task derived object. The coding guidelines dictate that references or pointers to objects that exist outside the host object cannot be data members of the ER_Task derived object. Verification of the coding guidelines by static analysis can ensure that this does not happen even if the programmer neglects to follow the guideline.

Modifying the return value returned by the material segment and accessing data of a scope outside of the host class is something that the framework cannot itself prevent and therefore must be controlled through the coding guidelines. Like the previous problem, verification of the coding guidelines by static analysis can ensure that this does not happen even if the programmer neglects to follow the guidelines.

What has just been covered is an exhaustive list of possible situations where race conditions could occur and their corresponding solutions. Limiting auxiliary thread computation to the residual segments simplifies what is needed to ensure that a program is free from race conditions by isolating potential race conditions to well defined units of the program (i.e. residual segments).

5.1.2 Deadlock

In this thesis we will only concern ourselves with deadlock that is induced by concurrency synchronization. For deadlock to occur, the following four conditions must all hold simultaneously [25]:

- **Mutual exclusion:** At least one resource must be held in a non-sharable mode. In our case, the resource is the object lock on an Early-Reply object that has been acquired.
- **Hold and wait:** There must exist a process (or thread) that is holding at least one resource and intends to acquire additional resources that are currently held by other processes. In our case, a thread holds the object lock and wishes to acquire another lock or the same one again.
- **No preemption:** Resources cannot be preempted. The object lock will not be released until the thread that has acquired it has finished its task.

- **Circular wait:** There must exist a set of waiting processes (or threads) that is involved in a circular wait (or cycle) for a resource to be released. In our case, this could include a self-cycle where a thread has acquired a lock on a specific Early-Reply object and attempts to acquire the lock on the same object again without releasing the initial lock.

To prevent deadlock from occurring, we need to ensure that at least one of these conditions does not hold in each potential case. In our case we will prevent deadlock induced by the synchronization of the concurrency.

As with race conditions the framework facilitates the prevention of deadlock by isolating where deadlock can occur. Public member functions of an Early-Reply class are the only places where concurrency synchronization is implemented. Furthermore, since Non-Early-Reply member functions do not actually acquire the object lock, their definitions do not need to be considered. As such, we only need to examine the material and residual segments for the following potential unsafe actions:

- Recursive or nested Early-Reply calls on behalf of the same object.
- Early-reply calls on behalf of objects that exist outside the host object. Early-reply calls on behalf of completely self-contained sub-objects are legal.

The restriction of not allowing recursive or nested Early-Reply calls on behalf of the same object may be lifted in future versions. This restriction is put in place as to not unnecessarily complicate the first effort of the framework. Recursive or nested Early-Reply calls cannot be made in either the material or the residual segment. Verification of the coding guidelines by static analysis can ensure that this does not happen even if the programmer neglects to follow the guideline.

Early-reply calls made on behalf of objects that exist outside the host object can cause deadlock in the following situation: An Early-Reply call is made on behalf of object B within the body of Early-Reply member function of object A, and an Early-Reply call is made on behalf of object A within the body of Early-Reply member function of object B. This case is moot within residual segments because the race condition coding guidelines do not allow residual segments access to

objects that exist outside the host object. Material segments, however, do allow access to objects that exist outside the host object. Verification of the coding guidelines by static analysis can ensure that this does not happen even if the programmer neglects to follow the guideline.

5.1.3 Starvation

Starvation is when one or more processes (or threads) waits indefinitely trying to gain access to a critical section [25]. Starvation is possible if it can be shown that there is no finite bound on the number of times that other processes (or threads) are allowed to enter their critical section after a process (or thread) has made a request to enter its critical section and before that request is granted. In our case, the critical section is access to the Early-Reply object's state through either an early or non-early reply call of the same object.

We have already discussed in the Benefits of the Framework Subsection in Section 2 that the framework dictates that there can be at most one early or Non-Early-Reply call blocked while trying to gain access to an Early-Reply object's state. This makes priority queues and any scheduling unnecessary. Therefore, there is finite bound of "one" on the number of times that other threads are allowed to enter their critical section before or after another has made its request to enter the critical section, which means that starvation is not possible.

5.2 Static Analysis for Automatic Verification

In this section we will show how verification by static analysis can be done to guarantee that an Early-Reply program is free from race conditions, deadlock, and starvation. Although it is possible that a diligent developer that strictly follows the rewrite procedure and the coding guidelines will produce a program free from the concurrency error conditions, conventional wisdom says that this will not be the norm. Two of the most likely conditions for this not to be so are when the program has multiple authors or when the program is very large and is modified over a long time interval. To be certain an automated process for verification such as static analysis would be best. Verification using static analysis will involve two primary phases:

- Verification that the Early-Reply rewrite was performed correctly.
- Verification that the framework coding guidelines were followed.

If verification of both of these phases turns out positive, then we can be certain that the program is free from the primary concurrency error conditions of race conditions, deadlock, and starvation.

5.2.1 Rewrite Procedure Verification

In order not to unnecessarily complicate the rewrite procedure verification phase, we will assume that the Early-Reply framework is the only concurrency approach used with the program. We feel that it is very unlikely that this assumption would be violated by developers. Given this assumption, it is only necessary to perform static analysis on Early-Reply class definitions and their associated member function implementations. This is a powerful property of the framework which simplifies the static analysis verification for rewrite to the following four steps:

- Identify the Early-Reply classes.
- Identify the Early-Reply and Non-Early-Reply member functions of each Early-Reply class.
- Verify the rewrite for each Early-Reply class definition.
- Verify the rewrite for each Early-Reply and Non-Early-Reply member function.

Identifying the Early-Reply classes is accomplished by analyzing each class definition in the program to see if it contains an ER_Coordinator as a data member. If it does, it is considered an Early-Reply class, and the member name is stored.

Once an Early-Reply class has been identified, the Early-Reply member functions are identified. An easy way this could be done would be to require the Early-Reply implementer to add the suffix "_resid" to all residual segment names, while leaving the name of the material segment unchanged from the original. Conversely, a brute force approach could be used that analyzes the definition of each member function looking for er_entry() calls on behalf of the ER_Coordinator object discovered earlier. Regardless of the approach used, a list is maintained that keeps track

of all of the class's Early-Reply member functions and their corresponding residual segments. Any public member functions not in the list would be considered Non-Early-Reply member functions, and maintained in a separate list.

After identifying each Early-Reply class and their associated Early-Reply member functions, each Early-Reply class definition is analyzed to confirm that a nested class derived from ER_Task has been defined for each different Early-Reply member function. Each of these nested classes is further analyzed to confirm that call_residual() is to invoke the appropriate residual segment.

The last step of the rewrite verification involves checking that the definitions of the material segments, residual segments, and Non-Early-Reply member functions have been rewritten appropriately. The material segments (i.e. original member function) must create an ER_Task derived nested class on the heap as the first statement. Its constructor must contain a pointer to the host object (i.e. this) and to the contained ER_Coordinator. Any parameters needed for the residual segment must also be included in the constructor call. The very next statement must be a call to the ER_Coordinator's er_entry() that passes a pointer to the newly created ER_Task derived object. The statement just before the material segment's return statement must be a call to the ER_Coordinator's er_start() that also passes a pointer to the ER_Task derived object. Verification of residual segments only involves checking that unpacking of any optional function parameters has been done. Verification of Non-Early-Reply member functions is simply a check that a call to the ER_Coordinator's entry() is the fist statement.

If static analysis of the Early-Reply rewrite does not generate any errors, verification will be considered successful. At this point, a sufficient structure imparted by the Early-Reply framework has been verified so that if the coding guidelines are verified in the next phase, the program will be considered free from race conditions, deadlock, and starvation.

5.2.2 Coding Guidelines Verification

Because of the framework's structure and confirmation that it holds by the rewrite verification phase, the second and last phase of static analysis verification of the coding guidelines is greatly

simplified. This phase is focused solely on verifying the absence of race conditions and deadlock. Recall that with this framework starvation is not even possible. With this in mind, only the definitions of the material and residual segments need be analyzed:

- Material segments for potential deadlock.
- Residual segments for potential race conditions and deadlock.

A very useful property that the framework provides is that a residual segment is the only part of the program that can execute on an auxiliary thread. Verification of the rewrite procedure in the previous phase ensures that this property is indeed valid. This property allows us to focus solely on the residual segment for determining if an Early-Reply program is free from race conditions. Consequently, analysis on the residual segment will be stricter than it is for the material segment.

Another useful property of the framework is that Non-Early-Reply member functions do not need to be considered for coding guidelines verification. As long as the residual segments are deemed safe from race conditions, Non-Early-Reply member functions cannot pose a problem because they cannot execute at the same time as a residual segment from the same object. Moreover, since Non-Early-Reply member functions do not actually acquire the object lock, deadlock involving either the host object or another Early-Reply object is not possible.

Since the material segment cannot execute at the same time as a residual segment from the same object, analysis can allow aliased variables inside the material segment. Aliased variables can originate from the material segment's function parameters (i.e. references or pointers to objects) or from scopes outside class scope. However, the Early-Reply call cannot then be used inside a residual segment as a sub-object Early-Reply call. Generally speaking, aliased variables must not find their way into a residual segment. Analysis on the material segment is focused on detecting deadlock possibilities. Listed below are possible violations that must be checked in the material segment:

- To prevent deadlock, Early-Reply calls cannot be made within the material segment on behalf of the following objects:
    - Host object (i.e. this).

  - Reference or pointer to an object passed in as a function parameter.
  - Object of scope outside of class scope (i.e. global or namespace).

Analysis for the residual segment is much more involved than for the material segment, since this is the only place where code will be analyzed for potential race conditions. Analysis for deadlock will also be performed since it is possible in residual segments. Listed below are possible violations that must be checked in the residual segment:

- To prevent race conditions, no reference or pointer to an object whose scope is outside of class scope or passed in as function parameters can be assigned to the ER_Task derived object.
- To prevent race conditions, the return value from the material segment cannot be modified.
- To prevent race conditions, access to variables of scope outside of class scope are not allowed. This includes variables of global or namespace scope and accesses done indirectly through function calls. Early-reply calls on behalf of sub-objects of the host object can only be made if they are entirely self-contained (i.e. sub-object cannot have any kind of access to variables of scope outside the sub-object class scope).
- To prevent deadlock, Early-Reply calls cannot be made within the material segment on behalf of the following objects:
  - Host object (i.e. this).
  - Reference or pointer to an object passed in as a function parameter.

After static analysis is completed for all of the material and residual segments, the program can be deemed free from race conditions and deadlock. Recall that the absence of starvation has been taken care of by verification of the rewrite procedure in the earlier phase which confirms the framework's structure has been properly implemented.

6 EARLY-REPLY SIMULATION

6.1 Considerations

Despite the fact that concurrency can be an effective technique to improve a program's performance, it is often difficult to predict the magnitude of the potential performance gains, if any at all. With some programs, the performance gains may be so minimal that it is not worthwhile to expend the effort to parallelize them. Or worse yet, if the concurrency overhead is greater than the concurrency benefit, the program may run slower than it did before. Note that all concurrency approaches have this predicament and it is not unique to Early-Reply. To address this problem, an Early-Reply simulator program is developed to predict the performance gains that are possible for candidate Early-Reply programs.

The simulator program has two primary modes in estimating speedup for programs using the Early-Reply concurrency framework. The first mode is to estimate the theoretical best-case speedup that is possible that doesn't take the framework's overhead into consideration. The second mode is to estimate the actual expected speedup that includes the framework's overhead in its estimate.

6.2 Design/Implementation of Simulator

The simplistic way in which the framework is used makes it possible to implement such a simulator. Most of the major tasks that contribute to concurrency overhead are isolated within the framework and out of the control from users of Early-Reply, which makes it easier to determine their overhead contribution. For example, thread creation and destruction is handled by a thread pool that is completely encapsulated within the framework. Moreover, this overhead can be easily estimated since the framework creates and destroys threads only once per program.

An existing C++ simulator library called 'ADEVS' (A Discrete EVent Simulator) is used to help construct the Early-Reply simulator [26]. This library can be used for models described using

the Discrete Event System Specification [27]. A key feature of this specification is that the dynamic behavior of program using the Early-Reply framework can be modeled in terms of events.

An Early-Reply program can be modeled by breaking it down into function calls, and then representing the calls as ADEV events. In terms of trying to model the possible speedup achieved with Early-Reply, there are only four different kinds of function calls that are relevant as events:

- Material segment
- Residual segment
- Non-Early-Reply
- Unrelated

The material and residual segment events together represent the computation of an entire Early-Reply call, and must specify to which Early-Reply object they belong (represented by an index). A Non-Early-Reply event obviously represents the computation of a Non-Early-Reply call, and must also specify which Early-Reply object to which it belongs. An unrelated event is a 'catch-all' since it can represent any kind of computation that doesn't involve an Early-Reply object, even idle user time.

