

Encapsulating Concurrency with Early-Reply

Scott M. Pike
Computer & Info. Science
The Ohio State University
pike@cis.ohio-state.edu

ABSTRACT

Component methods often produce their final parameter values long before the method body is ready to terminate. To minimize client blocking, **Early-Reply** can be used to forward invocation results to the caller as soon as they are (safely) available. After executing **Early-Reply**, the method remainder and the client caller can proceed concurrently, modulo synchronization constraints. The prime motivation for **Early-Reply**, then, is to improve performance factors such as response time and resource utilization.

Early-Reply received previous attention as a construct for explicit concurrent programming. Its value for *sequential* programming, however, has not been widely recognized. The present research supplies a formal treatment of **Early-Reply** as a basis for concurrent execution of sequential programs. In particular, we reformulate **Early-Reply** under *local* proof obligations that encapsulate concurrency as a (temporal) unit of information hiding. The upshot is that software developers can use **Early-Reply** to exploit the performance benefits of concurrent execution, without compromising the reasoning benefits of sequential programming.

1. BACKGROUND

Early-Reply (also known as “post-processing”) originated as a concurrent programming mechanism for decoupling control flow from data flow. Canonic applications include spawning asynchronous background activities [7, 4], and ensuring that child threads are properly initialized prior to subsequent interaction with their parents [11]. **Early-Reply** has appeared in explicitly concurrent languages such as Lynx [10] and OREXX [5], and it is similar to related mechanisms for introducing concurrency in active object languages [1, 2, 3].

Other things being equal, we consider better performance to be better. A common drawback of **Early-Reply** in concurrent languages, however, is that *logical concurrency* can bleed into the reasoning system and thereby complicate the proof obligations required to establish program correctness [8]. At a minimum, testing and debugging will become more

complex unless the potential for runtime concurrency can be encapsulated from explicit consideration. Ideally, one would like to use **Early-Reply** to improve performance via runtime concurrency, but without compromising the benefits of sequential reasoning about program behavior. To this end, we have reconsidered **Early-Reply** in the domain of sequential programming [9].

2. SYNCHRONOUS CALLS

Component method bodies can generally be partitioned into a *material segment* and a *residual segment*. The material segment denotes the initial computation required to satisfy the postcondition of the method. The residual segment denotes the subsequent computation required to reestablish the representation invariant, optimize its structure, pre-fetch future results, etc. In a naïve implementation of a synchronous calling model, the caller blocks during the residual computation, even though the results of the invocation may already be determined. 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.

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, **Early-Reply** can be used to decouple the flow of data back to the caller from the flow of control in the method body. **Early-Reply** provides a basis for component-level parallelism and increased resource utilization by enabling methods to forward final parameter values back to the caller as soon as their final values are determined.

3. OPERATIONAL SEMANTICS

Informally, we may view **Early-Reply** 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 remainder of the method body. **Early-Reply** “locks” the component instance *self* for the duration of its residual computation. An invocation on the component during this

period blocks the caller until the current invocation completes, whereupon `self` becomes “unlocked” to handle the next method. During the residual computation, the component instance is obligated to reestablish its representation invariant and to satisfy the method’s postcondition with respect to `self`. Since local locking enforces mutual exclusion, the component instance can never be observed in an inconsistent state. This is critical to supporting compositional reasoning in a synchronous calling model, despite the potential for concurrent execution. Finally, `Early-Reply` is considered idempotent; subsequent occurrences within the same method body are executed as a `skip`.

4. PROOF OBLIGATIONS

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, we must encapsulate the potential concurrency of residual computations from client observability. A conservative approach would require *all* formal parameter values to satisfy the method’s postcondition when replied to the caller. We can amend this policy to exclude the distinguished parameter `self`, which can employ a local locking mechanism to synchronize itself without system-wide support. Upon termination, we must dispatch further proof obligations that the representation invariant is true and that the postcondition holds for *self*.

To preserve sequential client-side reasoning about component behavior, we add the following proof obligation; namely, residual computations cannot change (write) the values of replied parameters. 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 read *aliased* parameter values. 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. This problem can be dispatched by replacing assignment with *swapping* as the primary data-movement operator, since it can be implemented to execute in constant time without creating aliases [6].

Formalizing the foregoing proof obligations is at the heart of our current work. In particular, we require a formal commutativity proof to establish the following result: any legal interleaving of actions between a client and a component’s residual computation is isomorphic to the sequential interleaving (which consists of the sequence of residual actions followed by subsequent client actions).

5. PIPEBRANCHING COMPONENTS

In earlier work [9], we motivated the value of `Early-Reply` for sequential programming by showing how `Early-Reply` can be dovetailed with standard techniques such as hashing, prefetching, and amortized methods to improve performance. Criticisms have claimed, however, that (a) `Early-Reply` provides insufficient *fan-out* to improve performance substan-

tially; and (b) any such gains would be swamped by data-driven synchronization [8]. Our research addresses these concerns by focusing on layered, component-based development in which the benefits of `Early-Reply` are multiplied by the fan-out of the component hierarchy. In hierarchical software, a single client invocation can precipitate multiple lower-level invocations on subcomponents whose fan-out can realize potentially pipebranching parallelism. Moreover, overhead costs are minimized by eschewing data-driven synchronization in favor of a local-locking mechanism. By avoiding a system-supplied synchronization policy, `Early-Reply` components can better support incremental deployment into both existing and developing applications.

In addition to theoretical contributions, our work also offers practical value. We plan on injecting the formal results of our research into enterprise languages. `Early-Reply` can be “faked” in modern languages with explicit multithreading; doing so, however, undermines the benefits of reasoning sequentially about `Early-Reply` components. Thus, we are developing a library-based implementation that realizes `Early-Reply` as a first-class language construct in both Java and C#. The resulting product will provide both a language mechanism for software development as well as a research vehicle for further performance analysis and benchmarking.

6. REFERENCES

- [1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
- [2] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object Oriented Concurrent Programming*, pages 199 – 220. MIT Press, 1987.
- [3] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
- [4] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development*. Addison-Wesley, 1993.
- [5] Tom Ender. *Object-Oriented Programming with REXX*. John Wiley and Sons, Inc, 1997.
- [6] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Engineering*, 17(5):424–435, May 1991.
- [7] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.
- [8] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, 2000.
- [9] Scott M. Pike and Nigamanth Sridhar. Early reply components: Concurrent execution with sequential reasoning. In *Proceedings of the 7th International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 46–61. Springer-Verlag, April 2002.
- [10] Michael L. Scott. The lynx distributed programming language: Motivation, design, and experience. *Computer Languages*, 16(3):209–233, 1991.
- [11] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.