Early Reply: A Basis for Pipebranching Parallelism with Sequential Reasoning[1]

Scott M. Pike
Nigamanth Sridhar

OSU-CISRC-10/01-TR13

Copyright © 2001 by the authors.

[1] A newer version of this paper appears in the Proceedings of the 8^{th} International Conference on Software Reuse (ICSR-8). Apr 2002. Austin TX

Abstract

Pipelining is a hardware technique for realizing instruction-level parallelism during program execution. To minimize pipeline stalls due to data hazards, forwarding is used to make operands available to future instructions as soon as they are produced by earlier instructions. Thus, forwarding partitions the pipeline into two phases: the *material computation* required to produce the results, and the *residual computation* required to complete the instruction. We extend this distinction to software components by introducing Early-Reply as a basis for component-level parallelism. In general, a component method completes its material computation as soon as its postcondition is satisfied. At this point, the output parameters can be forwarded to the client caller using Early-Reply without relinquishing control to complete the method. This enables the client to proceed in parallel with the residual computation of the component method. Since residual computations cannot violate their postconditions, Early-Reply components enable layered applications to be designed and reasoned about as sequential programs, despite their potential for exploiting physical concurrency at run-time. When Early-Reply components are composed hierarchically, higher-level calls can cascade to form a superlinear pipeline of method invocations on lower-level subcomponents. This potential for *pipebranching* parallelism can realize order-of-magnitude improvements in method response time among concurrently executing component operations.

Keywords. Component-based software, sequential reasoning, design reuse, concurrency, parallelism, pipelining, performance analysis

1 Introduction

Pipelining was a performance breakthrough in computer architecture [1, 2]. Rather than executing instructions as atomic units of work, pipelining subdivides the work of each instruction into smaller steps called pipe stages. This enables *independent* stages of consecutive instructions to be overlapped in parallel execution. The pipe stage is the atomic unit of physical work in the pipeline, but as a purely implementation-side technology, it does not compromise the client view of instructions as the atomic unit of *logical* work. Consequently, programmers benefit from the potential speedup of instruction-level parallelism [10] without having to reason about the concurrency implied by the parallel execution of overlapping pipe stages.

Pipelining by itself, however, does not always deliver the ideal speedup. Overlapping execution is only possible when physically concurrent pipe stages are causally independent. Hazards may arise that force future instructions to stall until dependencies on earlier instructions have been resolved. One of the most common dependencies is the Read After Write (RAW) data hazard, illustrated by the following instruction sequence:

$$\mathsf{X} := \mathsf{X} - 1$$
$$\mathsf{Y} := \mathsf{X} > 0$$

Figure 1 below diagrams the execution of this instruction sequence on a processor with four pipe stages: Instruction Fetch/Decode; Operand Fetch; ALU; Result Write-back. When the second instruction is scheduled to fetch its operands in clock cycle 3, the value in memory for operand X is stale. The required value has been changed by the preceding instruction, but it is not yet available until the first instruction writes the new value of X back to memory. Since this does not happen until clock cycle 4, a RAW data hazard precipitates a pipeline stall.

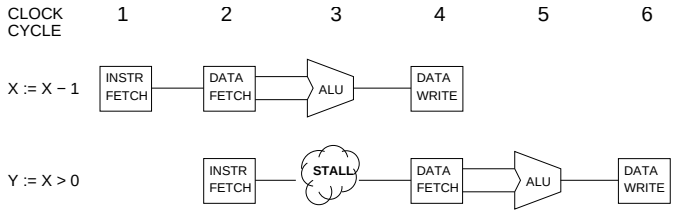

Figure 1: Pipelined execution with RAW data stalls

This stall can be avoided by using a technique called *forwarding*. After evaluating $\mathsf{X} - 1$ in the first instruction, the ALU feeds this result back to itself, so that it is available for immediate use by the next instruction. Forwarding the result as soon as it becomes available enables the second instruction to proceed, rather than stall, despite the parallel execution of the first instruction's residual write-back stage. The new execution using result forwarding is illustrated in Figure 2.

As with hardware, so too with software. Implementations of terminating component operations can be subdivided into two phases, corresponding to the *material computation* required to satisfy the operation's postcondition, and the *residual computation* required to complete the operation *in toto*. The residual computation often reëstablishes component invariants, which may involve restructuring data, deallocating memory, or other organizational activities. In principle, a client should be allowed to proceed with execution as early as results satisfying the postcondition are determined. By definition,

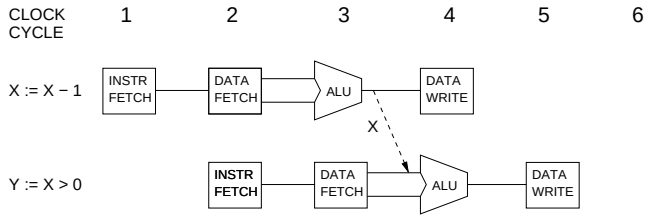

Figure 2: Pipelined execution using result forwarding

these results are available upon completing the material computation. Clients that are forced to block further during the residual computation of an operation are analogous to instructions that are forced to stall due to pipeline dependencies on earlier instructions.

In this paper, we present a case study that extends the technique of result forwarding to the domain of software componentry. We develop this technique as a means of realizing the performance benefits of implementation parallelism, *without* compromising the reasoning benefits of sequential client code. In Section 2, we introduce Early-Reply as an abstract construct to decouple data flow from control flow in sequential programs. Further characterization of Early-Reply is presented in Section 3.In Section 4, we illustrate the use of Early-Reply in exploiting component-level parallelism by pipelining method invocations in layered components. Section 5 analyses the worst-case response time of Early-Reply components. In Section 6, we use a Set component as an example to show how the response time of idle components can be optimized. An approach to further minimizing the amount of client blockage time is described in Section 7. We show how Early-Reply can be used as a lightweight approach to transforming off-the-shelf components into Early-Reply implementations in Section 8. We introduce the notion of pipebranching parallelism in Section 9. Some directions of related research are listed in Section 10. We summarize our contributions, outline a few areas of future work and conclude in Section 11.