A program or program segment is modeled by a collection of events contained within an input file to the simulator program. Each type of event is specified with somewhat different parameters, but all of them must specify the event's execution time. The function call or calls that are represented by each event must have their execution time(s) measured in some way before using the simulation program. The event execution times are the basis for estimating possible speedup for either theoretical or actual mode.

To better estimate actual speedup when the overhead of the framework is considered, different potential time consumers of the framework must be measured. However, instead of measuring them outside of the simulation program and specifying the values as input parameters, they are computed automatically within the simulation program. This is possible because the Early-

Reply framework must be used in a specific way and is thus very predictable in how it is used. For a more accurate estimate of overhead, the simulation program should be run on the same computer on which the Early-Reply program will be run, and use the same concurrency primitives from the same concurrency library (e.g. pthread) within the simulation program as the ones used in the Early-Reply program.

Fortunately and by design, the interface for using the Early-Reply framework from the user program is minimal, which makes it easier to implement the contributors of overhead in the simulation program. Any time the actual mode is used when running the simulation program, the following overhead contributors will be included in the estimate:

- Thread pool creation and destruction (occurs only once per program)
- er_entry() call for Early-Reply material segments
- entry() call for Non-Early-Reply member functions
- er_start() at the end of material segments
- er_exit() at the end of residual segments

Each of these overhead activities are measured using calls to Unix's gettimeofday(). Estimating the thread pool overhead is done by measuring the time needed to create and destroy the desired number of threads. The remaining overhead involves calls to er_entry(), entry(), er_start(), and er_exit(), which make up the interface of ER_Coordinator.

Overhead for er_entry() involves creating a small ER_Task object, possibly performing a conditional wait if the Early-Reply object is locked, and another conditional wait if all of the threads from the thread pool are busy. By contrast for entry(), only a possible conditional wait needs to be considered if the Early-Reply object is locked. Overhead for er_start() involves signaling a waiting thread in the pool. While for er_exit(), a signal is sent only if there is a waiting member function of an Early-Reply object. It is important to note that the simulation program can decide whether to include any of the optional paths just mentioned in the overhead time estimation based on how the model of events actually interact. Figure 33 illustrates how the simulator is constructed.

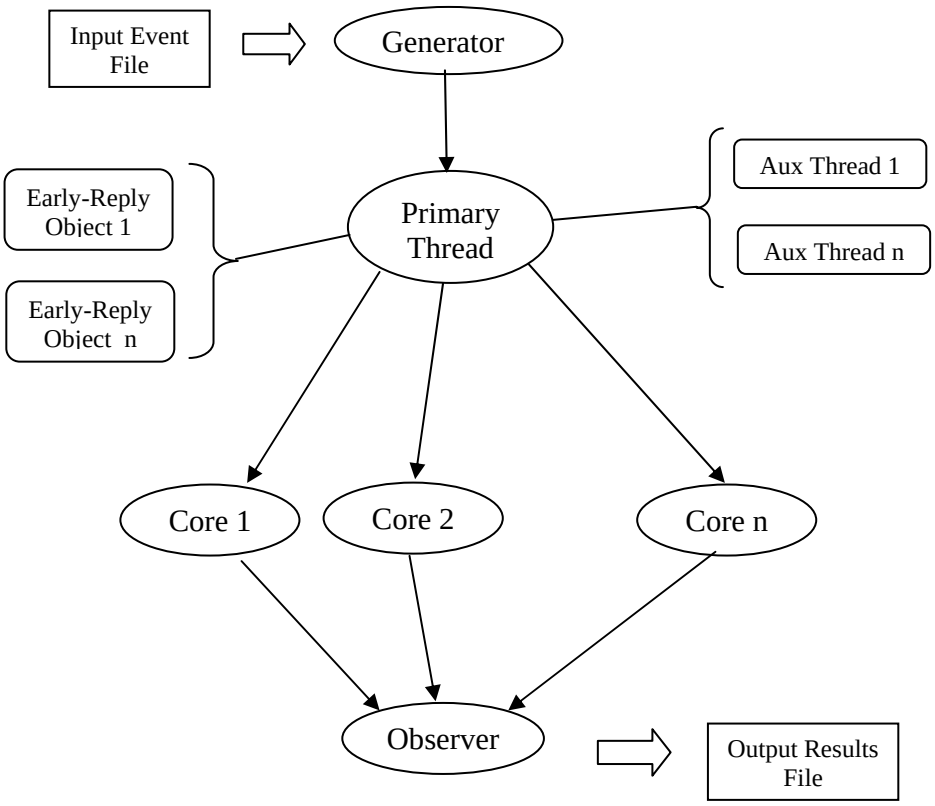

Figure 33. Early-Reply simulation program components.

Both the Input Event File and Output Results File contain a collection of the different types of events that model the computation of the Early-Reply program or program segment that they represent. Each line in the input file contains an identifier indicating its event type, along with its associated parameters that most importantly include the estimated computation time of the event. Each line in the output file contains the final timing results of each of the events specified in the input file. The parameters that can be specified for each type of event within the input file and the different timing values for each event in the output file will be discussed in detail in the 'Using the Simulator' section next. The Generator is an ADEV object that simply reads the input file and passes each of the events to the Primary Thread object when their specified start time is reached.

The Primary Thread is also an ADEV object that provides the critical functionality of advancing each of the events when appropriate as they come in from the Generator. Depending upon the

event, it can be held because there are no available Core objects, its respective Early-Reply object is locked, or an auxiliary thread is not available. As each event arrives from the Generator, it is put in an event queue and then de-queued when it is appropriate to advance. Listed below are the four types of events along with their associated restrictions on advancement:

- Unrelated – need only wait for an available core.
- Non-Early-Reply – must wait until its respective Early-Reply object is in an unlocked state and then for an available core.
- Material segment – must wait first until its respective Early-Reply object is in an unlocked state, then for an available auxiliary thread from the thread pool, and finally for an available core.
- Residual segment – need only wait for an available core, since its material segment cannot start until the object lock is released and an auxiliary thread is available.

Note that to make it easier to create the input file, the material and residual segment events are actually defined as one Early-Reply event. After the event arrives on the Primary Thread's event queue it is split up into its respective material and residual segment events.

Each Early-Reply and Non-Early-Reply event must specify the Early-Reply object number to which they belong so that the simulator can create an Early-Reply object for each unique object number encountered within the input file. Each Early-Reply object keeps track of the object lock along with other information that assists the simulation program.

The number of auxiliary threads and cores are specified as command line arguments. They are represented by creating the appropriate number of Auxiliary Thread and Core objects. The Auxiliary Thread objects are simply used to keep track of the residual segment events still held in Core event queues for thread pool availability. Core objects are ADEV objects that simulate the actual CPU processing. They hold the event until the simulation clock reaches the end of the event's computation time, and then advance the event to the Observer object for final bookkeeping.

6.3 Using the Simulator

The Early-Reply Simulator program involves:

- Decomposing the computation of the original program into events and measuring their computation times
- Composing the Input Event file
- Running the program and analyzing the Output Results File

The first step is the most difficult one because it requires mapping computation from the original program into a model of instances of one of the following three events:

- Early-Reply
- Non-Early-Reply
- Unrelated

Each different member function from the original program that will eventually become an Early-Reply member function must be analyzed to see how it can be split up into a material segment and residual segment. After this is done, the execution times for the two segments must be measured. Execution times must also be measured for each different potential Non-Early-Reply member function.

All of the remaining relevant computation from the original program can be modeled by the 'unrelated' events, as they can be used in many different ways. One very useful way for an unrelated event to be used is to model computation from large parts of the program, such as all of the computation before arriving at the first Early-Reply or Non-Early-Reply call. This means it can represent multiple function calls, one function call, or simply a set of statements that is not a function within itself. When trying to simulate interactive programs, an unrelated event can also represent user idle time between interactive actions. Special care must be taken so that unrelated events are fine grained enough when they are interleaved with the Early-Reply events to ensure the concurrency is appropriately modeled. As with the other two types of events, the execution times of each different unrelated event must be measured.

After the computation times from all the different events have been measured, they must be assigned to appropriate instances and put into the appropriate order within the Input Event File. For Early-Reply and Non-Early-Reply events, the appropriate Early-Reply object number must be assigned. A simple example is shown in Figure 34 to better illustrate how the input file is composed and how the simulator program is run.

Input Event File:

1	2	3	4	5
u	0.0	2.1		
e	0.0	1.3	1.1	0
u	0.0	0.5		
n	0.0	0.53		1
e	0.0	1.3	1.1	0
u	0.0	0.7		
n	0.0	0.25		0
u	0.0	1.0		

Input Event File Format:
Column 1: Event action ('e', 'n', 'u')
Column 2: Desired start time (seconds)
Column 3: Event computation time (seconds)
Column 4: Residual segment computation time (seconds)
Column 5: Early-Reply object number (0 through n)

Figure 34. Sample input event file and its format.

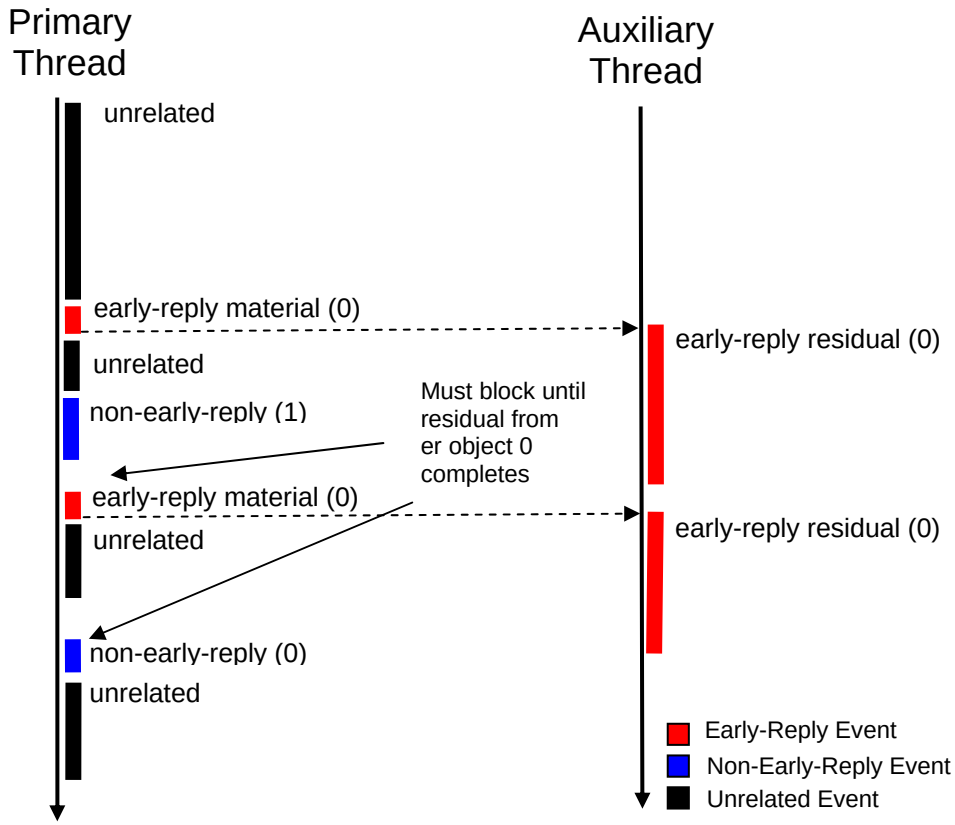

Figure 35. Pictorial view of simulator example.

As can be seen by the diagram in Figure 35, a performance gain is possible because a significant amount of computation can execute concurrently assuming that at least two cores are available. Note that the second Non-Early-Reply call must block until the lock is released from its corresponding Early-Reply object.

The syntax to run the simulation program is as follows:

er_sim o/a infile outfile numthreads numcores

The first argument is the program name. This is followed by the computation mode whether the program should compute optimal times without overhead or actual times that include the framework's overhead. The third and fourth arguments specify the name of the input and output

files respectively. The fifth and sixth arguments specify the number of total threads and cores to simulate respectively.

With this example there wouldn't be any benefit in using more than two threads or two cores. The resulting output file is shown in Figure 36.

