Early-Reply Components: Concurrent Execution with Sequential Reasoning

Scott M. Pike and Nigamanth Sridhar

Computer and Information Science, Ohio State University,
2015 Neil Ave, Columbus OH 43210, USA
{pike,nsridhar}@cis.ohio-state.edu

Abstract. Generic software components have a reputation for being inefficient. Parallel implementations may improve performance, but can thwart reuse by being architecture-dependent or by exposing concurrency to client-side reasoning about component interactions. To address performance, we present Early-Reply as an alternative to blocking method invocations. Component operations can be partitioned into a *material computation* required to satisfy the postcondition, and a *residual computation* required to reëstablish the component invariant, optimize its representation, etc. Early-Reply exploits opportunities for parallelism by forwarding final parameter values to the caller as soon as the material computation completes, thereby offloading the residual computation to execute in parallel with subsequent client activities. Proof obligations for Early-Reply support a synchronous calling model, so clients can still reason sequentially about component behavior. Also, since Early-Reply components do not depend on system-wide support for component synchronization, they can be deployed incrementally. Finally, Early-Reply can improve the response time of idle components by orders of magnitude; when composed hierarchically, performance benefits are magnified by the potential fan-out of concurrently executing components.

1 Introduction

Mature engineering disciplines are witness to the fact that component-based development is fundamental to the construction of scalable systems. Despite the benefits of componentry, however, modularization techniques for software suffer from a perceived view of being inefficient. This reputation derives from two primary factors. First, when encapsulation and information hiding are observed, executing component code requires invocations which introduce synchronization costs and calling overhead. Second, reusable components should be applicable to a wide variety of contexts, a desideratum that precludes many domain-specific and architecture-dependent optimization techniques [8]. Calling overhead and context independence are certainly performance factors, but are generic components inherently inefficient, or is their reputation for inefficiency an artifact of current practice? We examine this question under the following two-fold thesis: synchronous software components can be implemented to execute concurrently

C. Gacek (Ed.): ICSR-7, LNCS 2319, pp. 46–61, 2002.
© Springer-Verlag Berlin Heidelberg 2002

(1) without compromising sequential reasoning about component behavior, and (2) without presupposing system-wide support for runtime synchronization.

Components typically adhere to a synchronous calling model, where a client *blocks* for the duration of an invocation, until the method body completes by returning control and results (if any). Strangely, the common-case implementation of a synchronous calling model is with synchronous calling conventions. This practice is somewhat daft, given that a calling model is just that: a *model*. As such, this abstraction can be implemented in many ways. We present Early-Reply as a construct that supports concurrent component execution in a synchronous calling model. We develop Early-Reply to exploit the performance benefits of concurrency, *without* compromising the reasoning benefits of synchronous calls.

Component methods can be partitioned into *material* and *residual* segments, corresponding to the initial computation required to satisfy the postcondition, and subsequent computation required to complete the operation, respectively. In a naïve implementation of a synchronous calling model, the caller blocks during the residual computation, even though the results of the invocation are already available. This is analogous to read-after-write data hazards in pipelined architectures, where future instructions may stall due to dependencies on earlier instructions in the pipeline. A well-known technique for minimizing pipeline stalls is to forward results to future instructions as soon as they become available. This increases both instruction-level parallelism and overall resource utilization.

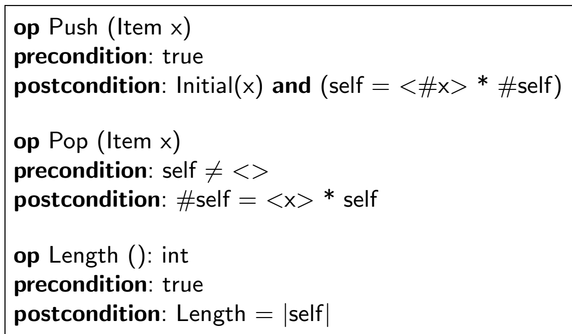
In this paper, we extend the idea of result forwarding to software components. In Section 2, we introduce Early-Reply as an abstract construct to decouple data flow from control flow in sequential programs. Further characterization of Early-Reply is presented in Section 3. In Section 4, we use a Set component as an example to show how the response time of idle components can be optimized. Section 5 analyzes the worst-case response time of Early-Reply components. An approach to further minimizing the amount of client blockage time is described in Section 6. We show how Early-Reply can be used as a lightweight approach to transforming off-the-shelf components into Early-Reply implementations in Section 7. Some directions of related research are listed in Section 8. We summarize our contributions, outline areas of future work and conclude in Section 8.