2 Decoupling Data Flow from Control Flow

Traditional software design — informed by a preponderance of programming languages — couples the flow of data with the flow of control. As such, component implementations cannot return data to clients before relinquishing control to execute further statements. This entanglement of data and control rules out many efficient implementations for exploiting parallelism. To separate these concerns, we introduce Early-Reply as an abstract construct for decoupling data flow from control flow.

In our first example, we consider the implementation of a parameterized Stack component. We model the abstract value of type Stack as a mathematical string of Item, where the parameter Item models the type of entry in the Stack. Furthermore, we assume that when a variable x is declared, it is created with an initial value for that type, which we specify using the predicate Initial(x). The initial value for a variable of type Stack is the empty string, denoted by <>. The operator $*$ denotes string concatenation; the string constructor <x> denotes the string consisting of the single item x; the length operator |s| denotes the length of string s; and x :=: y denotes the swap operator, which exchanges the values of x and y. The pre- and postcondition specifications for the usual stack operations Push, Pop and Length are shown in Figure 3, where $\#\alpha$ in a postcondition denotes the incoming value of α, and α denotes its outgoing value. By convention, we view the left-most position in the string as the top of the Stack. Also, we use the distinguished parameter self to denote the component instance through which

the method was invoked, which, in this example, is a component of type Stack.

op Push (Item x)
precondition: true
postcondition: Initial(x) **and** (self = <#x> * #self)

op Pop (Item x)
precondition: self ≠ <>
postcondition: #self = <x> * self

op Length (): int
precondition: true
postcondition: Length = |self|

Figure 3: A specification of Stack<Item>

Figure 4 shows a common representation of Stack using a singly-linked list of nodes, plus an auxiliary field for length. Each Node contains two fields: a data field of type Item, and a next field which points to the next node in the linked list. The two data members of the representation are self.top (which points to the first Node in the linked list), and self.len (which is an int denoting the length of the Stack).

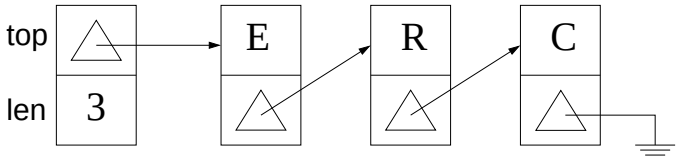

Figure 4: Representation of S: Stack<char> with the abstract value <E, R, C>

Informal descriptions of the representation invariant and abstraction function for this example are shown below. The invariant determines the legal state space of representation values, which, in this case, constrains self.len to be non-negative, and self.top to point to a null-terminated, singly-linked pointer structure. The abstraction function, **AF**, is a function from the representation state space to strings of Item. This function determines how to interpret a legal representation as an abstract value of the Stack it represents.

Representation Invariant: self.len ≥ 0 **and**
self.top points to the first node of a singly-linked list containing self.len nodes of type Node, with the next field of the last node being Null.

Abstraction Function: AF (self) =
the string composed of the data fields of the first self.len nodes in the singly-linked list pointed to by self.top, concatenated together in the order in which they are linked together.

Figure 5 shows a common implementation of Push and Pop in which the flow of data is coupled to the flow of control. That is, the formal parameter values of each method invocation are returned to the caller only upon relinquishing control at the termination of each method. These are *blocking* implementations in the sense that the caller is forced to block for the entire duration of the method body. The Length operation, which we omit for brevity, simply returns the current value of the member variable self.len.

```
op Push (Item x)
begin
1:    Node p;
2:    p := new Node;
3:    x :=: p.data;
4:    p.next := self.top;
5:    self.top := p;
6:    self.len++;
end
```

```
op Pop (Item x)
begin
1:    x :=: self.top.data;
2:    Node p := self.top;
3:    self.top := p.next;
4:    delete p;
5:    self.len--;
end
```

Figure 5: Blocking implementations of Push and Pop

Consider the implementation of Push in Figure 5. The first two statements declare and allocate a variable p of type Node, which is created with its data field as an initial value for the type Item; that is, the predicate Initial(p.data) holds. In the third statement, the formal parameter x is swapped with p.data, which is the data field of the new Node. Upon completing this swap operation, the postcondition for Push is satisfied with respect to the value of the formal parameter x. That is, x satisfies the postcondition requirement Initial(x). At this point, we claim, it is no longer necessary to keep the client waiting for the Push operation to complete in the following sense: there is no *client-observable* distinction between returning the value of x now — as opposed to at the end of the method body — so long as the Stack instance defers servicing additional method invocations until after the remainder of the Push operation has executed to completion. Similar remarks apply to the implementation of Pop, but even earlier: the postcondition of Pop is satisfied with respect to the formal parameter x immediately after completing the first statement, whereupon x = #self.top. Again, we claim, it should be unnecessary for the client to block for the remainder of the method body. But why should this be the case?

A key insight of the foregoing remarks is that the final (i.e., postconditional) value of the parameter x in each method is produced early on, *and is never changed thereafter.* Moreover, the fact that the parameter x is never even *used* after its final value is determined underscores the independence of subsequent client activities involving x, and any further computations by Push or Pop. To exploit this independence, we introduce an abstract construct called Early-Reply, which returns the current values of the formal parameters to the client caller, and then continues with execution until the method body has completed. Thus, Early-Reply separates the flow of data back to the client from the flow of control within the method body. Figure 6 shows how Push and Pop can be implemented using Early-Reply to reduce response time, and to exploit the potential for parallel execution by overlapping client and method computations.