Output Results File:

1	2	3	4	5	6	7	8	
u			0.0	2.1	2.1		0	
e	m	0	2.1	2.3	0.2	0	0	
e	r	0	2.3	3.4	1.1	0	1	
u			2.3	2.8	0.5		0	
n		1	2.8	3.32	0.52		0	
e	m	0	3.4	3.6	0.2	0	0	// block for prev residual
e	r	0	3.6	4.7	1.1	0	1	
u			3.6	4.3	0.7		0	
n		0	4.7	4.95	0.25		0	// block for prev residual
u			4.95	5.95	1.0		0	

Output file format:
Column 1: Event action ('e', 'n', 'u')
Column 2: Material or residual segment ('m', 'r', ' ')
Column 3: Early-Reply Object Number (0 through n)
Column 4: Actual event start time (seconds)
Column 5: Actual event finish time (seconds)
Column 6: Event computation time (seconds)
Column 7: Auxiliary thread number (0 through n)
Column 8: Core number (0 through n)

Figure 36. Sample output event file and its format.

Assuming that two threads and two cores are used, optimal execution time of the example program is 5.95 seconds. To compute the optimal speed up the simulator needs to be run the same as before except using only one thread and one core, which gives a speedup of 1.29.

One last item to note is that if a different concurrency library is used other than pthread, the simulator program can be edited by substituting the equivalent primitives in the appropriate places and recompiling.

7 EXPERIMENTS AND RESULTS

7.1 Test Programs

To demonstrate that the Early-Reply framework is capable of providing sufficient performance gains for candidate programs, the framework is applied to the following three different test programs:

- Decaf Compiler [28]
- Word Search in a Directory of Text Files
- Word Search in a Large Text File

The motivation for the first two programs is to improve overall execution time by improving throughput, while the third is to improve response time. Performance is improved for the Decaf compiler program by using Early-Reply to pipeline tasks, while the two word search programs use Early-Reply to pre-compute results that can be accessed instantly at a later time.

7.2 Experiment Environment and Procedure

The experiments were carried out using a Dell Inspiron 6400 laptop with an Intel Core Duo T2300 processor running at 1.66GHz with .99 GB of RAM. This computer is attractive for several reasons:

- It has the new Dual Core technology that will make parallel computing accessible to many.
- The computer is private so tests can be run with limited contention from other processes.
- Its memory is shared with Uniform Memory Access (UMA).

To qualify the performance gains achieved by all three test programs, we will employ the speedup formula that is typically used to evaluate parallel algorithms [2]. The resulting value of the speedup formula is unit less and is actually a ratio of execution times as shown below:

Speedup = (Sequential Execution Time) / (Parallel Execution Time)

Two different versions of each program are used to produce the speedup result; one sequential without any parallel programming constructs, and the other one parallel using the Early-Reply framework. Sequential execution time is measured with the unmodified sequential version running on one processor with one primary thread. Parallel execution time is measured with the version modified by the Early-Reply framework running on multiple CPU cores (or processors) with two or more threads. Because of the computer used for the experiments, no more than two CPU cores can be used simultaneously. Execution time can either be the program's total execution time or the time over a program segment. Unix's gettimeofday() system call is inserted in appropriate places within both versions of each program to record the actual execution times. The execution times are recorded while the operating system is in recovery mode to minimize the number of unnecessary processes that might compete for processor usage such as the X Server, desktop, etc..

The parallel version of each test program is modified so that it can accept a parameter to specify the number of auxiliary threads to create and use during execution. However, the performance benefit of using more threads is limited by the maximum number of available CPU cores (two) for the Decaf compiler test program, and by the pre-compute strategy used within the word search test programs. To illustrate the benefit of additional auxiliary threads, a different instance of the Decaf compiler parallel version is run using 1 through 10 auxiliary threads. It is the only one of the three that could potentially benefit from having more than one auxiliary thread, since it can have more than two concurrent operations.

To improve the accuracy of the execution timings, the sequential version and each instance of the parallel version are run 10 different times for each test program. Both an average is computed and the shortest execution time is recorded over these 10 different runs. Speedup is

then computed using the average results and the best execution times from the sequential and parallel versions.

7.3 Decaf Compiler

7.3.1 Overview

This test program involves parallelizing an instructional compiler written for the Decaf programming language [28]. Decaf is an abbreviated object-oriented language similar to Java and C++ that was developed at Stanford University for use in compiler courses. It is a revision of the SOOP programming language developed by Maggie Johnson and Steve Freund. Decaf is used in compiler related courses in many other top universities, including Texas A&M. This parallelized version of a Decaf compiler shows how Early-Reply can be used with a form of software pipelining to improve throughput (or shorter execution time).

7.3.2 Program Description

The test program was created by applying the Early-Reply framework to a student's final compiler project used in an undergraduate compiler course at Texas A&M. The final project was originally written for sequential execution on one processor. The original version includes the following functionality:

- Scans and parses a Decaf source file to create an internal representation (IR)
- Performs semantic analysis on the IR
- Performs a few minimal optimizations on the IR
- Generates MIPS assembly code so a program can actually be executed using the SPIM MIPS simulator
- Does not perform any linking

Performance gains are achieved with the framework by pipelining the front-end and back-end tasks of the compiler. This approach is effective only when there are multiple input source files to compile, so that back-end processing of earlier source files can occur in parallel with front-end

processing of more recent source files. The original project was required to process only one input source file, so the code was modified to accept multiple source files.

The framework is applied to the compiler project by creating an Early-Reply Compiler class that contains a member function compile(). The front-end functionality of scanning, parsing, and semantic analysis is placed in the material segment of compile(), while the back-end optimization and code generation is placed in the residual segment of compile(). No inter-procedural analysis is performed which means that each translation unit is self-contained. This allows separate Early-Reply Compiler objects to be created for each input file so that the residual segments of compile() can safely execute in parallel with material and residual segments from other translation units.

7.3.3 Effects on Speedup

Performance gains are achieved with the compiler test program by allowing a translation unit's residual segment (back-end computation) to execute simultaneously on an auxiliary thread with the material segments (front-end computation) from any subsequent translation units on the primary thread. Furthermore, since the translation units are independent from each other, additional performance gains are possible when two or more residual segments are able to execute simultaneously with each other. For this additional gain to be possible, the computation time of the residual segment must be longer than that of the material segment that follows. Additional auxiliary threads and CPUs must also be available for the residual segments to execute in parallel. Figure 37 illustrates how the parallelism can be exploited for 'n' input source files.

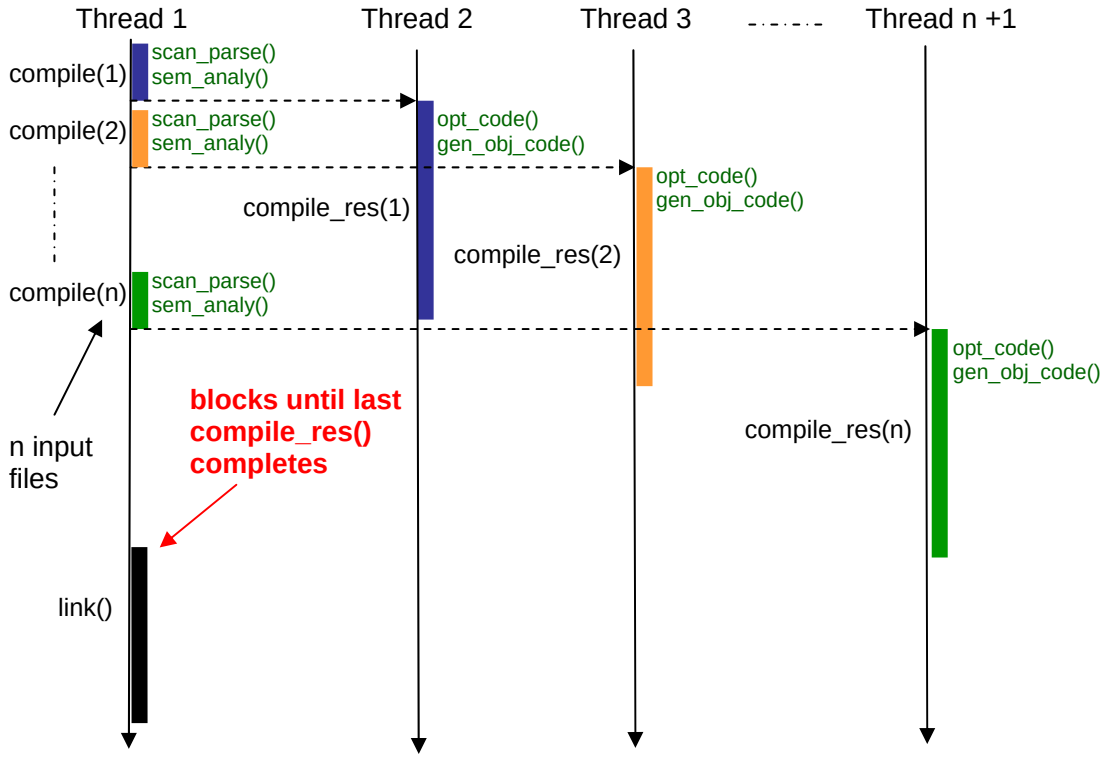

Figure 37. Pictorial view of compiler test program.

As can be seen in Figure 37, each material segment is responsible for the front-end computation of scanning, parsing, and semantic analysis, while the residual segment takes care of the back-end computation of code optimization and object code generation. In this diagram we show an approximate material segment residual segment ratio of 1:3, which makes it possible for each residual segment to overlap with multiple material segments and multiple residual segments.

In the case of this compiler test program, speedup can be affected by the following:

- Size of the input files
- Ratio of computation time for material segment and residual segment
- Number of input files
- Number of auxiliary threads and CPUs

The file size must be such so that the computation time of the residual segment is greater than any overhead of running it concurrently on an auxiliary thread. The larger the input file size the less impact the framework's overhead will have in degrading the speedup improvements.

If the computation time of the residual segment is longer than that of the material segment, more overlapping computation will be possible. The ratio of computation time between the material and residual segment is a dependent upon the functionality provided by the compiler itself. Scanning, parsing, and object code generation are fairly predictable, but semantic analysis and particularly code optimization can vary substantially from compiler to compiler. In the case of this test program, where almost no code optimization is done, the material / residual segment ratio will be smaller than a compiler that performs code optimization, thus reducing the amount of possible overlapping computation.

Assuming that both the input file size and material / residual segment ratio is held constant, speedup will be improved as the number of input files increases. This is simply due to the fact that a greater percentage of total execution time will involve overlapping execution.

If the computation time of the residual segments is less than or equal to that of the material segments, there isn't any benefit in having more than one auxiliary thread and more than two CPUs. This is because the material segments always execute on the primary thread and would only ever execute in parallel with one residual segment since each residual segment would complete before the next material segment starts. However, in most compilers the back-end computation time is sufficiently longer than that of the front-end, as is the case of this compiler.

7.3.4 The Experiment

Listed in Figure 38 are the different inputs and parameters used in the experiment for the compiler test program.

Input file size:	**60 kbytes**
Number of input files:	**5 and 10**
Number of auxiliary threads:	**Varies from 1 through 10**
Number of CPUs:	**2**

Figure 38. Compiler inputs and parameters.

Initial timing tests revealed that an input file size of 6 kbytes produced very marginal speedups. Since the ability to include header files within an input source file was not part of the final project's functionality, a larger input file size would reflect a more realistic scenario. Consequently, all timings are done with an input file size of 60 kbytes. Moreover, each input source file is exactly the same in content and size to simplify the interpretation of the results. This could cause some concern that there would be significantly fewer cache misses occurring than if each input file were different. However, since speedup is computed as a ratio between the sequential and parallel execution times the effect should be mitigated. In order to confirm that speedup is improved as the number of input source files is increased, timing tests are done with both 5 and 10 input files, each of the same size.

Given the experiment's input file size of 60 kbytes, the observed computation time of the material and residual segments are .033 seconds and .102 seconds respectively, producing a material / residual segment ratio of approximately 1:3. Since almost no code optimization is done in the test program and code optimization is typically a time consuming part of a compiler, material / residual ratios of 1:4 or 1:5 would not be unrealistic.

To determine the optimal number of auxiliary threads, a different instance of the parallel version of the test program is run with each using a different number of auxiliary threads (1 through 10). However in theory, because of the 2 CPU limitation any more than 2 auxiliary threads should not be beneficial with this test program.

7.3.5 Predicted Results

In order to determine both the theoretical and predicted actual speedup of this specific test program, the Early-Reply simulation program is used. The computed theoretical speedup represents the upper bound that does not take the framework's overhead into consideration,

while the predicted actual speedup does. To compute potential speedups the simulation program needs to know the actual computation time of the material and residual segments before the framework is applied. As stated earlier, the observed computation time of the material segment is .033 seconds and the residual segment .102 seconds.