2 Decoupling Data Flow from Control Flow

The common-case implementation of a synchronous calling model couples the flow of data to the flow of control. As such, component implementations cannot return parameter values to the caller without relinquishing control to execute further statements in the method body. Entangling data with control precludes many implementations that can improve performance by exploiting parallelism. To separate these concerns, we present Early-Reply as an abstract construct for decoupling data flow from control flow.

As a motivating 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 any variable x is created with an initial value for its 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. Figure 1 gives specifications of basic stack operations, where #x in a postcondition denotes the incoming value of x, and x denotes its outgoing value. By convention, we view the left-most position of a string as the top of the Stack it models. Also, we use the distinguished parameter self to denote the component instance through which methods are 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|

Fig. 1. Specification of basic Stack operations

Figure 2 shows a common linked-list representation of Stack. Each Node n contains two fields: n.data of type Item, and n.next which points to the next node in the linked list. Each Stack instance has two data members: self.top (which points to the first Node in the linked list), and self.len (which denotes the current length of the Stack). Figure 3 shows the representation invariant and abstraction function. The former defines the state space of valid representations. The later defines how to convert such representations into their abstract Stack values.

Figure 4 shows common implementations 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

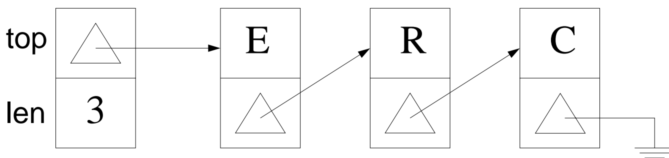


Fig. 2. Representation of a Stack<char> with abstract value < E, R, C >

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, and where the next field of the final node is 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.

Fig. 3. Representation Invariant and Abstraction Function for Stack

```
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
```

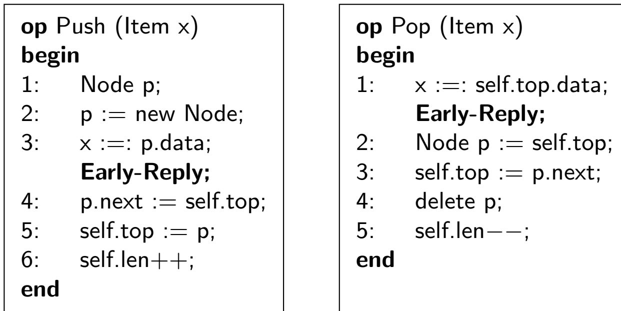
Fig. 4. Blocking implementations of Push and Pop

at the termination of each method. These are *blocking* implementations, in the sense that caller activities are suspended for the duration of the method body. The Length operation, which we omit for brevity, simply returns the current value of the member variable self.len.

Consider the implementation of Push. The first two statements declare and allocate a Node variable p. Recall that the Item field p.data is created with an initial value so that Initial(p.data) holds. The third statement swaps the formal parameter x with p.data, after which the postcondition for Push is satisfied with respect to x, since Initial(x) now holds. At this point, we claim, it is unnecessary for the client to continue blocking in the following sense: there is no *client-observable* difference between returning the value of x now – rather than when the method terminates – so long as the Stack instance defers additional method invocations until the remainder of the Push operation completes. Similar remarks apply to the implementation of Pop, but even earlier: the postcondition of Pop is satisfied with respect to the formal parameter x after the first statement, whereupon x = #self.top. Again, we claim, the client should not have 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 between subsequent client activities and the residual computations of Push and Pop. To

exploit this independence, we must decouple the link between data flow to the client and control flow in the method body. Accordingly, we present Early-Reply as an abstract construct for returning final parameter values to the caller without relinquishing local control to complete the method body. Figure 5 shows implementations of Push and Pop that use Early-Reply to reduce response time and to increase the potential for overlapping client and method computations.



```
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
```

Fig. 5. Early-Reply implementations of Push and Pop

In Figure 5, statements preceding each Early-Reply represent *material* computations, insofar as they establish the postcondition for 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 respect to the distinguished parameter self; and (3) reëstablish the representation invariant for the Stack instance.