In Figure 6, all statements preceding the Early-Reply in each operation represent *material* computations; that is, code required to establish the postcondition with respect to the formal parameters. Statements subsequent to each Early-Reply represent *residual* computations, insofar as they: (1) maintain the postcondition with respect to the formal parameters; (2) establish the postcondition with

```
op Push (Item x)
begin
1:    Node p;
2:    p := new Node;
3:    x :=: p.data;
      Early-Reply;
4:    p.next := self.top;
5:    self.top := p;
6:    self.len++;
end
```

```
op Pop (Item x)
begin
1:    x :=: self.top.data;
      Early-Reply;
2:    Node p := self.top;
3:    self.top := p.next;
4:    delete p;
5:    self.len--;
end
```

Figure 6: Early-Reply implementations of Push and Pop

respect to the distinguished parameter self; and (3) reestablish the representation invariant for the Stack instance. Provided that additional client calls are deferred until after each method body completes, the residual computations cannot affect client-side reasoning about program correctness with respect to postconditional semantics.

3 A Silhouette of Early-Reply

The implementation of Stack above is essentially straight-line code. As such, it is a simple illustration of the distinction between material and residual computations in software components. In general, however, this distinction is more complex, depending on the possibility of hierarchically composed Early-Reply components. An abstract characterization of material and residual computations is beyond the scope of this paper, as are the formal proof obligations for using Early-Reply correctly. Fortunately, such an in-depth analysis is not essential to understanding the basic insights developed herein. Accordingly, we digress for a moment to sketch a silhouette of the pertinent aspects of Early-Reply, postponing the formal details for a separate treatment in future work.

The operational semantics of Early-Reply can be characterized informally as follows: executing an Early-Reply statement causes a *logical fork* in the flow of control, with one branch completing the client's invocation by returning the current values of the formal parameters (excluding the distinguished parameter self), and the other fork continuing with subsequent computation of the method body until completion. After executing an Early-Reply statement, the component instance self defers execution of additional method invocations until *after* the current method has completed its execution *in toto*. If a client invokes another method on the component instance during this period, it will block until the previous method invocation has terminated, thereby "unlocking" self to proceed with the next method.

For a given method invocation, the *material computation* is defined as the sequence of statements executed up to, and including, the first Early-Reply statement, if any. The *residual computation* is defined as the sequence of statements executed subsequent to the first Early-Reply statement, up to the completion of the method body. This operational characterization of material and residual computations depends, in part, on the particular implementation of the method. As a boundary case, a method implementation that does not use, or execute, an Early-Reply statement has an empty residual computation. Furthermore, different traces of the same implementation on different parameter values may, or may not, execute an Early-Reply statement due to looping or branching control structures. Thus, the

material and residual computations depend not only on a method's implementation, but also on the parameter values passed on each invocation. Finally, Early-Reply is idempotent; that is, any Early-Reply statement encountered during a method's residual computation is executed as a no-op.

We now turn to the question of when is it safe to Early-Reply? That is, when can an implementation forward the results of a method invocation back to the client without compromising the client's *view* that the method has completed? Ordinarily, at the beginning of a method's body, the implementation can assume that the representation invariant holds, and that the values of the formal parameters satisfy the method's precondition. During execution, the representation invariant may be violated, but upon termination, the implementation must dispatch proof obligations that (1) the postcondition holds, and (2) the representation invariant has been (re)established. To ensure design-by-contract [8] – which is based on the pre- and postconditions of methods – an obvious proof obligation for using Early-Reply correctly is that the formal parameter values communicated back to the client must satisfy the method's postcondition.

For simplicity, this case study adopts a conservative interpretation of this requirement; namely, an implementation can Early-Reply safely only if *all* of its formal parameters satisfy the method's postcondition, with the possible exception of the distinguished parameter *self*. As an example, recall the Early-Reply implementation of Push in Figure 6. After swapping the formal parameter x with p.data in the third statement, x satisfies what the postcondition requires of it; namely, that Initial(x) is true. At this point, it is safe to Early-Reply, despite the fact that the distinguished parameter self has not yet satisfied its role in the postcondition. In particular, the value of self.len is incorrect, and the representation invariant has been violated, since the new top Node has not yet been linked into the list. The operational semantics of Early-Reply ensure that these aspects cannot be observed by the client until the component instance self becomes "unlocked" at the end of the method body. Thus, upon executing an Early-Reply statement, the implementation must dispatch the same proof obligations stated above; namely, that upon termination the postcondition holds for all formal parameters, *including self*, and that the representation invariant is true.

We add to these proof obligations two more conditions. The first is that the postcondition is never violated during the residual computation of a method. The second is that the values of the formal parameters cannot be changed during the residual computation. The first condition essentially rules out dangerous uses of aliasing, which are generally bad practice anyway. The second condition is to handle the possibility of relational postconditions which do not uniquely determine the final values of the formal parameters. In such cases, a residual computation could change the value of a formal parameter without violating the postcondition. Obviously, Early-Reply should preclude such aberrant behaviors.

4 Component-level Parallelism by Pipelining

Next, we present a layered implementation of an indexable Sequence using two Stack components. This implementation of Sequence is parameterized by an arbitrary implementation of Stack. Consequently, the correctness of the Sequence code can be reasoned about independently of which particular Stack implementation is selected at component integration time. When an Early-Reply implementation is selected, however, the performance of Sequence is transparently enhanced by virtue of the component-level parallelism of its Stack member variables.

Consider a Sequence component parameterized by the entry type Item, and supporting the operations Add, Remove and Length. We model the abstract value of a Sequence by a mathematical string of Item.