The simulation program is first used to gain a better understanding of the theoretical upper limits of speedup that can be obtained for this compiler test program. This includes determining the optimal number of CPUs and what to expect when compiling different number of source files. Recall that the material / residual segment computation time ratio actually determines the number of tasks that can execute simultaneously (i.e. controls the depth of the pipeline). Since this ratio is fairly rigid, there is a limit on the number of CPUs that can be used to improve performance. The framework's overhead is not taken into consideration with the first run of the simulation program.

To determine the theoretical upper limit of useful CPUs and the affect of increasing the number of input source files, the simulation program is given different inputs that vary both the number of CPUs and the number of input source files with the results shown in Figure 39.

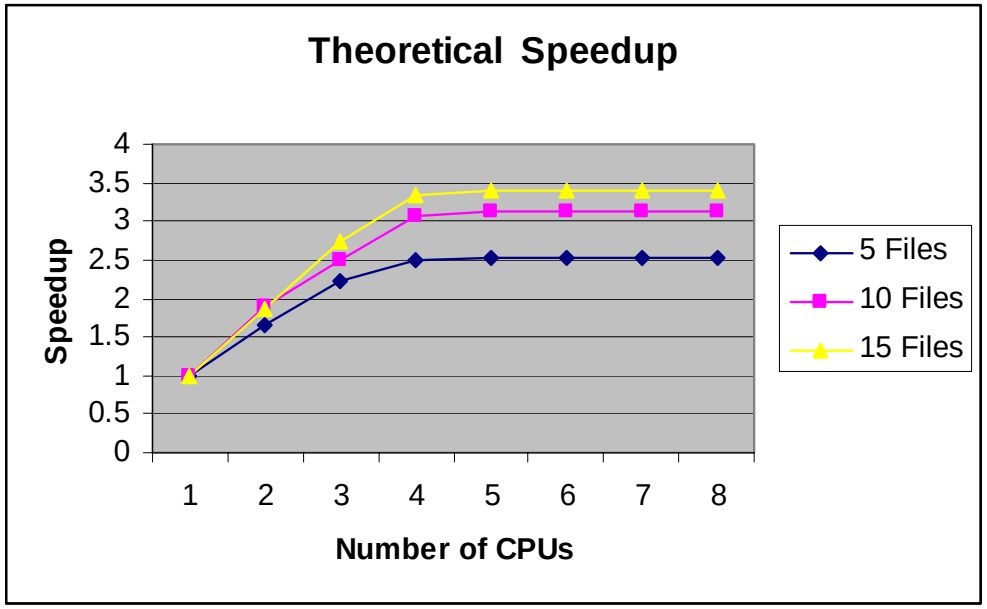

Figure 39. Theoretical speedup with an unlimited number of CPUs.

Given the content and size of the input source files, the results show that there isn't any additional benefit in using more than 5 CPUs no matter how many input source files are compiled. In fact, there isn't much improvement in using 5 CPUs over 4 CPUs. When compiling 15 source files an optimal speedup of 3.39 can be achieved with 5 CPUs compared with a speedup of 3.34 with 4 CPUs. Note that in this run of the simulation program, n + 1 threads are used for n CPUs to make sure enough threads are available. The results also show that better speedups are possible as more input source files are compiled. However, the improvement diminishes as the number increases.

Next, the simulation program will be used to predict potential speedups that include the framework's overhead. From this point onward, only 2 CPUs will be considered for both the simulation and actual framework runs since that is what is available on the test hardware for the measurements. Figure 40 and Table 1 show both the theoretical and predicted actual speedups possible for 2 CPUs when compiling both 5 and 10 input source files and varying the number of auxiliary threads.

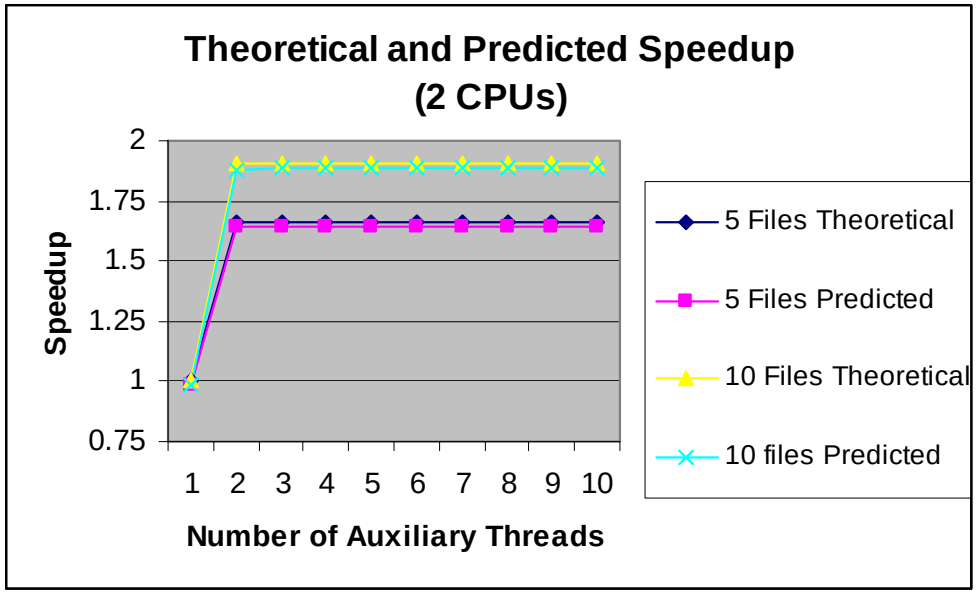

Figure 40. Theoretical and predicted speedup for 2 CPUs.

Table 1. Precise theoretical and predicted speedup values for 2 CPUs.

	No. of Input Source Files	Speedup for Number of Auxiliary Threads									
		1	2	3	4	5	6	7	8	9	10
Theoretic	5	1.0	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67
Predicted	5	.98	1.64	1.65	1.65	1.65	1.65	1.65	1.65	1.65	1.65
Theoretic	10	1.0	1.91	1.91	1.91	1.91	1.91	1.91	1.91	1.91	1.91
Predicted	10	.98	1.88	1.89	1.89	1.89	1.89	1.89	1.89	1.89	1.89

The results of the simulation program indicate that there shouldn't be any benefit with using more than 2 auxiliary threads when running on a 2 CPU computer, and that using 1 auxiliary thread (in addition to the primary) is not enough and even detrimental to speedup. This is shown by the predicted speedup results of .98 for both the 5 and 10 input file cases that include the framework overhead.

It may be a surprise for some that when using two threads (one primary and one auxiliary), theoretical and predicted speedups of 1.00 and .98 are computed for both the 5 and 10 input file cases. This is expected because the current framework will not allow a material segment to proceed past the er_entry() call until a thread is available (and reserved) from the thread pool for its corresponding residual segment. As long as one residual segment is executing, the only auxiliary thread is taken and prevents any subsequent material segment from proceeding which in turn makes the program behave as if it were sequential.

7.3.6 Observed Results

Timing tests are performed with both the sequential and parallel versions of this test program to determine the actual speedups that are possible. The only parameters that are varied for the experiment are the number of input files (5 or 10) and the number of auxiliary threads used for each instance of the parallel version. The results are shown in Figure 41, Figure 42, and Table 2.

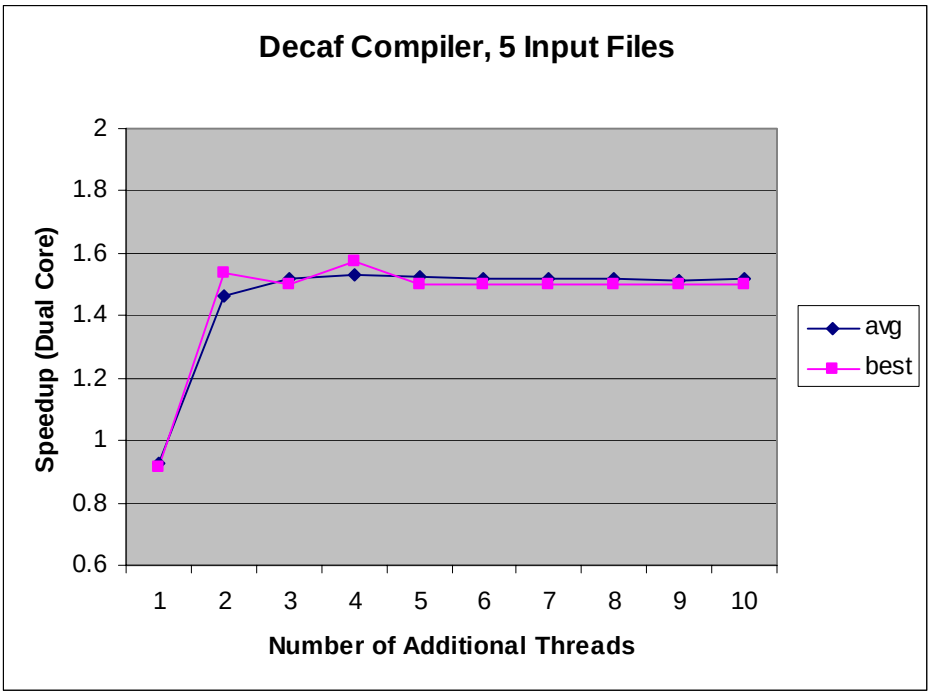

Figure 41. Observed speedup for 5 input files with 2 CPUs.

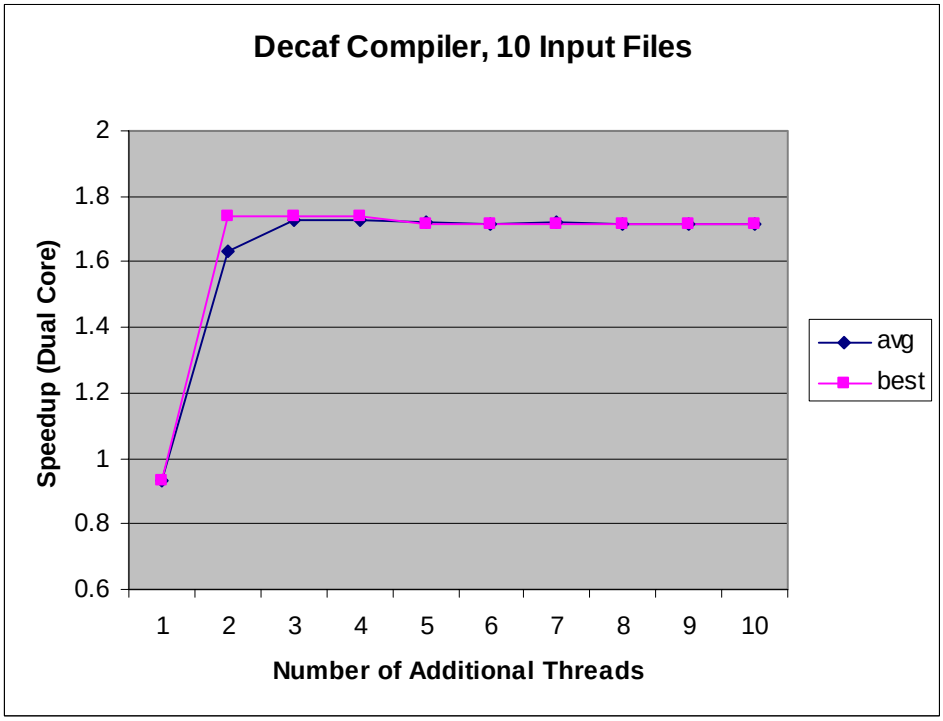

Figure 42. Observed speedup for 10 input files with 2 CPUs.

Table 2. Precise observed speedup values for 5 and 10 input files with 2 CPUs.

		Speedup for Number of Additional Threads					
Average or Best	No. of Input Source Files	1	2	3	4	5	10
Avg	5	.93	1.46	1.52	1.53	1.52	1.52
Best	5	.91	1.54	1.50	1.57	1.50	1.50
Std Dev	5	.005	.076	.057	.056	.054	.035
Avg	10	.93	1.64	1.73	1.73	1.72	1.71
Best	10	.93	1.74	1.74	1.74	1.72	1.72
Std Dev	10	.012	.064	.007	.005	.005	.007

Standard deviations for the sequential versions of the 5 and 10 input file cases are .0047 and .0067 respectively. The observed execution times for the sequential versions are in the range of .63 to .65 seconds for the 5 input file case and 1.27 to 1.29 seconds for the 10 input file case. Likewise, for the parallel versions, they range from .40 to .69 for the 5 input file case and from .73 to 1.4. Consequently, the standard deviations are all small enough to suggest reliable results.