3 A Silhouette of Early-Reply

The Stack example above is essentially straight-line code. As such, it motivates the distinction between material and residual computations in software components. A formal characterization of this distinction is beyond the scope of this paper, but an in-depth analysis is not essential to understanding the fundamental insights developed herein. This section sketches a silhouette of Early-Reply, postponing advanced formalisms for special treatment in future work.

The operational semantics of Early-Reply are characterized informally as follows: executing an Early-Reply statement causes a *logical fork* in the flow of control, with one branch returning the current values of the formal parameters to the caller (excluding the distinguished parameter self), and the other branch completing execution of the residual method body. Early-Reply "locks" the component instance self for the residual computation. An invocation on self during this period blocks the caller until the current invocation completes, whereupon self becomes "unlocked" and handles the next method. Local locking enforces mutual exclusion, however, so the component represented by self can be *passed as a parameter* to another component without blocking the caller.

For each invocation, we define the *material computation* as the sequence of statements executed up to, and including, the first executed Early-Reply, if any. We define the *residual computation* as the sequence of statements executed after the first Early-Reply up to method termination. Obviously, material and residual computations depend on the implementation. As a boundary case, method bodies that do not use, or execute, an Early-Reply statement have empty residual computations. Also, due to branching control structures, the same implementation may, or may not, execute an Early-Reply statement depending on the formal parameter values. Finally, Early-Reply is idempotent; that is, any Early-Reply statement encountered during a residual computation is executed as a no-op.

But when is it safe to Early-Reply? That is, when can a method return parameter values to the caller without compromising the client-side *view* that the method has completed? To ensure design-by-contract [6] in a synchronous calling model, we must encapsulate residual computations from client observability. At a minimum, this requires the formal parameter values to satisfy the method's postcondition when replied to the caller. An eager interpretation of this requirement could reply each parameter as soon as its final value was obtained, but this approach complicates reasoning about component correctness, and typically requires system-wide support such as data-driven synchronization [2]. We adopt a more conservative interpretation; namely, a method can Early-Reply only once *all* of its formal parameters satisfy the method's postcondition. We amend this policy to exclude the distinguished parameter self, which can use a local locking mechanism to synchronize itself without system-wide support.

As an example, recall the Early-Reply implementation of Push in Figure 5. Upon invocation, the component instance self becomes locked. After swapping the formal parameter x with p.data in the third statement, x satisfies the postconditional requirement that Initial(x) holds. At this point, it is safe to Early-Reply, despite the fact that the distinguished parameter self has not yet satisfied its part in the postcondition. In particular, the value of self.len is wrong, and the representation invariant is broken with respect to linking in the new top Node of the list. Early-Reply ensures that these aspects are not observable by the client because self is locked during the residual computation of Push. Upon termination, however, we must dispatch proof obligations that the representation invariant is true, and that the postcondition holds for every parameter, *including self*.

To preserve sequential client-side reasoning about component behavior, we add the following proof obligation; namely, residual computations cannot change the values of parameters that were already replied to the caller. This protects the client in two ways. First, it prevents a method from violating its postcondition *ex post facto*. Second, it prevents potential inconsistencies arising from relational specifications, where a method could maintain its postcondition even while surreptitiously changing its replied parameter values.

A final proof obligation to protect implementations from subsequent client activities is that residual computations should not use *aliased* parameter values after an Early-Reply. After Early-Reply, the caller views the method as having completed. This fact is essential to supporting the abstraction of a synchronous

calling model, but component correctness may be compromised if aliased parameters are altered by the caller during a residual computation. Consider, for instance, the effect of changing a key value while it is being inserted into a binary search tree! To avoid aliasing, a method can deep-copy parameter values that are needed during its residual computation. This is potentially expensive, especially when data needs only to be *moved* rather than *copied*. For brevity and efficiency, we use swapping in this paper as the primitive data-movement operator, since it can be implemented to execute in constant time without creating aliases [4].