The initial value for variables of type Sequence is the empty string, denoted by <>. The operation Add(pos, x) inserts the Item x at the zero-based position indexed by pos, where index zero is viewed as the left-most position in the Sequence, by convention. The Length() operation returns the length of the Sequence, denoting the number of Items it contains. The operation Remove(pos, x) is similar to Add, except that it extracts into the parameter x the Item indexed by pos. The representation of Sequence uses two Stack instances as member variables, called before and after. Recall that type Stack is modeled by a string of Item, and that the left-most position in the string is viewed as the top. The abstraction function for this implementation is:

Abstraction Function: **AF** (self) = reverse (self.before) * self.after

To illustrate how self.before and self.after are used, consider a Sequence variable seq with the abstract value seq = <A, B, W, C, D, E, F>. Figure 7 shows a tracing table for the value of seq and its member variables seq.before and seq.after with respect to a call seq.Remove (2, ch).

seq	*ch*	*Call*	*seq.before*	*seq.after*
<A, B, W, C, D, E, F>	?		<D, C, W, B, A>	<E, F>
		seq.Remove (2, ch)	<C, W, B, A> <W, B, A> <B, A>	<D, E, F> <C, D, E, F> <W, C, D, E, F>
<A, B, C, D, E, F>	'W'		<B, A>	<C, D, E, F>

Figure 7: Tracing table for seq.Remove (2, ch)

The implementation of Sequence can directly access only the top Item on each of its representation Stacks. Thus, implementing Add and Remove as shown in Figure 8 involves traversing the Sequence to the zero-based index designated by the input parameter pos. This is achieved by calling a local operation Move_To_Index, which pushes and pops the before and after Stacks to shift Items left or right until the desired index is reached. At this point, Add and Remove complete by pushing or popping the input Item x onto or from the top of the after Stack, respectively.

```
op Add (int pos, Item x)
begin
        Move_To_Index (pos);
        self.after.Push (x);
end
```

```
op Remove (int pos, Item x)
begin
        Move_To_Index (pos);
        self.after.Pop (x);
end
```

```
local op Move_To_Index(int index)
begin
        var t: Item;
        while (self.before.Length () < index)
                self.after.Pop (t);
                self.before.Push (t);
        end while
        while (self.before.Length () > index)
                self.before.Pop (t);
                self.after.Push (t);
        end while
end
```

Figure 8: Implementations of Add and Remove for type Sequence

The performance of the Move_To_Index operation — and hence, of Add and Remove — depends, in part, on which implementation of Stack is selected at component-integration time. Figures 9 and 10 profile the calling sequence used by Move_To_Index, with respect to the blocking Stack and Early-Reply Stack implementations shown previously in Figures 5 and 6.

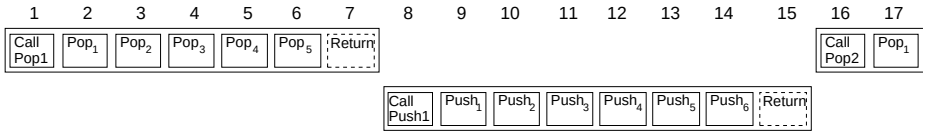

Figure 9: Sequence layered on the blocking implementation of Stack

When traversing the Sequence with a blocking implementation of Stack, the Move_To_Index operation must wait for each Pop invocation on one Stack to complete *in toto* before it can transfer the current Item to the other Stack with a corresponding Push invocation. Similarly, Move_To_Index must block for each Push invocation to complete before issuing a subsequent Pop. Since the flow of data is coupled to the flow of control, Move_To_Index is forced to block for the entire duration of each method. The blockage time experienced by Move_To_Index is illustrated in Figure 9.

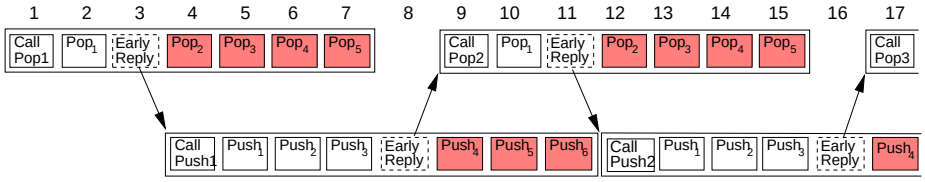

Figure 10: Sequence layered on an Early-Reply implementation of Stack

When an Early-Reply implementation of Stack is used, however, the residual computation of each Push or Pop on one Stack can be overlapped with the material computation of the subsequent Push or Pop on the *other* Stack. The component pipelining that results from this overlapped execution sequence is profiled in Figure 10. Each method execution is illustrated by a sequence of white and colored boxes, depicting material and residual computations, respectively.

The natural component pipelining enabled by the Early-Reply Stacks makes Move_To_Index — and hence, Add and Remove — more efficient when Stack operations can be executed in parallel.

Although our implementation of Sequence was not explicitly designed as an Early-Reply component, its performance benefits from Early-Reply implementations of the Stack instances it uses. Reasoning about the correctness of the Sequence client code can be done modularly, in the sense that it depends only on the pre- and postconditional semantics of the Stack type. Thus, we preserve sequential reasoning despite the potential for parallel execution of Stack implementations.

5 Performance Analysis of Early-Reply Components

So far, our case study has presented two component implementations using the Early-Reply construct — directly in the case of Stack, and indirectly in the case of Sequence (as a client of Stack). But how does Early-Reply affect the performance of these components in real systems? This section serves as a

synchronization point wherein we explore this question before moving on to hierarchically composed systems.

Analyzing the performance of Early-Reply components requires some special considerations analogous to those for evaluating the performance of pipelined architectures. The performance gains of instruction-level parallelism are measured relative to *ideal speedup*, which is bounded by the depth of the pipeline. The *actual speedup* must take into account the stalls arising from cache misses, data dependencies, and other pipeline hazards. Similarly, the ideal speedup of component-level parallelism arising from the use of Early-Reply components is bounded by the amount of residual computation that can potentially be overlapped in parallel execution with subsequent client computation. The actual speedup is more complex and depends on at least two factors.

The first factor is the availability of run-time resources for supporting *physical concurrency* in hardware — for example, with multiprocessor environments or with network-based computing systems such as LANs. Obviously, software composed of Early-Reply components will not deliver physical speedup when executed on a single, stand-alone processor. The isolated computing device, however, is rapidly becoming a thing of the past; even cell phones and PDAs are capable of participating in computing networks these days. When support is available, however, the logical concurrency encapsulated by Early-Reply components automatically scales to exploit physical concurrency in the run-time system. This performance scaling is transparent to the developer in the sense that the client application does not have to be re-engineered to exploit the speedup of physical concurrency.