7.3.7 Observations

As the observed results show, compiling 10 input files provides a substantial increase in speedup over compiling 5 files (1.72 vs. 1.52) when each file is of the same size. This confirms that speedup improves as a greater percentage of total execution time involves overlapping execution. Although not extremely close, the simulation program predictions for speedup of 1.64 and 1.88 that include overhead are not too far off the actual observed speedups of 1.52 and 1.72 for the 5 and 10 input source files respectively.

As predicted through the simulations, there isn't a noticeable performance benefit in using more than 2 auxiliary threads, as the observed speedups are virtually flat from 2 to 10 auxiliary threads for either 5 or 10 input file test runs. In contrast, a degradation of performance (speedup = .93) is observed when only 1 auxiliary thread is used because of the framework's overhead. Recall that the simulation program predicted a speedup of .98 that includes overhead when one auxiliary thread is used.

Note also that there is an anomalous difference observed between the average and best speedup results when using 2 auxiliary threads. In the 5 input file case the average speedup is 1.46 and the best is 1.54, while similarly with the 10 input file case, the average speedup is 1.64 and the best is 1.74. The reason for this is due to the runtime system's scheduling and to the current implementation of the framework. It is important to note that many runtime systems (including this one), non-deterministically schedule threads on the available processors. It was observed during the experiments that the same material or residual segment could go back and forth executing on different cores. Moreover, a material segment could begin to execute on a particular core, only to have a residual segment come in and briefly take over the core before allowing the material segment to come back and complete. This involves a lot of context switching and makes it difficult to control the optimal interactions between the material and residual segments.

To help make this a little more concrete lets consider the 5 input file case that creates only two auxiliary threads in the thread pool as shown in Figure 43.

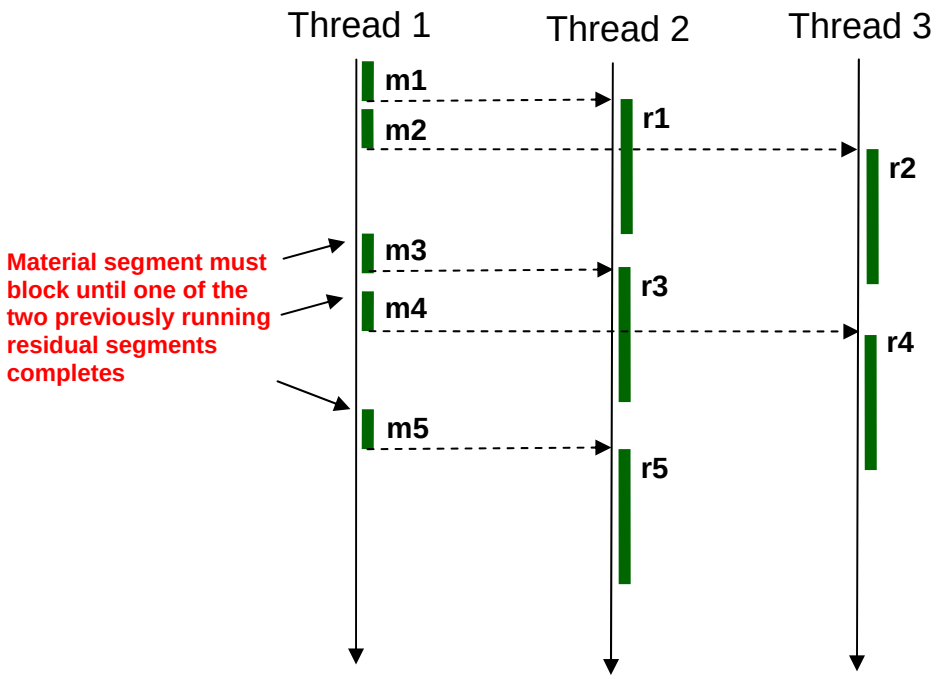

Figure 43. Compiling 5 input files with 2 auxiliary threads on 2 CPUs.

The framework causes problems for a similar reason that performance is degraded when using only one auxiliary thread. A material segment must block at its er_entry() call if an auxiliary thread cannot be reserved from the thread pool (because all are in use) for that particular function call. Notice that this can happen when the third translation unit is ready to be parsed, because there may be two residual segments that have yet to finish. Theoretically, there shouldn't be a difference in performance with a compiler instance that uses more than 2 auxiliary threads, because if the material and residual segments get scheduled at the right times, they will use the two processing cores just as efficiently as instances with more auxiliary threads. However, if for example, residual segment one takes a little longer than other times because of the way it is scheduled on the CPU, material segment three will be delayed from starting. This can cause a chain reaction in delaying the subsequent material and residual segments from starting, ultimately increasing the overall execution time. If three or more auxiliary threads are created in the thread pool, material segment three will be able to start right after material segment two (no chance of blocking), which saves the overhead of a potential block, and possibly improve the overall execution time when compared to the two auxiliary thread case.

There are actually at least three ways to remedy this problem. The first is to simply create more auxiliary threads in the thread pool, as this intermittent performance anomaly is not observed when using additional auxiliary threads. The Early-Reply pipelining approach used in this test program may be sensitive to the number of threads used. The second way to remedy this problem involves allowing material segments to begin executing even if there isn't a currently available auxiliary thread in the thread pool. This will move the thread pool blocking mechanism from the material to the residual segment. The third way is to allow the thread pool to expand automatically as new threads are needed and is discussed in section 8.3.

This particular test program makes an interesting example for using the Early-Reply framework for a number of reasons:

- Scales well if more CPUs are available
- Shows how Early-Reply can be used for a pipelining approach
- Requires only a few modifications when applying the framework rewrite
- Provides a good example of a common application that could benefit many

As stated earlier, the material / residual segment ratio would be closer to 1:4 or 1:5 than 1:3 for this compiler, when a more realistic amount of code optimization is done. This means that even better speedups could be observed if 3 or 4 CPUs are available.

Surprisingly, performing the Early-Reply rewrite on the existing sequential compiler project was a fairly easy task. The majority of the time involved was spent in understanding the layout of the code. Once understood, the modifications were minimal and quick. Furthermore, future modifications to the compiler should be no more difficult than before the rewrite. The Early-Reply framework (or a similar framework) could facilitate the migration of several sequential compilers over to parallelized versions.

What's more, since compilers are invoked repetitively many times during the course of a developer's day and are used by many people, a parallelized faster version could be a worthwhile endeavor. Moreover, the hardware is right in step since multi-core processors are now standard with most PCs sold today and are available for a reasonable price.

7.4 Word Search in a Directory of Text Files

7.4.1 Overview

In this test program, an option from the Unix ‘grep’ command is mimicked to show how Early-Reply can be used with another form of computation, pre-fetching. The program will perform a word search in a directory of text files specified by the user. As each occurrence of the word is found (if any), the filename in which it is found will be printed to standard output along with the line and column number. Like the Decaf compiler test program, the performance improvement is in increased throughput (or shorter total execution time).

7.4.2 Program Description

There are two primary tasks to perform with this program; read the contents of each file found in the specified directory into a container, and search each container for the specified word. Since it is desirable to give the user feedback as soon as possible, each container is searched as soon as the file is read. Consequently, the ideal strategy is to alternate the reading of each file with the word search. A sequential version of this approach is shown in Figure 44.

```
Dir_Read dr;
Word_Search ws;

dr.set_directory(dir);
ws.set_word(word);
while ( (vec = ds.read_file()) )  {       // 0 when no more files
   ws.find_word(vec);
}
```

Figure 44 . Sequential version of word search in a directory of files.

In the sequential approach, the word search cannot begin until after the file has been read into the container at the beginning of the loop. The performance of this code can be improved if the two tasks are made concurrent by converting read_file() into an Early-Reply member function. A pre-fetching strategy can be used by having read_file()’s material segment return the container of

the previously read file, while the residual segment pre-fetches a read of the next one. Since the material segment simply returns a value it can execute in constant time O(1). The Early-Reply approach is shown in Figure 45.

```
Dir_Read dr;
Word_Search ws;

dr.set_directory(dir);
ws.set_word(word);
dr.read_first _file();    // same as read_file(), but call is synchronous
while ( (vec = dr.read_file()) ) {
   ws.find_word(vec);
}
```

Figure 45. Early-Reply version of directory word search.

Notice that to get the pre-fetching started off correctly, read_first_file(), a Non-Early-Reply member function of Dir_Read, reads the first file of the directory before the loop begins. This allows the container to be searched immediately when the Early-Reply member function, read_file(), is called for the first time in the loop.

7.4.3 Effects on Speedup

Performance gains with the word search in a directory test program are achieved by executing the residual segment of the read_file() on an auxiliary thread simultaneously with the standard member function find_word() on the primary thread. The closer in execution time that find_word() and the residual segment of read_file() are to each other, the better the performance gain. Overall, the speedup of the word search in a directory test program can be affected by the following:

- Computation time ratio between residual segment of read_file() and find_word
- Size of the text files
- Number of input files

Speedup is most affected by the ratio of computation time between the residual segment of read_file() and that of the standard member function find_word(). The closer the ratio is to 1:1

the better the speedup. This is because these two functions are the only significant computation that takes place during the execution of the loop. Fortunately, as the size of the text file increases so will the search time, which keeps the two computation times fairly close to each other. Computation time of the residual segment read_file() is completely determined by the size of the text file. However, find_word() will depend upon how many occurrences of the specified word are found and how similar the prefix of the specified word is with that of the other words in the text file. At each occurrence of the search word, a time consuming print statement is executed that could significantly increase the computation time of find_word() if called many times. Furthermore, the search in find_word() will take longer when more words are encountered that match the prefix of the search word, because more characters must be examined to determine if a match is possible.

The size of the text files to be searched should be sufficiently large so that the residual segment computation time (i.e. reading the input file into a container) is greater than any overhead of running it concurrently on an auxiliary thread.

If both the input file size and read_file() / find_word() computation time ratio are held constant, speedup can be improved slightly as the number of input files increases. Like the compiler test program, this occurs because a greater percentage of total execution time is involved in overlapping execution.

7.4.4 The Experiment

Listed in Figure 46 are the different inputs and parameters used in the experiment for the word search in a directory test program.

Input file size:	**10 Mbytes**
Number of input files:	**5 and 10**
Number of auxiliary threads:	**1 and 2**
Number of CPUs:	**2**

Figure 46. Directory word search inputs and parameters.

A file size of 10 Mbytes is used for each file. A large file size is used to make sure that the test program's total execution time is out of the noise of the concurrency overhead. The larger file size is also used to reduce the complexity of the test program by approximating a more rigorous search function than a simple "word find", which executes extremely fast. For the same reasons as with the compiler test program, each input file is exactly the same in content and size. Timing measurements are performed with 5 and 10 input files in the search directory, in order to confirm that speedup is improved as the number of input text files is increased.

Unlike the previous experiment, where the material / residual segment computation time ratio is fixed according to the compiler's functionality, the read_file() / find_word() computation time ratio can be varied by input parameters. The ratio is varied by using two different search words, 'kermit' and 'themselves', where one word has a common prefix and the other does not so that both extremes can be observed. Many words within the input file begin with "th" which increases the execution time of find_word() and thus alters the read_file() / find_word() computation time ratio. Both words are placed only at the beginning and end of each text file so the analysis of the results can be simplified.

Since there can be at most two simultaneous tasks running, there is not any benefit in using more than one auxiliary thread or more than two CPUs. Nevertheless, test runs of the parallel version will include measurements with 2 auxiliary threads in addition to 1 auxiliary thread just to confirm that any additional threads are not beneficial.

7.4.5 Predicted Results

To determine the theoretical and predicted actual speedup for the test program when searching for 'kermit' and 'themselves', the execution times of read_file() and find_word() were measured and input into the Early-Reply simulator program. A time of .12 seconds was observed for read_file(), while times of .078 seconds and .11 seconds were measured for the 'kermit' and 'themselves' instances of find_word() respectively. Listed in Table 3 are the theoretical speedups that could be obtained using one additional auxiliary thread and two CPUs, assuming zero overhead for one instance searching for the word 'kermit' and the other 'themselves', and then varying the number input text files from 5 to 10.

Table 3. Theoretical speedup for directory word search with 2 CPUs.