4 Reducing the Response Time of Idle Components

A component is said to be *idle* if it has completed its most recent invocation; that is, it is not currently executing any residual (or material) code that would defer the immediate execution of subsequent methods. Note that blocking components satisfy this criterion trivially: since the caller blocks for the entirety of each method, the residual computation is always empty, and so the component immediately transits back to idle. Early-Reply components can reduce response time by offloading the work of non-empty residual computations. For idle components, Early-Reply can deliver order-of-magnitude improvements in response time. We substantiate this claim in the context of search-and-query components such as symbol tables and databases. For simplicity, we present a parameterized Set component in Figure 6 as a representative of this class of components. We represent the Set using a standard binary search tree (BST) [3]. The sole member variable of the component instance self is self.tree, which denotes the BST.

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

op Add (Item x)
precondition: $x \notin$ self
postcondition: Initial(x) **and**
self $=$ #self $\cup \{$#x$\}$

op Remove (Item x, Item x_copy)
precondition: $x \in$ self
postcondition: x_copy $=$ x **and**
self $=$ #self $- \{x\}$

op Remove_Any (Item x)
precondition: self $\neq \{\}$
postcondition: $x \in$ #self **and**
self $=$ #self $- \{x\}$

op Is_Member (Item x): boolean
precondition: true
postcondition: Is_Member $=$ ($x \in$ self)

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

Fig. 6. Specification of type Set<Item> and its methods

The representation invariant and the abstraction function for this component are presented in Figure 7. 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; that is, if x is the root node of any subtree of

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

Fig. 7. Representation Invariant and Abstraction Function for Set

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. The predicate Items_Are_Unique requires that self.tree contain no duplicate key values. This property enforces the uniqueness of elements in a mathematical set, which is the model of the Set type. Given the strength of the representation invariant, the abstraction function is almost trivial: the abstract value of a Set is simply the set of all key 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 8 make use of local operations on the BST representation. We describe the behavior of each local operation informally below. To ensure efficient implementations of the Set methods, each of the local tree operations Insert, Delete, and Delete_Root is assumed to re-balance 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 [3].

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, which completes its material computation in constant time. 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. For an idle Set component, deferring this task to the residual computation enables an Early-Reply implementation to reduce the worst-case response time of an Add invocation from $O(log\ n)$ to $O(1)$.

A blocking implementation of Remove would proceed roughly as follows: find the node with value x (by searching the tree), swap its value into x_copy, delete the old node, re-balance the tree, and then return control to the client. In an Early-Reply implementation of Remove, the work of deleting and re-balancing can

```
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
```

Fig. 8. An Early-Reply implementation of Set

be offloaded to the residual computation. The response time of both implementations is $O(log\ n)$, but the constant factor for the Early-Reply implementation is a fraction of the blocking implementation.

The Remove_Any operation is included for completeness, so that a client can manipulate a Set without knowing the key values of its elements. Since Remove_Any extracts an arbitrary element from the Set, a valid implementation is simply to remove the root node. For all but trivially small trees, this requires replacing the root in order to reëstablish 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 and re-balancing to the residual computation. As with Add, the worst-case response time of Remove_Any on an idle component can be reduced by an order of magnitude. Figure 9 summarizes the foregoing results.

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

Fig. 9. Worst-case response time for idle Set components

5 Performance Analysis of Early-Reply Components

The *ideal* speedup of Early-Reply components is bounded by the amount of residual computation that can potentially execute in parallel with subsequent client computation. The *actual* speedup, however, depends on at least three factors, including synchronization overhead, hardware support for physical concurrency, and runtime component usage. We discuss these factors below, each in turn.

The first factor is the brute cost of synchronization. If an Early-Reply implementation replaces local invocations by remote procedure calls, this cost can be a significant bottleneck. Synchronization overhead can be masked, however, for applications that already require a distributed infrastructure such as CORBA, Java RMI, or COM+. For local environments, invocations can be replaced by lightweight threading constructs. Assuming a constant upper bound on synchronization costs, this overhead can be absorbed by Early-Reply components yielding order-of-magnitude performance improvements on suitably large data sets.

A second factor is run-time support for *physical* concurrency in hardware, as in multiprocessing or networked environments. When support is available, Early-Reply components automatically scale to exploit physical concurrency at run-time. This scaling is transparent, because client applications can reap the performance benefits without being re-engineered. Even for uniprocessor environments, Early-Reply is not without benefits, since multitasking increases resource utilization. For example, I/O-bound applications can increase resource utilization by overlapping residual computations with blocking I/O system calls.