A second factor determining actual speedup depends on how intensively an Early-Reply component is used. A client making two successive calls to an Early-Reply component may temporarily block on the second call if the residual computation of the first invocation has not yet completed. One may argue that intensive calling is not the common case for data-structure components, except perhaps during I/O and initialization. Currently it is a matter of further inquiry, but real software is often characterized by client code being interspersed with interleaving method invocations to subordinate components. Moreover, research in multi-invariant data structures [9] has explored techniques for a method implementation to abandon or early exit its residual computation in response to bursty periods of client calling.

We note that the hardware factors of networking and multiprocessor support for physical concurrency are independent of the software factors affecting actual speedup. An in-depth treatment of this topic is beyond the scope of this paper. Instead, we focus on the *response time* of idle components, which is an aspect of overall performance depending only on the complexity of a method's worst-case material computation.

A component is said to be *idle* if it has completed the residual computation of the client's most recent method invocation; that is, it is not currently executing any residual code that would defer the immediate execution of any forthcoming method invocation. Note that in a synchronous calling model, ordinary components satisfy this criterion almost trivially; since the client is forced to block for the entire duration of each method, the residual computation is always empty, and so the component immediately transits back to idle upon completing a method and returning control to the client caller.

Figure 11 compares the worst-case response time of idle components, with respect to the blocking, Early-Reply, and layered method implementations presented in this case study so far. In these examples, the Early-Reply and layered implementations do not achieve any order-of-magnitude improvements in response time when compared to their blocking cousins. The improvement in response time for these cases is real, but is only by a constant factor, and so, the benefits of Early-Reply are hidden by the order-complexity analysis of these methods. This is not a limiting factor of Early-Reply itself, however, but rather of the simplicity of the examples considered so far. Next, we extend our case study to more

Operation	Blocking	Early-Reply	Layered
Stack Push	$O(1)$	$O(1)$	—
Stack Pop	$O(1)$	$O(1)$	—
Sequence Add	$O(n)$	—	$O(n)$
Sequence Remove	$O(n)$	—	$O(n)$

Figure 11: Worst-case response time for idle components

complex examples where Early-Reply can yield order-of-magnitude improvements in the response time of idle components.

6 Optimizing the Response Time of Idle Components

The next example of our case study presents a Set component parameterized by Item, the type of entry in the Set. We model the abstract value of the component type Set by a mathematical set of Item. Please refer to Figure 12 for formal specifications of the operations exported by the type Set. Informally, Add (x) inserts the Item x into the Set; Remove (x, x_copy) removes a specific Item from the Set with the same value as the formal parameter x, and returns it in x_copy; Remove_Any (x) extracts an unspecified (i.e., arbitrary) member from the Set; Is_Member (x) queries the membership of Item x in the Set; and Size() returns the cardinality of the Set.

Set is modeled by finite set of Item
initialization ensures: self = {}

op Add (Item x)
precondition: x ∉ self
postcondition: Initial(x) **and**
self = #self ∪ {#x}

op Remove (Item x, Item x_copy)
precondition: x ∈ self
postcondition: x_copy = x **and**
self = #self − {x}

op Remove_Any (Item x)
precondition: self ≠ {}
postcondition: x ∈ #self **and**
self = #self − {x}

op Is_Member (Item x): boolean
precondition: true
postcondition: Is_Member = (x ∈ self)

op Size (): int
precondition: true
postcondition: Size = |self|

Figure 12: Specification of type Set and its methods

This implementation of Set uses a standard binary search tree (BST) for its representation, the specifics of which can be found in most algorithms texts [5]. The sole member variable of the Set representation is self.tree, which denotes a binary tree containing the elements of the Set. The representation invariant and the abstraction function for this component are presented in Figure 13.

The representation invariant places two constraints on self.tree. The predicate Is_BST expresses that the binary tree denoted by self.tree is actually a binary *search* tree; namely, if x is any root node in a subtree of self.tree, then every node y in the left subtree of x satisfies $y \leq x$, and every node z in the right subtree of x satisfies $x < z$. This property is required for the correctness of local tree operations.

Representation Invariant:	Is_BST (self.tree) **and** Items_Are_Unique (self.tree)
Abstraction Function:	**AF** (self) = Elements (self.tree)

Figure 13: Representation Invariant and Abstraction Function for Set

The predicate Items_Are_Unique requires that self.tree contains no duplicate node values. This property is required to support the uniqueness of elements in a mathematical set, which is the model of our type Set. Given the strength of the representation invariant, the abstraction function **AF** is almost trivial: the abstract value of a Set is simply the set of all node values contained in the binary tree denoted by self.tree.

We now present an Early-Reply implementation of Set using the representation described above. The method bodies in Figure 14 make use of local operations on binary search trees. The behavior of each local operation is described informally below. To ensure efficient implementations of the Set methods, each of the local tree operations Insert, Delete, and Delete_Root are assumed to rebalance the tree after altering it. Various mechanisms for maintaining balanced binary trees guarantee a worst-case time complexity of $O(log\ n)$ for each tree operation below, where n is the number of nodes in the tree [5].

Find (bst, x): Node

Searches bst, returning the node with value x, or null if no such node exists.

Insert (bst, x)

Traverses bst and inserts a node with value x at the appropriate location.

Delete (bst, n)

Deletes node n and restores bst to a binary search tree.

Delete_Root (bst)

Deletes the root node of bst and restores bst to a binary search tree.

The Add operation simply creates a local variable y of type Item, and swaps it with the formal parameter x. At this point, the postcondition is satisfied with respect to x — that is, Initial (x) holds — and so Add can Early-Reply, thus completing its material computation. The residual computation of Add performs the actual insertion of the original value of x (which is now in the local variable y). By contrast, a blocking implementation would physically insert x before returning control to the client. Deferring this task to the residual computation enables an Early-Reply implementation to reduce the worst-case response time of an Add invocation on an idle Set from $O(log\ n)$ to $O(1)$, since both the declaration of y and the swap with x can be done in constant time [6].