Word to Search	No. of Input Files	Theoretical Speedup
Kermit	5	1.46
Kermit	10	1.55
Themselves	5	1.62
Themselves	10	1.76

7.4.6 Observed Results

Timing measurements were performed with both the sequential and parallel versions of this test program to determine the actual speedups that include the overhead of the framework. The parameters that are varied for the experiment are the search word (‘kermit’ and ‘themselves’), the number of input files in the directory (5 or 10), and the number of auxiliary threads used for each instance of the parallel version (1 or 2). The results are shown in Table 4.

Table 4. Observed speedups for directory word search with 2 CPUs.

			Speedup for No. of Additional Threads	
Average or Best	Word to Search	No. of Text Files	1	2
Avg	kermit	5	1.36	1.36
Best	kermit	5	1.36	1.36
Std Dev	kermit	5	.0031	.0031
Avg	kermit	10	1.42	1.42
Best	kermit	10	1.42	1.42
Std Dev	kermit	10	.0051	.0031
Avg	themselves	5	1.46	1.46
Best	themselves	5	1.48	1.48
Std Dev	themselves	5	.0067	.0071
Avg	themselves	10	1.55	1.55
Best	themselves	10	1.55	1.55
Std Dev	themselves	10	.0048	.0042

As expected, the results confirm that there is not any benefit in using more than one auxiliary thread and so this issue will not be discussed further. Standard deviations for the sequential versions of the 5 and 10 input file cases are .0069 and .0048 respectively when searching for ‘kermit; and .0042 and .0094 respectively when searching for ‘themselves’. Considering that the observed execution times are in the range of .73 to 1.01 seconds the standard deviations are all small enough to suggest reliable results.

7.4.7 Observations

The observed results confirm that the computation time ratio between read_file() and find_word() has a significant impact on speedup. This is evident because a search for ‘themselves’ produces a better speedup (1.46) for 5 text files than the speedup (1.42) searching for ‘kermit’ in 10 text files. The ideal ratio of 1:1 is approximated more closely when searching for ‘themselves’ than it is for ‘kermit’ because of the difference in execution times of 0.11 and

0.078 seconds respectively for a call to find_word(). In other words, the closer find_word()'s execution time is to 0.12 seconds, the better. The text file used in the experiment contains many words that begin with 'th' and not many that begin with 'k', so find_word() takes longer to execute when searching for 'themselves' than for 'kermit'.

Also as expected, a mild improvement in speedup is observed when searching in more input text files, since a higher percentage of the total execution time is involved in overlapping computation. It is also clear that there isn't any benefit in creating and using more than one auxiliary thread.

Although not extremely precise, the simulation program is able to estimate the expected speedups within a reasonable tolerance, especially the relative speedup improvements. For example, it is able to predict that a search for 'themselves' on 5 files produces a better speedup (1.62) than a search for 'kermit' on 10 files (1.55). These values are not too far off from the observed speedups of 1.46 and 1.42.

The test program makes an interesting example for using the Early-Reply framework for a number of reasons:

- Shows the advantage of returning a value early through pre-fetching
- Illustrates an Early-Reply object's self-blocking mechanism
- Needs exactly one additional thread
- Does not degrade performance when running on one CPU
- Demonstrates a very simple example that can achieve a reasonable speedup improvement

One very useful feature of Early-Reply that does not exist in many other concurrency approaches is that an Early-Reply function call can return a value just like any ordinary sequential function. In this test program, a pointer to a container is returned right away when the Early-Reply member function read_file() is called, allowing the program to use the pointer in the subsequent find_word() call, or terminate the loop if the pointer is null. However, just before read_file() returns, it asynchronously begins the read of the next file so that the read_file() call can return.

This ability to both synchronously return a value early and to asynchronously begin another operation is why an Early-Reply call is well suited for pre-fetching.

This example also highlights how an Early-Reply object can block itself to insure that race conditions do not occur. If a find_word() call completes before the residual segment of read_file() within the 'while loop' (which is likely), the er_entry() call within the read_file() member function will block further activity on the primary thread until the residual segment completes its pre-fetching operation.

Since the word search involves two alternating tasks, read_file() and find_word(), only one auxiliary thread needs to be created by the framework's thread pool. The framework reuses this same thread each time read_file()'s residual segment is called, and there cannot be two simultaneous calls to read_file() because of the self-blocking just described above. On the other hand, since there are only two alternating tasks, a drawback of this approach is that it will not scale well to machines with more than two CPUs.

A surprising benefit of this particular test program is that the performance will not be degraded by running the application on a computer with one CPU, even with the overhead of the framework. This is because the residual segment of read_file() involves a substantial amount of I/O that is typically offloaded to computer's DMA (Direct Memory Access) which allows the CPU to perform other tasks such as executing find_word(). A mild speedup of 1.04 was observed on a machine with 1 CPU. This kind of performance behavior is rare for most parallelized programs.

One last observation is that this program is a good "bare bones" example to illustrate many of the key features of the Early-Reply framework.

7.5 Word Search in a Large Text File

7.5.1 Overview

This test program mimics a typical ‘find’ command found in most text editors by continually searching for the occurrence of a specified word in a text file. Like the previous test program, a form of pre-fetching is used to achieve a performance gain. However, unlike the previous two test programs that highlight improvements in throughput, this test program shows how the framework can be used for improving response time over a program segment.

The motivation for using Early-Reply with this application is to have the ‘find’ respond immediately when the search word is found rather than a noticeable delay. A program that hangs up during user interaction can lead to reduced productivity, and is one reason why improving response time is important.

7.5.2 Program Description

The test program performs two independent tasks that alternate within a loop; a search for the next occurrence of the specified word within the text, and an interactive prompt asking if it is desired to continue the search for the next occurrence (if one was just found). The second task is particularly interesting because it involves human processing rather than computer processing. The sequential version of this approach is shown in Figure 47.

```
File_Read fr;
vector<string> vec = fr.read_file(file);

Word_Search ws(vec, word);

while ( ws.find_word() )  {
   if ( ! continue_search() )
        break;
}
```

Figure 47. Sequential version of word search in a large file.

The program begins by reading the contents of the text file into a container before entering the loop. The first task involved in the loop, find_word(), searches the container for the next occurrence of the specified word and displays the line number in the file where it was found (if successful). The second task, continue_search(), interactively prompts the user for feedback about whether or not to continue the search.

The performance of this code can be improved if the two tasks are made concurrent by converting find_word() into an Early-Reply member function. A pre-fetching strategy can be used by having find_word()'s material segment return a pre-computed boolean value indicating if the previous search instance was successful in finding another occurrence. Just before the material segment completes the residual segment is asynchronously launched to search for the next occurrence of the word and to pre-compute the boolean success value. Since the material segment simply returns a pre-computed value, it can execute instantly (i.e. constant time O(1)). The concurrent Early-Reply approach is shown in Figure 48.

```
File_Read fr;
vector<string> vec = fr.read_file(file);

Word_Search ws(vec, word);

if ( ws.find_first_word() ) {
   while ( rw.find_word() )  {     // breaks if user wishes to stop
      if ( ! continue_search() )   //   or no more occurrences
         break;
   }
}
```

Figure 48. Early-Reply version of word search in a large text file.

Notice that to get the pre-fetching started off correctly, find_first_word(), a Non-Early-Reply member function of Word_Search, searches the file for the first occurrence of the word (if it exists) and stores the boolean success value with the object before the loop begins. Then within the loop, find_word()'s material segment can immediately return the success value computed earlier, and then have its residual segment pre-compute the next success value by continuing the word search. The search performed by find_word() runs concurrently with the interactive prompt from the freestanding continue_search() function.

7.5.3 Effects on Speedup

This test program was chosen because it can show how the Early-Reply framework can also be used to improve response time for interactive programs. Moreover, with this particular program, improvements in throughput speedup can also be shown and are a direct result of improving the program's response time. Improved throughput is considered because it would be desirable to step through the entire document as quickly as possible, especially if there are many occurrences of the search word. As such, both forms of speedup will be examined within the experiment.

Performance gains with the word search in a text file test program are achieved by executing the residual segment of the find_word() on an auxiliary thread simultaneously with the freestanding function continue_search() on the primary thread. More importantly, however, is that continue_search() involves user reaction time to decide whether or not to continue. This reaction time should be considered execution time even though it is really human processing time.

The closer in execution time that the residual segment of find_word() and continue_search() are to each other, the better the performance gain. Since we are only interested in the speedup over the program loop, the ratio of find_word() and continue_search() execution times determine the resulting speedup. Consequently, only the following computation that can affect this ratio will be considered:

- Number of words between occurrences of the search word
- Prefix of the word to be searched
- User reaction time when indicating to continue the search

Since user reaction time can vary from one user to another and can vary within a particular session of the same user, it can tend to obscure the results when other items are varied. Therefore, neither the number of words between occurrences of the search word nor the search word is varied, to make the results of the experiment easier to interpret.

7.5.4 The Experiment

The experiment involves reading one very large text file (50 Mbytes) into a container, and searching the container for four equally spaced occurrences of the word ‘themselves’. The text file is large to simplify the experiment. A typical GUI text editor renders the pages near each occurrence of the found word, and so the rendering time can be approximated somewhat by increasing the number of words between each occurrence of the search word.

Two different sets of timing tests are performed to illustrate the performance benefits of using the framework; one for improved response time, and the other for improved throughput. For showing improved program response time, the time is measured over the duration of each call of find_word(). To show improved throughput, the time is measured from just after the text file is read into a container until the program terminates, and includes finding all four occurrences of the search word.

Similar to the previous test program, there can be at most one residual segment running concurrently with another. In this case it is because there is only one Early-Reply call within the loop, and each next invocation will block until the residual segment of the previous Early-Reply call completes. Consequently, there is not any benefit in using more than one auxiliary thread or more than 2 CPUs, so all tests involving the parallel version will use only one auxiliary thread.

7.5.5 Predicted Results

In the previous two experiments, the simulation program’s input event times were taken from measurements on their respective sequential programs. However, any computation that involves user reaction time (or human processing) that is measured from the sequential program may not be valid in predicting performance improvements in the parallel version. The sequential version includes a significant delay just before each display of the interactive “continue search” prompt. This delay is the word search, and can cause the user’s mind to wander for a second or two, which can cause another delay. To remedy this problem a conservative approach will be taken that may underestimate the potential for speedup, but at least won’t promise too much.

The simulation program will not be used to predict response time improvements since this involves measuring execution times across one function call, find_word(). Recall that in the sequential version, the interactive continue prompt is not displayed until after the current search has completed, whereas with the parallel version, the prompt is displayed immediately because of the pre-fetching.

The time for continue_search() is conservatively measured by taking the time difference from displaying a “Do you want to continue?” message to standard output until the user responds, without performing any word search. In this way the reaction time is optimal during the measurement because the user is able to respond instantly because the message is displayed instantly.

The simulation program is used to predict throughput speedup for all four complete iterations of the word search program. Input execution times of 0.46 seconds were measured for continue_search(), and 0.20 seconds for find_word() when searching for the word ‘themselves’. Like the previous test program, that can have at most two simultaneous tasks running, there is not any benefit in using more than one additional thread or more than two CPUs. Consequently, the theoretical speedup (with zero overhead) that is possible using one additional auxiliary thread and two CPUs is as follows:

$$\mathbf{Speedup}_{\text{sim\_no\_overhead}} = 2.64 / 1.72 = 1.53$$

As discussed earlier, the theoretical throughput speedup computed by the simulation program is conservative because the same input time of 0.20 seconds for continue_search() is used for both sequential and parallel runs. In reality, this optimal continue_search() call would take longer in the sequential version because the user’s mind would wander when the program’s response is sluggish.

7.5.6 Observed Results

For showing improved program response time, the time is measured over the duration of each call of find_word(), except the first call. The time interval over the first call is not measured

because the parallel and sequential versions are essentially the same. For the sequential version, the time is measured over the entire call to find_word(), whereas with the parallel version, the time is measured over just the material segment of find_word(). Since each occurrence of the word 'themselves' is spaced about the same throughout the text file, the response time is measured separately for each one as shown in Table 5.

Table 5. Observed response time speedup for word search in a large text file.

	Response Time Speedup for Occurrence Number of Searched Word		
Average or Best	2	3	4
Avg	687	672	785
Best	722	750	854
Std Dev	.000017	.000023	.000024

As expected, the response time speedup improvements are huge. This actually reflects what is observed when running the sequential and parallel versions, because in the first case there is a noticeable delay before interactive continue prompt appears, and in the second case the prompt appears immediately. Standard deviations of .00042, .00052, and .00042 are computed for each of the three cases of the sequential version of the test program. Considering that the observed execution times are in the range of .195 to .223 seconds for the sequential versions, and .00026 to .00032 seconds for the parallel versions, the standard deviations are all small enough to suggest reliable results.

