Efficient Compilation of Concurrent Call/Return Communication in Actor-Based Programming Languages

W. Kim and G. A. Agha *
Open Systems Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: { wooyoung | agha}@cs.uiuc.edu
URL: http://www-osl.cs.uiuc.edu

R. B. Panwar†
Application Dev. Technology Institute
IBM Santa Teresa Labs
San Jose, CA 95141, USA
Email: rajendrap@VNET.IBM.COM

Abstract

Concurrent call/return communication (CCRC) allows programmers to conveniently express a communication pattern where a sender invokes a remote operation and uses the result to continue its computation. The blocking semantics requires context switching for efficient utilization of computation resource. We present a compilation technique which allows programmers to use CCRC with the cost of non-blocking asynchronous communication plus minimum context switch cost. The technique transforms CCRCs into non-blocking asynchronous sends and encapsulates continuations into separate objects. A data flow analysis is used to guarantee that only necessary context is cached in continuation objects.

1 Introduction

Actors are fine-grained active objects which encapsulate a thread as well as data and procedures. Computation is expressed in terms of actors communicating each other via asynchronous message passing. Since communication abstracts computation, the efficiency of communication and scheduling mechanisms determines execution performance of applications.

The most efficient mechanism for an actor to communicate with remote actors is non-blocking asynchronous communication [1]; neither a sender nor a receiver need block on communication. The lack of context switching means less overhead in communication and scheduling, which is translated to better performance. On the other hand, programming with asynchronous communication often requires explicit manipulation of continuations and synchronization. The separation of computation's logical flow into multiple continuations may degrade readability and make actor programming tedious and error prone.

In contrast, call/return communication (e.g. remote procedure call) which captures a communication pattern where a client invokes a procedure on a server and uses the returned result to continue its computation (i.e., the continuation to the result) makes unnecessary the explicit continuation and synchronization manipulation by employing blocking semantics which comply stack discipline. Concurrent call/return communication (CCRC), a concurrent version of call/return communication, has more concurrent semantics in that nothing but data dependence constrains execution orders between multiple request message sendings in a method and between computations resulted by them. CCRC greatly improves programmability and readability by freeing programmers from manipulating continuations and synchronization. Unfortunately, convenience never comes without cost. CCRC's implicit blocking semantics demand context switching – an expensive operation without special hardware support – for better utilization of computation resources.

In this paper we propose a compilation technique and runtime support to implement CCRC. The implementation allows programmers to use CCRC with the

*The research described has been made possible by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). We thank JaeHoon Kim and JoonKyoo Yoo for their careful review of the draft of the paper. We also thank the UIUC NCSA for use of their TMC CM-5 machine.

†The author was at the Open Systems Laboratory, University of Illinois Urbana-Champaign when the work related to the paper was done.

[1] We assume that reliable communication is guaranteed by the runtime system or by some hardware support.

cost of non-blocking asynchronous communication plus minimum context switch cost. For a set of mutually data-independent request message sendings, it transforms each request into a non-blocking asynchronous send and separates out their *join continuation* [3, 11] into a *join continuation closure* (JCC) [10]. Join continuation is a continuation which is dependent simultaneously on multiple replies; it is executed as all the replies are available. JCC encapsulates a join continuation and caches the context for its execution. The compiler minimizes the context to be cached through a separate data flow analysis.

2 Background

This section presents a brief introduction to the Actor model and an informal description of the semantics of CCRC. It also describes an implementation technique of CCRC adopted in other systems and motivates the development of our compilation technique.

2.1 The Actor model

Actors [1, 2] are fine-grained active objects which encapsulate a thread as well as data and procedures. Computation is expressed in terms of actors communicating each other through asynchronous message passing. Each actor is assigned a unique mail address which is used for other actors to send it messages. Since communication is asynchronous, messages are buffered on their destinations. In response to a message, an actor may: send messages to other actors, create new actors, and change its state to process the next message. Though simple and primitive, actor operators form a powerful set upon which to build a wide range of higher-level abstractions.