A blocking implementation of Remove would proceed roughly as follows before returning control to the client: find the node with value x (by searching the tree), swap its value into x_copy, delete the old node, and rebalance the tree. In the Early-Reply implementation of Remove, the final two steps are offloaded to the residual computation. In particular, the client does not have to block while the tree is

being rebalanced, which can take up to $O(log\ n)$ time. The response time of both implementations is still $O(log\ n)$, but the constant factor for the Early-Reply implementation is smaller.

When a client does not know the contents of a Set, it may not be able to Remove elements by explicitly naming their values. Thus, we include the Remove_Any operation for completeness. This method removes an arbitrary element from the Set. Thus, a valid implementation of this requirement is to simply remove the root node. For all but trivially small trees, this requires replacing the root node in order to re-establish the representation invariant that self.tree is a binary search tree. Blocking implementations of Remove_Any must establish the invariant before returning control to the client. With Early-Reply, however, an implementation can simply swap self.tree.root with the formal parameter x, and Early-Reply to the client, thereby offloading tree restoration to the residual computation. This technique reduces the worst-case response time of invocations to Remove_Any on an idle Set component from $O(log\ n)$ to $O(1)$.

Figure 15 compares the worst-case response times of method invocations on idle Set components. This step in our case study illustrates how Early-Reply can be used to achieve order-of-magnitude improvements in response time, when compared to ordinary blocking implementations. Recall that a blocking component transits to idle immediately after returning control to the client. An Early-Reply component, however, reduces response time by offloading work to the residual computation of each method invocation, during which it defers handling additional client calls. Thus, an Early-Reply component is *not* idle during its residual computation.

Early-Reply improvements in response time are a big win if a component is frequently idle, but long residual computations can negate this potential performance enhancement. Consider the Early-Reply implementation of Set presented in Figure 14. If a client invokes an Add(x) method, the component will Early-Reply to the client in constant time, and postpone to its residual computation the $O(log\ n)$ amount of work required to physically insert the Item x into self.tree. This is great for the client, so long as the next method invocation to the Set does not occur during this residual computation. If the

```
op Add (Item x)
begin
        var Item y;
        x :=: y;
        Early-Reply;
        Insert (self.tree, y);
end

op Remove (Item x, Item x_copy)
begin
        var Node y;
        y := Find (self.tree, x);
        x_copy :=: y.data;
        Early-Reply;
        Delete (self.tree, y);
end
```

```
op Remove_Any (Item x)
begin
        x :=: self.tree.root;
        Early-Reply;
        Delete_Root ();
end

op Is_Member (Item x): boolean
begin
        return (Find (self.tree, x) ≠ null)
end

op Size (): int
begin
        return self.tree.size;
end
```

Figure 14: An Early-Reply implementation of Set

Operation	Blocking	Early-Reply
Set Add	$O(log\ n)$	$O(1)$
Set Remove	$O(log\ n)$	$O(log\ n)$
Set Remove_Any	$O(log\ n)$	$O(1)$

Figure 15: Worst-case response time for idle Set components

client does make such a call, the Set component will defer servicing the request until after the residual computation of the initial Add method has completed. Thus, the response time experienced by the client may degrade during bursty periods of method invocations.

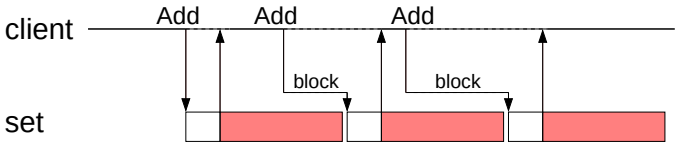

Figure 16: Early-Reply Set with long residual computations

Such a calling pattern is illustrated in Figure 16. When two consecutive calls are made to Add, the second invocation blocks for the remainder of the $O(log\ n)$ residual computation of the previous invocation. This degrades the response time of the second call to $O(log\ n)$. Thus, a key to leveraging the full potential of Early-Reply is to *minimize* the length of residual computations. Surprisingly, this can be accomplished by hierarchically composing Early-Reply components. We explore a general technique for realizing this goal in the next section.

7 Minimizing Residual Computations

The key insight of this section is that the residual computation of a method can be reduced by offloading work to a subordinate Early-Reply component. Surprisingly, this goal can be accomplished without introducing novel stratagems. In particular, this section illustrates how to apply the well-known implementation technique of *hashing* to reduce the duration of residual computations.

Typical applications of hashing employ chaining as a technique for conflict resolution. For example, a standard hashing implementation of the Set component presented in Section 6 would use a hash table represented as an array of singly-linked lists. Adding an element to the Set involves hashing to the appropriate bucket, and then linking a new node into the list of nodes contained therein. Removing an element also involves hashing to the appropriate bucket, and then chaining along the linked list to find and extract the desired node.

Although this implementation approach is commonplace, we observe that the use of linked lists is unwarranted; closer inspection reveals that they essentially function *as sets*, insofar as they are only used to store and retrieve elements. Accordingly, we present a parameterized realization of Set that is implemented using an array of other Set instances. For ease of reference, let us call this implementation Hash-Set to indicate that it is a hashing implementation of the Set component presented in Section 6.

The implementation Hash-Set has two parameters: Item (which is the type of entry in Set), and Base-Set (which is an arbitrary implementation of type Set). The representation of this implementation

contains a single member variable, denoted by self.table, which is of type Array of Set_Base. Note that the hash table self.table represents the abstract Set as a string of instances of type Base-Set. This is a key feature to reducing the residual computations of the higher-level Set methods. Finally, the representation invariant and abstraction function for Hash-Set are presented in Figure 17, where self.table.i refers to the Base-Set instance in bucket i of the hash table. The representation invariant says that the Base-Set instances in each bucket of the hash table are pairwise disjoint, and the abstraction function says that the abstract model of the Set is the union over all Base-Set instances in the hash table.