A final factor is how intensively an Early-Reply component is used. During a bursty period of method calling, a client may block temporarily while the residual computation of a previous call completes. Overcoming this performance bottleneck involves reducing the lockout duration of residual computations. One approach is to use multi-invariant data structures to define *safe points* at which methods can *early exit* to service incoming calls [7]. For example, the re-balancing code for a BST can be abandoned at points where the representation invariant holds. In this paper, we propose a reuse-oriented alternative to new constructs like Early-Exit which minimizes residual computations by layering new Early-Reply components on an efficient substrate of existing Early-Reply components.

6 Minimizing Residual Computations

Early-Reply improvements in response time are a big win if a component is frequently idle, but long residual computations can negate the potential payoff. Recall the Early-Reply implementation of Set in Figure 8. An idle instance of this Set can Early-Reply to an Add(x) invocation in constant time, but will remain locked during its $O(log\ n)$ residual computation for inserting x into the BST. This is great if the client does not invoke another method on the Set during this time. The response time of invocations during the lockout period, however, can degrade to that of ordinary blocking components. This performance pattern is illustrated by an intensive calling sequence in Figure 10.

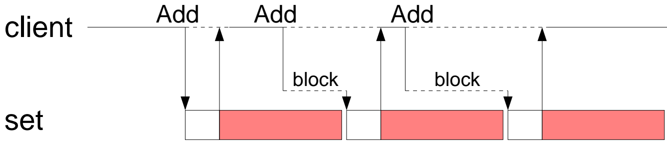


Fig. 10. Early-Reply Set with long residual computations

The key insight of this section is that the residual computation of a method can be reduced by offloading work to one or more subordinate Early-Reply components. When the internals of a component exhibit a high degree of independence, this goal can be accomplished without introducing novel stratagems. Sets, symbol tables, and databases are exemplary case studies in this respect. This section illustrates how *hashing* can be used as as a standard technique for reducing the residual computations for such components.

Consider a hashing implementation of Set represented using an array of subordinate Set instances. For clarity, we refer to this implementation as Hash-Set to distinguish it from the BST-Set implementation presented in Section 4. Hash-Set has two parameters: Item (which is the type of entry in the Set), and Base-Set (which is an arbitrary implementation of Set). A representation for this implementation contains a single member variable, denoted by self.table, which is of type Array of Base-Set. The hash table self.table represents the abstract Set as a string of subordinate Base-Set instances. This is a key feature to reducing the residual computations of the client-level Set methods. The Base-Set instances in each hash bucket are independent, so residual computations in multiple buckets can execute concurrently in addition to overlapping with subsequent invocations to the client-level Set. The representation invariant and abstraction function for Hash-Set are shown in Figure 11.

Hash-Set is also an Early-Reply implementation of the abstract component Set. The material computation of Add in Figure 12 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 Hash operation 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. Good hash functions are independent of the number of elements in the Set, and can often be computed in constant time. Thus, the primary determinant of the length of the residual computation of Add depends

Representation Invariant: The Base-Set instances in each bucket of the hash table are pairwise disjoint.

Abstraction Function: The abstract model of the Set is the union over all Base-Set instances in the hash table.

Fig. 11. Representation Invariant and Abstraction Function for Hash-Set

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

Fig. 12. Implementation of Add in Hash-Set

on the duration of the lower-level invocation to Add on the Base-Set instance in the appropriate hash bucket.

At this point, it should be clear that we can reduce the lockout time associated with the *residual computation* of Add on the client-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 to be selected at component-integration time. The only requirement on a component selected for Base-Set is that it implements the Set component specification. 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>. In the material computation of the client-level ER-Set, the Add operation simply swaps the input parameter into a local variable, say y, and then returns control to the client via Early-Reply. During the residual computation, y is hashed to the correct bucket, and Add (y) is called on the corresponding BST-Set. Since the material computation of this secondary Add is also constant-time for an idle BST-Set (see Section 4), the residual computation of the client-level ER-Set is effectively minimized.