2.2 The language

We incorporated CCRC in a high-level actor-based language THAL [4, 11] which also provides other high-level communication abstractions, such as local synchronization constraints, delegation, and multicast. THAL supports CCRC using two primitives, `request` and `reply`. `request` invokes a method on a remote actor and `reply` sends a result back to the requester. Figure 1 shows a THAL version of an N-queen problem [13] which computes the number of the solutions. (“.” represents `request` message send.) The THAL runtime system [10] is currently running on the TMC CM-5 [14].

```
behv NQueen
 method comp(col,diag1,diag2,maxcol,depth)
  | i, c, c1, sols, replies, where |
  replies = zeros(N);
  c = ((col|diag1)|diag2);
  c1 = ((c+1) & ~c);
  sols = 0;
  for i = 1, N do
   if (c1 > maxcol) then break; end
   if ((col | c1) == maxcol) then
    sols = sols + 1;
   else
    if (depth < N/2) then
     where = random () % #no_nodes;
     replies[i] = (NQueen.new() on where).comp(
      (col|c1),(((diag1|c1)<<1)&maxcol),
      ((diag2|c1)>>1),maxcol,depth+1);
    else
     replies[i] = (NQueen.new()).comp(
      (col|c1),(((diag1|c1)<<1)&maxcol),
      ((diag2|c1)>>1),maxcol,depth+1);
    end
   end
   c1 = (((c1<<1)+c)&~c);
  end
  reply(add_all_replies(replies)+sols);
 end
end
```

Figure 1: THAL version of an N-Queen problem.

2.3 Semantics of CCRC

Although CCRC captures the same communication pattern, namely call/return communication, as remote procedure call (RPC), it has concurrent execution semantics distinct from the RPC semantics. Both CCRC and RPC have implicit blocking semantics where a caller blocks on a remote call (or, a remote send) until the result is available. However, execution order between multiple CCRCs is constrained only by data dependence (i.e., *concurrent*) whereas control dependence as well as data dependence affects execution order between multiple RPCs (i.e., *sequential*). Consider a method which has two remote calls, one next to the other. Assume that the two calls have no data dependence between them. Although they may be executed concurrently, an RPC-based implementation guarantees that the computation caused by the first call is completed before the execution of the second call. Consequently, the first call's result is available by the time the second call is made. By contrast, a CCRC-based implementation allows concurrent execution of the two calls. The second call may be made before the first one is completed; the second call's result may even be

available earlier than the first one's. Moreover, it enforces no execution order among computations induced by the two calls. Thus, programmers are discouraged to assume any specific message delivery order when using CCRC. If it is necessary to enforce a certain processing order on messages, the order must be specified explicitly using separate programming constructs, such as local synchronization constraints [11].

2.4 Futures

A technique used in a number of systems to implement CCRC-like abstractions is *futures* [9]. Futures are a promise for a value – a place holder whose value is yet to be defined. A future-based CCRC implementation creates a future for a request send and returns it as the result. Instead of blocking on the request send, sender's computation continues to execute with the future until its value is actually accessed (the future is said to be *touched*). If the value is already available when a future is touched computation continues with the value. Otherwise, the computation blocks on the future until its value is available. Although simple and easy to implement, futures have a potential source of performance bottleneck: concurrent execution using *futures* requires context switching, an expensive operation without special hardware support (e.g., 52 μsec in the TMC CM-5 [6]). Suppose an actor creates N children in response to a `comp` message in Figure 1. A future-based implementation will create N futures and assign them to the identifiers `replies[i]` for i ranging from 1 to N. The next expression to be evaluated is a summation of the returned results; here the execution blocks on a future whose value is not available. In the worst case, $2N$ context switches are required to complete the computation (i.e., 2 for each touch). Furthermore, out of order arrival of replies does not reduce the time to complete the sender's computation because futures are generally touched in sequence according to the order they appear.

3 Join continuation transformation

In this section we present a source-to-source transformation algorithm as an alternative to futures. The algorithm examines data dependence relation between request sends and other statements to extract join continuations. Since join continuation concurrently waits for multiple replies, it may implement CCRC with far less cost than futures. We start with a base algorithm which exploits *functional parallelism* existing in message/function argument evaluation. Then, we augment it with a data dependence analysis.

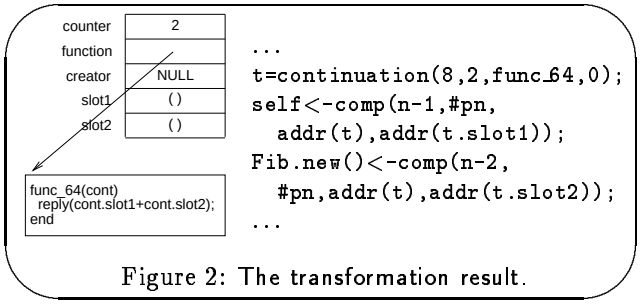


Figure 2: The transformation result.