For showing improved throughput the time is measured from just after the text file is read into a container until the program terminates. This time includes four complete iterations of continually searching for the word 'themselves' within the text file. The results are shown below.

$\mathbf{Speedup}_{avg} = 4.17 / 1.95 = 2.14$

Speedup$_{best}$ = 3.70 / 1.76 = 2.10

A standard deviation of .22 is computed for the parallel version of this experiment, while a standard deviation of .36 is computed for the sequential version. Considering that the observed execution times are in the range of 3.70 to 4.76 seconds for the sequential version, and 1.76 to 2.36 seconds for the parallel version, the variability is much higher than for the previous test programs. However, this is understandable because of the unpredictability of how fast a user can respond.

7.5.7 Observations

The observed response time speedup results of 672 to 785 confirm that the Early-Reply framework can indeed be effective at improving response time over a program segment. Most have experienced interactive programs that delay for much longer than a second or two before returning control back to the user. This framework could be an effective approach in eliminating or reducing substantially these delays.

As discussed earlier, a nice side effect is that by improving response time the throughput of the entire program can be improved. The observed throughput speedup result of 2.14 confirms that this is possible. Human processing time is why the observed speedup is greater than 2.0 seconds despite the fact that the measurements are run on a two CPU computer. Because of the human processing involved, this particular program could even show reasonable response time and throughput speedups on one CPU, as long as two threads are used.

Note that the actual observed speedups are better than the theoretical speedups. As discussed earlier, this is because an optimal response time for continue_search() was measured for the simulation program that does not consider the user's mind wandering. This was done to keep the theoretical speedup predicted by the simulation program as an upper bound.

What's more, since this kind of "find" functionality is found in most text editors and used very often during text editing sessions, a parallelized faster version could be a worthwhile endeavor. An Early-Reply type of approach could facilitate the transition.

8 DISCUSSION

8.1 Satisfaction of General Goals

In our efforts to improve concurrent programming, the following three general goals were identified and have been the primary driving force throughout this work:

- Provide a simpler way to write concurrent programs
- Improve the reliability of concurrent programs by preventing race conditions, deadlock, and starvation from occurring
- Exploit enough concurrency so that sufficient performance gains can be realized

This section examines how each of these goals has been achieved within the context of the test programs used in the Experiments and Results section.

8.1.1 Simplifying Concurrent Programming

The framework has been able to simplify the development of concurrent programs for both users and implementers of Early-Reply member functions. A framework interface that is able to hide the concurrency allows users to design and reason about client code that invoke concurrent Early-Reply calls using the more familiar sequential programming model. Whereas a minimal framework interface allows Early-Reply functions to be implemented quickly and easily through a simple re-write procedure.

Encapsulation and data abstraction from object-oriented programming are appropriate tools in the simplification effort. Encapsulation of data that can be accessed simultaneously into a special Early-Reply object permits the framework to provide implicit synchronization on accesses to this data, and implicit launching of concurrent tasks that operate on this data. Hiding both concurrency synchronization and initiation from the user is an absolute must for maintaining a sequential programming model for client code. Data abstraction facilitates the

construction of a minimal framework interface which simplifies the process of implementing Early-Reply code.

All three test programs demonstrated that client code could be developed using the sequential programming model. Inspection of the client code shows a complete absence of concurrency constructs, including any that provide concurrency synchronization or initiation. This lets users design and reason about their code using the more familiar sequential programming model which means that they don't have to worry about introducing errors such as race conditions and deadlock into their programs. Furthermore, very few changes to client code were required after the Early-Reply framework was applied. Client code changes were made in the Decaf compiler test program only because it was modified to accept multiple input source files. The pre-fetching "word search" test programs did require some adjustments to boot-strap the pre-fetching, but the changes were minor and more importantly left the client code looking like a sequential program.

The test programs also demonstrated the simplicity of implementing the Early-Reply objects and their associated member functions. Very little time was expended during the implementation phase primarily because the framework's re-write procedure makes it clear which steps are needed. Although the implementer needs to understand which sequential functions are going to be transformed into concurrent functions, he/she doesn't need to know how the framework is going to make it happen. This means an in-depth understanding of concurrency is not required to develop a concurrent program. What's more, the minimal interface of the framework required very few steps to be performed during the re-write procedure.

8.1.2 Making Concurrent Programming More Reliable

The framework has also been able to make concurrent programming more reliable by preventing data race conditions, deadlock, and starvation from occurring. These safety and liveness error conditions are unique to concurrent programming and are the reason it is so error prone. The improved reliability is achieved by combining the concurrency synchronization and initiation into one mechanism, which is then able to isolate where race conditions and deadlock can occur to a few known places. It was shown earlier that this isolation makes it possible for coding guidelines to be used to prevent occurrences of race conditions and deadlock. Race conditions

and deadlock are not possible if the guidelines are strictly followed by implementers of Early-Reply functions, and their absence can be guaranteed through static analysis verification. Furthermore, starvation is completely eliminated by the combined concurrency synchronization and initiation.

In short, the improved reliability of the framework is attributed to a robust synchronization mechanism that is able to narrow down the places where the error conditions can occur, and a simple set of coding guidelines that prevents the error conditions from occurring in those isolated places.

Although the improved reliability was not explicitly demonstrated by the test programs, it was shown implicitly in some ways. Neither deadlock nor starvation was experienced during any of the test runs, which are the easiest of the error conditions to detect. Race conditions are much more difficult to detect, and consequently need a more rigorous examination to demonstrate that they do not occur (see Future Work).

8.1.3 Exploiting Enough Concurrency for Performance Gains

Enough concurrency has been exploited by the framework so that sufficient performance gains are possible, which is after all the primary motivation for using concurrency. This is achieved in several different ways. First of all, the framework's concurrency overhead is minimized to include a greater number of applications whose performance can be improved through the framework. Additional concurrency can be exploited for certain applications by allowing sub-object fan-out. Less restrictive Early-Reply synchronization models are able to exploit even more concurrency if the application can make use of it. Finally, an Early-Reply simulation program was developed to predict potential performance gains, since introducing concurrency into an application doesn't always produce a better performing application. The framework's simplified structure makes it possible to develop a simulation program that is able to predict potential performance within a degree of error.

The three test programs confirmed that the framework is indeed capable of helping applications achieve sufficient performance gains in either improved throughput or response time. The

speedups of 1.52 to 1.72 achieved by the Decaf compiler test program on a 2 CPU computer are exceptional considering that the speedups are for task parallelism rather than for data parallelism. Higher speedups are more easily achieved with data parallel programs because they can distribute the workload of a parallel algorithm over multiple CPUs more optimally.

The pipelining form of computation used by the Decaf compiler test program may be one of the framework's better suited forms for achieving impressive speedups. This is because it can have more than two tasks executing concurrently, which allows it to scale better to computers with more than 2 CPUs. In contrast with the "word search" test programs, no additional benefit is achieved by running on more than 2 CPUs because there can be at most two concurrent tasks executing.

It was also demonstrated that the Early-Reply framework is capable of improving performance in response time in addition to throughput. The "word search" in a large text file test program produced huge speedups in response time over the interactive segment of the program. What's more these improvements could just as easily of been achieved on a single CPU since human time is involved.

As additional test program examples are discovered, programs that have inherent sub-object fan-out (i.e. Early-Reply objects calling other Early-Reply objects) could potentially demonstrate speedups on par with the Decaf compiler. Applications that could have multiple member functions of an object reading its data simultaneously could benefit by less restrictive synchronization models, such as a reader-writer model.

Finally, the Early-Reply simulation program demonstrated that it was capable of predicting if an application is able to exploit enough concurrency to achieve a sufficient performance gain. Performance speedups were predicted within a small margin of error for all three test programs. This could be very helpful in saving developer time that could be wasted trying to parallelize applications when a sufficient gain is not possible.

8.2 Other Possible Approaches

As with any new framework built from the ground up, there are many alternative ways to construct the framework. The pros and cons of several of the alternative approaches will be discussed here for the benefit of any future generations of the framework. Some of the more notable alternative approaches are listed below and subsequently discussed in more detail:

- Early-Reply as a specialized 'return' language construct
- Queues for blocked Early-Reply function calls
- Thread assignment from thread pool within er_start()
- Create er_task object on stack within material segment

Recall that Pike and Sridhar suggested that Early-Reply could eventually become a first class language construct for object-oriented programming languages as a specialized 'return' expression. The idea is not to split an Early-Reply member function call into two separate functions as done in this work, but rather simply insert an Early-Reply return statement between the material and residual computation as shown in Figure 49 for object X.

```
int X::er_function( )
{
  /*  material segment computes return value for variable y */
  early_reply y;
  /*  residual segment executes on aux thread concurrently with caller */
}
```

Figure 49. Early-Reply as a language construct.

The most important advantage of this approach is that it greatly simplifies the implementation of an Early-Reply member function by not having to define two separate functions (i.e. material and residual), to worry about inserting the framework interface calls, and to worry about passing parameters to the residual segment (since they are already in its scope).

The drawback of this approach is that it involves changing the programming language and the compiler to make it function properly. A non-standard technique is needed to wrap the code

following the Early-Reply statement into a block of code that executes on an auxiliary thread and terminates at the end of the function block. Note that a traditional fork() call would continue to execute past the end of the function block unless instructed otherwise. Moreover, all variables within the function's scope need to remain in the scope of the residual segment. Another challenge is how to have subsequent member function calls of object X block when it is necessary, especially when the state of object X is modified within the residual segment and could be executing simultaneously with another member function.

One criticism of the current framework is that if an Early-Reply member function call must block, it stops the calling thread's execution dead in its tracks, even if subsequent statements following the call could execute without any conflicts. To alleviate this problem, any blocking Early-Reply member function could be put on a queue for the given object and executed on another thread when appropriate, thereby allowing subsequent non-conflicting statements to execute. Each blocked Early-Reply member function would then execute as soon as the preceding Early-Reply or Non-Early-Reply member function completes.

This approach exploits more concurrency by allowing more potential simultaneous execution, which could lead to improved performance. However, there are a few drawbacks. The biggest problem is how to handle blocked Early-Reply member function calls that actually return a value. If the function call is blocked before the material segment can compute the return value, any subsequent statement that uses the value would have to be blocked until the value is computed. This case is shown in the example code in Figure 50.

```
int main( )
{
   X x;                              // Early-Reply object
   x.foo();
   // residual segment of foo() still executing when bar() called
   int b = x.bar();
   int sum = a + b;
}
```

Figure 50. Non-blocking Early-Reply calls that return values.

In this example, if the call to bar() is put on a queue for object ‘x’ to be executed as soon as foo()’s residual segment completes, the following statement involving the usage of the return value for ‘b’ would have to block until the bar()’s material segment completes. This approach is very similar to that of Futures (mentioned in the related work section) and would require special help from the compiler. Another drawback is that the calling sequence may be no longer sequential which breaks the sequential reasoning enjoyed so far. For example, there could be desired computational side effects that are expected to occur in a specific sequence that may now occur in a non-deterministic order because of this feature. These drawbacks along with the extra complexity of managing the queues were enough not to incorporate the feature into the framework.

Whether right or wrong, a decision was made early on to make thread assignments for the residual segment within the framework’s er_entry() rather than within er_start(). The reasoning used was that any blocking would occur at the beginning of the material segment and no where else. After the er_entry() call completes, both the material and residual segments will execute in their entirety without any further blocking.

The argument for performing the thread assignment within er_start(), is that all available threads in the pool could be in use when er_entry() is called, causing the call to block even if the object is not locked. By waiting until er_start() is called before performing thread assignment, the material segment would be allowed to complete, creating the possibility that a thread from the pool becomes available at this later time, saving an unnecessary stall. This alternative approach could be employed in a future generation of the framework.

8.3 Future Work

At the time of this writing, there are not any known Early-Reply implementations involving concurrency synchronization that could be used as a model for this framework. As such, the framework is built from the ground up as a first effort that could undergo many changes as it evolves into a more mature concurrency mechanism. Moreover, since the framework is designed to be used with a static analysis and transformation infrastructure such as The Pivot for automated verification and rewrite transformations, specialized tools to perform these tasks will

need to be developed. Listed below and subsequently discussed in more detail are potential future enhancements to the Early-Reply framework, along with some relevant tools that could be developed to support the framework:

- Automated verification of the framework's coding guidelines by static analysis
- Develop formal proofs for safety and liveness properties.
- Extend the thread pool class to create additional threads as needed
- Further simplify the rewrite procedure through a reduced framework's interface
- Automate the rewrite procedure through program transformations
- Propose Early-Reply as a C++ language or library extension

Developing a tool (or add-on) to automate verification of the framework's guidelines would likely be the most important work that could be done in the near future. This is because verification of the coding guidelines ensures that race conditions and deadlock cannot occur. Starvation cannot occur simply because of how the framework works. A static analysis infrastructure such as The Pivot is needed to ensure that the guidelines have been followed. Section 5's "Verification on the Absence of Error Conditions" discussed in more detail how the verification could be accomplished. Recall that verification through static analysis is viable as long as the concurrency is initiated and synchronized solely by the Early-Reply framework.