Representation Invariant: $\forall i, j : \mathbb{N}$ where ($0 \leq i <$ | self.table | **and**
$0 \leq j <$ | self.table | **and**
$i \neq j$):
self.table.i $\bigcap$ self.table.j $= \{\}$

Abstraction Function: **AF** (self) $= \bigcup_{i=0}^{|\text{self.table}|-1}$ (self.table.i)

Figure 17: Representation Invariant and Abstraction Function for Hash-Set

In Section 6, we presented an Early-Reply implementation of the abstract component Set using balanced binary search trees. For ease of reference, let us refer to that implementation as the BST-Set, to distinguish it from the Hash-Set implementation presented here. Hash-Set is also an Early-Reply implementation of the abstract component Set, except that it uses a hash table of subordinate Base-Set instances, instead of a binary search tree.

```
op Add (Item x)
begin
        var Item y;
        y :=: x;
        Early-Reply;
        var int i := Hash(y);
        self.table.i.Add(y);
end
```

Figure 18: Implementation of Add in Hash-Set

Consider the implementation of Add in Figure 18. The material computation of Add simply creates a local variable y of type Item, swaps it with the formal parameter x, and then returns control to the client using Early-Reply. This can be accomplished in constant time. The local operation Hash in the residual computation applies the hash function to its argument, and returns an integer-valued index into self.table corresponding to the appropriate bucket. We note that good hashing functions are independent of the number of elements in the Set, and can often be computed in constant time. Thus, the duration of the residual computation of an Add invocation depends primarily on the duration of the lower-level invocation to Add on the instance of Base-Set in the appropriate bucket.

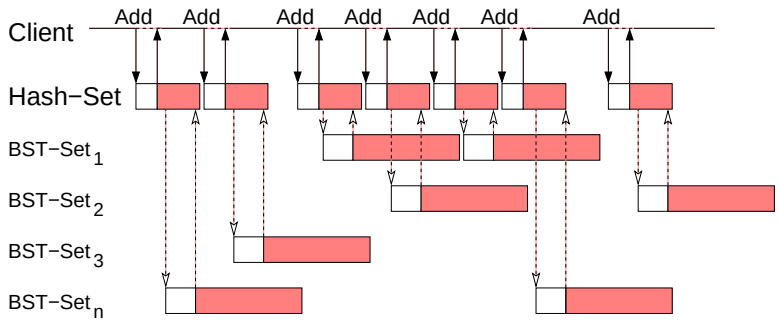

Figure 19: Execution of Hashing implementation of Set

At this point, it should be clear that we can reduce the *residual computation* of Add on the higher-level component by reducing the *response time* of Add on the lower-level Base-Set component. Recall that Hash-Set is parameterized by Base-Set, which is an arbitrary implementation of Set selected at component-integration time. Now suppose we select the Early-Reply component BST-Set as our implementation, and use it to instantiate Hash-Set to get a new implementation ER-Set = Hash-Set<BST-Set>. The material and residual computations of Add invocations to an idle ER-Set component will both be constant in duration. Moreover, the ER-Set distributes the work of client invocations among several instances of BST-Set in the hash table.

Figure 19 illustrates a possible trace of intensive client-calling on an ER-Set component. Given the nature of hashing, the odds of having consecutive Add operations hash to the same bucket are statistically low. Thus, an idle ER-Set can Early-Reply in constant time, and — with high probability — complete its residual computation in constant time as well. By layering Early-Reply components using hashing, we have optimized the common case that enables an ER-Set to minimize its residual computation, thereby decreasing the likelihood that additional client invocations will be blocked.

8 A Lightweight Approach to Reuse

In the previous section, we have shown how the response time of an Early-Reply component can be improved (by an order of magnitude) by layering it on other Early-Reply components. In the absence of such components, however, what can one do? We answer this question in this section by presenting a lightweight, non-invasive, and general technique for transforming off-the-shelf, blocking components into Early-Reply components.

We continue our treatment of the Set component in this section. Consider any arbitrary off-the-shelf implementation Block-Set of Set. We can take this component, and instantiate the Early-Reply Hash-Set template presented in the previous section. Observe that the component resulting from such an instantiation Lightweight-Set = Hash-Set<Block-Set> is an Early-Reply implementation of Set. For instance, when a client calls the Add operation on Lightweight-Set, the operation can just swap the formal parameter with a local variable, and Early-Reply to the client. After that, during its residual computation, the component could hash to the correct bucket and then call Add on the Block-Set corresponding to that particular bucket. Thus, the response time of the Add operation has been reduced from $O(log\ n)$ to constant time.

The foregoing discussion is a lightweight approach to transform off-the-shelf sequential components

into Early-Reply implementations. Such a transformation, however, may not apply to all methods in the component. Any method that either *consumes* or *preserves* its formal parameters can be recast as an Early-Reply implementation using this approach. Since the layered implementation uses the underlying implementation to service its clients, a blocking implementation of a method that *produces* a result cannot be transformed into an Early-Reply implementation.

Note, however, that even though the response time of the Add operation is $O(1)$ in Lightweight-Set, it has the same problem that we identified with the Early-Reply implementation of Set with the binary search tree representation presented in Section 6, namely, that the response time is $O(1)$ only under the condition that the client did not call operations on the Lightweight-Set component.

Since the problem with the two components is the same, we can reuse the solution from Section 7 to rectify the problem with Lightweight-Set, which is an Early-Reply implementation, at least with respect to Add. We use Lightweight-Set as the Base-Set parameter to the template Hash-Set to create a new implementation. This implementation Easy-Set = Hash-Set<Lightweight-set> is an Early-Reply component that has the same response time as the ER-Set component presented in the previous section. Now, wasn't that easy? ☺

9 A Basis for Pipebranching Parallelism

In section 4, we illustrated a single thread of component-level parallelism between the Stack components used in a layered implementation of Sequence. Later, in section 7, we extended this to exploit the potential for multiple threads of component-level parallelism among replicated instances of a Base-Set component used in a hashing implementation of Set. These examples both shared a common feature; namely, that a single method invocation spawned a single thread of parallel execution. As such, "component pipelining" is an adequate description. In this section, however, we sketch a profile of the landscape beyond component pipelining, in which a single method invocation may spawn a superlinear number of parallel threads of execution. We call this phenomenon *pipebranching parallelism*.