Figure 13 illustrates a possible trace of intensive client-calling on an ER-Set component. Given the nature of good hashing functions, 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

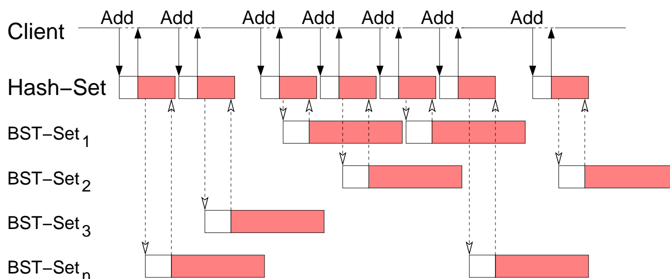


Fig. 13. Execution of ER-Set = Hash-Set<BST-Set>

ER-Set to minimize its residual computation, thereby decreasing the likelihood that subsequent client invocations will block.

The implementation of ER-Set is similar in mechanism to explicitly multi-threaded implementations that delegate client invocations to forking bucket updates. The primary advantage of our approach is that Early-Reply components encapsulate design choices about component concurrency and synchronization. In particular, the Set instances in each bucket support a synchronous *view* of component interaction. This insulates the hashing layer from having to reason about aspects of concurrency control or explicit thread management. A wider observation is that Early-Reply components are responsible for managing their own synchronization via local locking. This allows individual Early-Reply components to be incrementally deployed and integrated into existing or developing software projects without the need for system-wide synchronization support.

7 Lightweight Early-Reply Wrappers

In the previous section, we showed how the response time of an Early-Reply component could be improved by layering it on other Early-Reply components. In the absence of such components, however, what can one do? We address this question by presenting a lightweight, non-invasive technique for transforming some off-the-shelf, blocking components into Early-Reply components.

Consider an off-the-shelf blocking implementation of Set called Blocking-Set. One can transform this component into an Early-Reply component by using it to instantiate the parameterized Hash-Set wrapper presented in Section 6. Note that the resulting component Light-Set = Hash-Set<Blocking-Set> is now an Early-Reply implementation of Set with respect to the Add operation. When an Add(x) operation is invoked, the Hash-Set wrapper swaps the formal parameter with a local variable and then returns control to the caller with Early-Reply. Thereafter, the actual update can be delegated to the encapsulated Blocking-Set instance in the appropriate hash bucket.

The response time of the Add operation is now $O(1)$ for an idle Light-Set component, but it still suffers from the same problem that we identified with the BST-Set in Section 4; namely, the response time is $O(1)$ only under the condition that the client does not invoke subsequent operations on the Light-Set instance during its residual computation. Since the problem with Light-Set and BST-Set is the same, we can reuse the Hash-Set wrapper to minimize residual computations as we did in Section 6. Using Light-Set as the parameter to Hash-Set yields a new implementation Reuse-Set = Hash-Set<Light-Set> that is an Early-Reply component with minimized residual computations like the ER-Set from Section 6!

The foregoing wrapper technique is a reuse-oriented approach to transforming off-the-shelf blocking components into Early-Reply components. But when can this technique be applied? Methods that either *consume* or *preserve* their formal parameters can be directly recast as Early-Reply implementations using the wrapper approach. Since a layered-wrapper implementation uses the underlying implementation to service its clients, blocking implementations of methods

that *produce* results are more difficult to transform into Early-Reply implementations. For some methods that produce their final parameter values, however, Early-Reply can be exploited to *prefetch* results that are not functionally dependent on input parameter values. This is accomplished by executing the material computation of a future invocation as part of the residual computation of the current invocation. For further applications and illustrations of Early-Reply, we refer the interested reader to our technical report [10].

8 Related Work

Our work relates to research in the area of active objects, where each object in the system has its own thread of control. The post-processing section of methods in the language POOL [1] bears closest resemblance to our notion of residual computations. In POOL, all objects are active, and therefore execute on their own thread. An object executes its *body* upon creation until it receives a method invocation in the form of a message from another object. The method computes the results (if any), returns them to the caller, and then proceeds with its post-processing section while the caller continues execution.

Several people have worked on introducing concurrency into object-oriented systems. Most of this work is in the form of active object extensions to current OO languages [11,2]. These languages allow a method to return control to the client before the method actually completes execution. When the client later tries to access any of the formal parameters that were passed to the method, it is forced to block at that time. This is called *wait-by-necessity* [2] and the only synchronization used is *data-driven synchronization.*

Early-Reply is also related to amortized algorithms. Recall the implementation of the Length operation for Stack in Section 2. A naïve implementation of Length would count the number of items in the Stack upon each invocation. Instead, Push and Pop can be implemented to increment or decrement a local data member self.len to record the current length. This amortizes the net cost of Length over other Stack operations, so the response time of Length can be constant-time. The spirit of Early-Reply is similar, except that it amortizes the cost of an operation over calls to *different* components, rather than over calls to the *same* component.