To demonstrate the theoretical soundness of the Early-Reply framework with respect to concurrency synchronization issues, formal proofs could be developed. Proofs could be developed to show that race conditions, deadlock, and starvation cannot occur within a program that uses the Early-Reply framework as its sole mechanism to initiate and synchronize the concurrency. It is possible that other mechanisms could be safely used for initiating and synchronizing concurrency within an Early-Reply program but they would have to be used with extreme care and would unnecessarily complicate the proofs. The informal proofs discussed in Section 5 would make a good starting place.

The thread pool developed for the current framework creates exactly the number of threads specified by the first constructor call invoked from an Early-Reply object within the program and no more. This means that if another thread is needed for a residual segment to execute and all

the current threads are in use, the Early-Reply function would block until a thread becomes available from the pool. A better way to implement the thread pool would be to initially create the number of desired threads, as done currently, but also have the thread pool create additional threads on demand whenever they are needed up to some limit. This would be a fairly easy enhancement to make to the framework, and was not necessary to make this work relevant.

There is no doubt that the rewrite procedure could be simplified further with only a small effort. A generic programming approach is the likely candidate that could help hide the framework interface calls (e.g. er_entry(), er_start(), etc.) and the parameter passing ER_Task nested objects. The rewrite procedure could be reduced down to simply splitting the material and residual segments into two separate function templates and possibly containing an ER_Coordinator sub-object. This additional effort was not done because it would have added additional development time and also obscured the functionality of the key elements of the framework that would still be required even if they were hidden.

Another approach that could be used to simplify the rewrite procedure is to use a program transformation tool from a static analysis infrastructure to automate the process. Some help would be needed to get the transformation tool started, such as providing the residual segment definition for each Early-Reply member function. If the material segment member function retains the original function name and the residual segment member function adds a pre-determined suffix (e.g. _resid) to the original name, the transformation tool could recognize the two segments as an Early-Reply member function and their host object as an Early-Reply object. Once recognized, the ER_Coordinator could be declred as a sub-object within the Early-Reply object, the ER_Coordinator interface member functions could be inserted in the appropriate places, and the ER_Task nested objects could be defined appropriately. This automated insertion of code would not only save the developer some time but could also ensure that the re-write process is mistake-free.

As discussed earlier in the “Other Possible Approaches” section, Early-Reply could be proposed as a language or standard library extension to an object-oriented language such as C++ language. A language extension would require a significant effort to provide a sample implementation and require even more effort to convince others that the language should be changed. A library

approach would require less effort and much of what was learned here could apply. Moreover, if wrapper functions are ever added as a language feature, the effort could be greatly simplified. Wrapper functions are member functions that could be executed at the very beginning and/or end of a specified set of member functions of the same class. A wrapper function is a perfect place for inserting calls to ER_Coordinator's interface functions er_entry(), er_start(), er_exit(), and entry().

9 CONCLUSION

Concurrency is an effective technique for improving the performance of many applications. However, the parallel programming model adds an additional dimension of complexity over sequential programming which can make programs more difficult to design and reason about, and make them more unreliable due to error conditions unique to concurrent programming. These error conditions include data race conditions, deadlock, and starvation. Moreover, the additional complexity of concurrency along with its inherent overhead can make it difficult to predict potential performance gains when introducing concurrency into a program, which can waste development time if gains are not possible.

To overcome many of these problems, our work takes an existing concurrency mechanism, “Early-Reply”, and uses it as a basis to develop a comprehensive framework that makes concurrent programming simpler and more reliable, while still exploiting enough concurrency to achieve sufficient performance gains. Moreover, the framework’s simple structure makes it possible to design and implement an Early-Reply simulation program to predict potential performance gains of programs that make use of the framework.

In its raw form, Early-Reply is based on the idea that a function can produce its final return value well before the function actually terminates. An Early-Reply function can be thought of as a function broken up into two parts; a material segment that executes on the same thread as the caller and includes statements up to where the function returns, and a residual segment that executes the function’s remaining statements asynchronously on another thread. If the residual segment can safely execute at the same time as any computation that follows the Early-Reply call on the calling thread a performance gain is possible. Early-reply is more suited for “task parallelism”, where different instructions or tasks operate on different data, as opposed to “data parallelism”, where the same instruction operates on different parts of the same data. What makes Early-Reply desirable for our work is that the concurrency is initiated inside the function body, hiding the concurrency from the caller, and is capable of returning a value just as any ordinary synchronous function call. Pike and Sridhar [15] recognized these properties of Early-Reply as a way to get the benefits of concurrent execution while being able to develop programs

using the simpler and more familiar sequential programming model. In this work we expanded on the efforts of Pike and Sridhar by providing an implementation of Early-Reply in the form of a framework that includes a robust synchronization mechanism since Early-Reply in its raw form does not include one.

We were able to demonstrate that the framework simplifies the development of concurrent programs for both users and implementers of Early-Reply member functions by applying the framework's rewrite procedure to the three test programs. Users of the framework are able to design and reason about client code that invokes Early-Reply calls using the more familiar sequential programming model. This is because of the desirable properties of Early-Reply mentioned above and because all concurrency constructs are hidden inside the implementation of either the framework or the Early-Reply function. Furthermore, implementers of Early-Reply functions are able to transform a sequential program into a concurrent one with very little effort. A minimal framework interface makes this possible through a brief rewrite procedure.

The Early-Reply framework has been constructed to make concurrent programming more reliable by preventing data race conditions, deadlock, and starvation from occurring. The improved reliability is primarily achieved by initiating the concurrency only after program control passes safely through the framework's synchronization mechanism. This property isolates occurrences of race conditions and deadlock to a few known places. The isolation makes it possible to formulate a simple set of coding guidelines that can be used to prevent occurrences of race conditions and deadlock in those known places. The two error conditions cannot occur if the guidelines are strictly followed by implementers of Early-Reply functions. What's more, these same guidelines can be used to verify the absence of the error conditions through static analysis. The framework eliminates the possibility of starvation occurring because it does not use a prioritized thread scheme and does not do any scheduling of the threads.

Performance analysis on the three test programs confirmed that sufficient performance gains are achievable when using the framework to improve either throughput or response time on a 2 CPU computer. Moreover, the framework achieved speedups with two different forms of computation that are conducive to concurrency; pipelining and pre-fetching. Speedups of 1.52 to 1.72 were achieved by the Decaf compiler test program using a form of pipelining to improve throughput.

These speedup results are noteworthy because they are almost in the range for what would be expected by a data parallelism approach to concurrency, which is easier to optimize. The "word search" in a directory test program used a form of pre-fetching to obtain improvements in throughput speedup of 1.36 to 1.55. The "word search" in a large text file test program also used pre-fetching to obtain improvements in throughput speedups of 2.14, but was also able to achieve response time speedups in the 600s to 700s. This test program demonstrated that when human processing is involved, speedups greater than the number of CPUs are possible, especially when measuring over short program segments for response time improvements.

To make it easier to predict potential performance gains when designing and implementing concurrent Early-Reply programs, an Early-Reply simulation program was developed. Simulation results on all three test programs confirmed that it was possible to predict potential performance gains within a small degree of error. The program simulates the calling sequences and execution times for both the sequential and Early-Reply versions of a given application so that potential speedup can be determined. For Early-Reply versions, the framework's concurrency overhead can be included.

The Early-Reply framework could prove to be a viable alternative in the future over conventional approaches of developing concurrent programs primarily because of its simplicity and improved reliability with respect to concurrency error conditions. The ability to design and reason about concurrent programs as if they were sequential while still achieving the performance benefits of parallel programming is a strong motivation for using the framework. Furthermore, it's also assuring that a developer or static analysis tool can use the framework's coding guidelines to ensure that race conditions and deadlock do not occur within a program that uses the framework. Considering the rise in availability of multi-core processor computers today, the fact that programs have to be explicitly parallelized to take advantage of them, and the lack of experienced concurrent programmers, this type of approach could be even more relevant.

REFERENCES

[1] R. Sebesta, *Concepts of Programming Languages*, Pearson / Addison Wesley, 2004.

[2] J. JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[3] B. Kernighan and D. Ritchie, *The C Programming Language (Second Edition)*, Prentice-Hall, 1988.

[4] B. Stroustrup, *The C++ Programming Language (Special Edition)*, Addison-Wesley Longman, 2000.

[5] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Lanuage (Third Edition)*, Addison-Wesley Longman, 2000.

[6] M. Hall, S. Amarashinghe, B. Murphy, Shih-Wei Liao, M. Lam, "Interprocedural Parallelization Analysis in SUIF," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 4. July 2005.

[7] S. Rus, L. Rauchwerger, "Hybrid Dependence Analysis for Automatic Parallelization", Technical Report, TR5-013, Department of Computer Science, Texas A&M University, Nov 2005.

[8] M. Phillippsen, "Imperative Concurrent Object-Oriented Languages," Technical Report, International Computer Science Institute, TR95-049, Berkeley, CA, Aug.1995.

[9] F. Pagan, *A Practical Guide to Algol 68*, Wiley Series in Computing, John Wiley & Sons, 1976.

[10] E. W. Dijkstra, "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands, 1965, Reprinted in *Programming Languages*, F. Genuys, editor, pp. 43-112, Academic Press, New York, 1968.

[11] R. Halstead, Jr, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, October 1985.

[12] Mentat Research Group, "Mentat 2.5 Programming Language Reference Manual," Technical Report, University of Virginia, Charlottesville, VA, 1995.

[13] G. Smolka, M. Henz, and J. Wurtz, "Object-Oriented Concurrent Constraint Programming in Oz", *Principles and Practice of Constraint Programming*, P. van Hentenryck and V. Saraswat, editors, pp. 29-48, The MIT Press, 1995.

[14] G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[15] S. Pike and N. Sridhar, "Early Reply Components: Concurrent Execution with Sequential Reasoning," *Proc. of the 7th International Conference on Software Reuse*, vol. 2319 of LNCS, pp. 46-61, Springer-Verlag, April 2002.

[16] M. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, 2000.

[17] Doug Lea, *Concurrent Programming in Java*: *Design Principles and Patterns*, Addison-Wesley, 1997.

[18] G. Andrews and R. Olsson, *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, 1993.

[19] M. Scott, "The Lynx Distributed Programming Language: Motivation, Design, and Experience," *Computer Languages*, vol. 16, no. 3/4, pp. 209-233, 1991.

[20] H. Attiya and J. Welch, *Distributed Computing, Fundamentals, Simulations, and Advanced Topics*, John Wiley & Sons, 2004.

[21] C. A. R. Hoare, "Monitors: An Operating System Structuring Mechanism," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, Oct. 1974.

[22] M. Singhal and N. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw-Hill Inc., 1994.

[23] B. Meyer, "Applying Design by Contract", *Computer (IEEE*), vol. 25, no.10, pp. 40 -51, October 1992.

[24] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*, The MIT Press, 2001.

[25] A. Silberschatz and P. Galvin, *Operating Systems Concepts*, Addison Wesley Longman, 1998.

[26] A. Muzy and J. Nutaro, "Algorithms for Efficient Implementations of the DEVS & DSDEVS Abstract Simulators", *Proc. of the 1st Open International Conference on Modeling & Simulation*, pp. 273-279, June 2005.

[27] Hild, D., H.S. Sarjoughian, B.P. Zeigler, "Distributed Object Computing: DEVS-based Modeling and Simulation," *Proc. SPIE, Enabling Technology for Simulation Science III*, vol. 3696, pp. 147-157, June 1999.

[28] J. Zelenski, *Decaf Programming Language Specification*, Stanford University, 2001.

VITA

Stephen Wendell Cook received his Bachelor of Science degree in geophysical engineering from the Colorado School of Mines in Golden, Colorado. He received his Master of Science degree in computer science from Texas A&M University in May 2007. His research interest includes programming languages and their tools, along with distributed and parallel computing.

Mr. Cook can be reached at Southwest Research Institute, 6220 Culebra Road, San Antonio, Texas 78228. His email address is scook@swri.org.