TMR (triple modular redundancy) is a standard technique for implementing persistent data in a way that can mask single-site catastrophic failures or data corruptions. The idea is to copy the data value in triplicate, and then to store each physical copy in a geographically separate facility. When a client attempts to Read a persistent data value, the three copies are fetched from their respective locations, and they "vote" on the current value. In the case of a single failure, two of the three values will still "agree", and so the data corruption can be masked and corrected. Similarly, when a client attempts to Write a persistent data value, the three copies must agree to "commit" the update by having successfully completed the write-back transaction at their respective site.

Now consider a Persistent-Set component. This component could be realized as a layered implementation using three ER-Set components as its representation, each residing at a different physical site. When the client invokes an Add(x) method, the implementation copies the formal parameter x in triplicate, and then delegates the method call to each of its three subordinate ER-Set components. In order to commit the transaction, each ER-Set does not have to physically update its representation. Rather, it is sufficient to record the method call and its parameter values in an auxiliary log file, and then Early-Reply to the coordinating Persistent-Set parent component. From the client's perspective, the transaction has completed, despite the fact that the residual computation of each ER-Set may not yet have completed the physical update at its respective site. In the case of a site failure, each ER-Set can reconstruct its state using the transaction log file, provided that it is persistent too. This may involve additional calls to other subordinate components encapsulated by the implementation of the log file.

This example illustrates how a single method invocation on the part of a client may cascade into multiple invocations on independent lower-level subcomponents during the residual computation of the method's implementation. This potential for pipebranching parallelism is enabled by Early-Reply. The same phenomenon can be achieved with multi-threaded systems using parallel algorithms, except for the following critical difference. A component implemented using Early-Reply is *written* as a sequential program, and it is *reasoned about* as a sequential program. This differs greatly from multi-threading, wherein explicit awareness of concurrency must be taken into consideration. The value of encapsulating the concurrent aspects of an implementation behind a sequential view of a program — as Early-Reply does — cannot be underestimated.

10 Related Work

There has been a large amount of work done in the field of architectural pipelining with respect to achieving speedup in computer processors [13]. Stretch – the first general-purpose pipelined machine was built over forty years ago [1, 2]. Since then, several improvements have been proposed. In fact, Reduced Instruction-Set Machines (RISC) machines were originally designed with pipelining in mind. This idea, however, has stayed within the context of hardware, and has not been used very much in the realm of computer software.

Parallel algorithms have been a force among software designers. Several approaches to parallel programming have been proposed [3], and a variety of programming languages have been developed specifically for the purpose of writing parallel programs [12, 14]. Various models for constructing and reasoning about such systems exist [4, 7], but the reasoning exercise involved is a lot greater than the simpler model of sequential programming.

Another piece of work that is related to our research is the "Pipeline model" of thread organization [11], wherein a single process spawns off multiple threads of execution which forward data to one another in a similar fashion as with hardware pipelining. However, reasoning about such systems requires the programmer to explicitly reason about inter-process communication or multi-process data structures to act as shared buffers for passing data from producer threads to consumer threads.

11 Conclusions

The focus in traditional approaches to data structures has been just that: clever techniques for structuring data. Pipebranching components expands that focus to encompass novel techniques for structuring the *flow* of data. We view pipebranching components as spearheading a new avenue of research in software engineering. A basis for this direction is the Early-Reply construct presented in this paper. Early-Reply enables application developers to write and reason about *sequential* programs, despite the potential for physically concurrent execution at run-time. Moreover, systems composed of Early-Reply components automatically exploit the support for physical concurrency in the run-time system, thereby enabling non-invasive performance scalability. A fraction of these performance enhancements can be realized simply by substituting lower-level Early-Reply components into existing systems. Alternatively, we have presented a lightweight approach for converting ordinary off-the-shelf components into components supporting some Early-Reply methods whenever hashing is an applicable implementation strategy. Finally, we showed how systems hierarchically composed of Early-Reply components may achieve pipebranching parallelism as the result of individual method invocations.

We began this paper with an analogy to hardware pipelining, and introduced Early-Reply as a software equivalent to result forwarding. But at that point the analogy with hardware ends, and the complexities with software begin. Early-Reply presents many possibilities for future work including:

- Formal semantics and proof rules for Early-Reply.
- An in-depth characterization of material and residual computations.
- Performance analysis of actual speedup using Early-Reply components.
- Applications of Early-Reply to problems in distributed computing.
- Using Early-Reply to encapsulate instrinsically concurrent problems.
- Generalizing Early-Reply in the context of client-caller synchronization.

References

[1] E. Bloch. The engineering design of the STRETCH computer. In *Proc. Fall Joint Computer Conference*, pages 48–59, 1959.

[2] W. Bucholtz. *Planning a computer system : Project Stretch.* McGraw Hill, 1962.

[3] K. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, 1992.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, USA, 1988.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1990.

[6] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Engineering*, 17(5):424–435, May 1991.

[7] C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.

[8] B. Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance and Classification)*, 25(10):40–52, 1992.

[9] A. Moitra, S. Iyengar, F. Bastani, and I. Yen. Multilevel data structures: models and performance. *IEEE Trans. Soft. Eng.*, 18(6):858–867, June 1988.

[10] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. on Computers*, C-33(11):968–976, November 1984.

[11] S. J. Norton and M. D. DiPasquale. *Thread time: the multithreaded programming guide.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[12] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1997.

[13] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Computing Surveys*, 9(1):61–102, 1977.

[14] P. A. G. Sivilotti and P. A. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Department of Computer Science, California Institute of Technology, 1994.