Our research also relates to work on prefetching or precomputing results in anticipation of future method invocations [9]. One example of this would be a pseudo-random number generator. When a client requests the next number, the method can return a number, and then precompute the next number, so that it is available on demand for the next invocation. With Early-Reply, prefetching can be incorporated into the residual computation of previous invocations, thereby minimizing the material computation of future invocations. Another application of prefetching is in the tokenization phase of a compiler. The residual computation of a Get_Token method could prefetch and buffer subsequent tokens until the next invocation arrives. In general, prefetching can be applied to any method that produces a result that can be computed deterministically, independent of the input parameters and the method calling sequence.

Finally, a related approach to improving the efficiency of component operations is by incremental computation. The underlying idea is to transform programs so that an operation uses information from a previous run to compute future results [5]. When called repeatedly, such an operation performs better than a regular implementation since the entire operation is not executed every time; instead, results from a previous invocation are reused.

sectionConclusion

Component-based development is a cornerstone of scalable systems. Despite the benefits of componentry (both realized and promised), generic software suffers from a reputation for being inefficient. Among the cited benefits we include modular reasoning about components, abstractions, and their composition mechanisms. An acknowledged drawback, however, is the calling overhead associated with crossing the encapsulation barrier to interact with systems of components in layered or hierarchical compositions. Although procedure-calling overhead is rightfully a cost of information hiding, we believe that it may best be viewed as an *opportunity cost* for exploiting abstractions.

This paper has presented research directed at improving the performance of component-based software by exploiting the abstraction of a synchronous calling model. We have presented Early-Reply as a construct for introducing concurrency into synchronously-viewed systems by decoupling the flow of data to the caller from the flow of control within a component method. When components are designed to encapsulate design decisions about their data and implementation, Early-Reply can offload residual method computations to execute in parallel with subsequent client activities. In addition to reducing the response time of idle components, Early-Reply can be used in layered component implementations to increase resource utilization and reduce overall time complexity. We have also shown how lightweight, parameterized wrappers can be used to transform some off-the-shelf blocking components into efficient Early-Reply components.

Early-Reply leverages the performance benefits of encapsulated concurrency without compromising the sequential reasoning benefits of synchronous calls. To support this view, Early-Reply components lock themselves during their residual computations; this defers invocations that could otherwise compromise design-by-contract. In contrast to data-driven synchronization schemes which require system-wide support for deployment, Early-Reply components also encapsulate responsibility for their own synchronization via local locking. This aspect is critical to knowledge-transfer and reuse, because Early-Reply components can be incrementally deployed into both existing and developing systems.

Early-Reply presents many exciting avenues for future research including:

- Formal semantics and proof rules for Early-Reply.
- Formal 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 intrinsically concurrent problems.
- Generalizing Early-Reply in the context of caller-callee synchronization.

Acknowledgments

We wish to thank the anonymous reviewers whose insightful comments greatly improved this paper. Also, we gratefully acknowledge financial support from the National Science Foundation under grant CCR-0081596, and from Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or Lucent.

References

1. P. America. POOL–T: A parallel object–oriented language. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*, pages 199 – 220. MIT Press, 1987.
2. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1990.
4. D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Soft. Eng.*, 17(5):424–435, May 1991.
5. Y. A. Liu. *Incremental Computation: A SemanticsBased Systematic Transformation Approach.* PhD thesis, Dept. of Computer Science, Cornell University, 1996.
6. B. Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance and Classification)*, 25(10):40–52, 1992.
7. 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.
8. D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Comm.of the ACM*, 18(7):401–408, 1975.
9. H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, 1990.
10. S. M. Pike and N. Sridhar. Early Reply: A basis for pipebranching parallelism with sequential reasoning. Technical Report OSU-CISRC-10/01-TR13, Dept. of Computer and Inf. Science, Ohio State University, October 2001.
11. G. Roberts, M. Wei, and R. Winder. Harnessing parallelism with UC++. Technical Report RN/91/72, Dept. of Computer Science, University College, London, 1991.