3.1 Base algorithm

Let Δ_m denote the data dependence relation of a method m, Γ_m denote the control dependence relation of m (method execution is sequential), and R_m be a set of all `request` expressions in m. We define a *request send partition* $RSP_{m,k} \subset R_m$ such that for any $r_i, r_j \in RSP_{m,k}$, $(r_i, r_j) \notin \Delta_m$, $(r_j, r_i) \notin \Delta_m$, $(r_i, r_j) \notin \Gamma_m$, and $(r_j, r_i) \notin \Gamma_m$. Note that RSP is defined such that request sends in the same message send/function call belong to the same RSP provided that they have no data dependence between them.

The first step of the algorithm is to partition R_m into RSPs. Then, all `request` sends are promoted as assignments to compiler-generated temporary variables so that `request` sends belonging to the same RSP are placed adjacently. The resulting method should preserve Δ_m and Γ_m. The second step is to split the method. First, the control flow graph of m is traversed in a topological order. As soon as an RSP is encountered, m is split and the portion of m dependent on the RSP is separated into a join continuation. An allocation statement of an actor which encapsulates the join continuation is inserted before the RSP and each member `request` of the RSP is transformed into a non-blocking asynchronous send with a reply address to the actor. Finally, the split process is recursively applied to the join continuation. RSPs enclosed in structured statements, such as `if` and `for`, need similar but much finer split processes. Figure 2 has the transformation result of:

```
method comp (n)
 if (n<=1) then reply(1);
 else reply(self.comp(n-1)+(Fib.new()).comp(n-2));
end.
```

3.2 Join Continuation Closure

The behavior of join continuation is deterministic: as soon as all the expected replies are received, it executes

the specified computation and cannibalizes itself. We exploit the deterministic behavior and implement join continuation using *join continuation closure* (JCC). JCC has four components, *counter*, *function*, *creator* and *argument slots*. *Counter* contains the number of empty *slots*. As soon as all slots are filled (reply address is defined by a triple <`node number`, `JCC address`, `slot address`>), *function* is invoked with the JCC as its argument. Some slots are reserved for the continuation's execution context and filled at the creation time. Using data flow analysis the compiler guarantees that only necessary context is cached in the JCC, thereby minimizing the context switch cost.

3.3 Common continuation region analysis

Since the base algorithm was designed to exploit only the functional parallelism existing in argument evaluation, it is not capable to identify that `request` sends in Figure 1 are indeed mutually data-independent and can be executed concurrently. We developed a more general transformation framework which can identify mutually data-independent `request` sends across statement boundary.

The analysis is based on a simple observation: any two `request` sends which are executed in a method execution may be concurrently executed as long as they have no data dependence between them. Given an abstract syntax tree $AST = (AV, AE)$ for a method m [2], we define the common continuation region at $s \in AV$ (CCR_s) to be a subtree rooted at s in which all `request` sends are data independent each other and thus may share the same join continuation. A *maximal* CCR is defined to be a CCR which is not contained in any other CCRs in the AST.

The transformation begins by partitioning a method into maximal CCRs. After coloring the AST using the algorithm in Figure 3, maximal CCRs are maximal subtrees rooted at a gray colored node. Consecutive maximal CCRs whose roots are siblings each other and which have no data dependence between them are merged into a larger one. Figure 4 illustrates how maximal CCRs are determined for a simpler version of the method in Figure 1. After all maximal CCRs are determined, the conservative estimate number of slots to be filled by replies is computed and a join continuation is separated for each maximal CCR. If the estimate may not be determined, the subtree is changed to non

[2] We assume that all `request` expressions have been promoted to assignment statements as in the base algorithm. Since all relevant information is at statement level, we assume that AV is a set of all statements in m.

INPUT: The abstract syntax tree $AST = (AV, AE)$ of a method
OUTPUT: A colored abstract syntax tree
ALGORITHM: For each $s \in AV$, let $Child_s = \{c \mid (s, c) \in AE\}$ and let $color_s$ denote a color assigned to s.

1. Color each leaf node of the AST with *gray* if it is a request send or *white* if it is not.
2. For each $s \in AV$, color s with
   - *black* if $\exists c \in Child_s$, $color_c$ is *black*.
   - *white* if $\forall c \in Child_s$, $color_c$ is *white*.
   - *gray* if $D = \emptyset$ or *black*, otherwise. D is defined as $\{d \mid \text{for } c \in Child_s,\ color_c \text{ is } gray \wedge r \in R_c \wedge d \in AV_s \wedge r \mapsto d\}$ where R_c is a set of `request` send statements in a subtree rooted at c and AV_s is a set of statements in a subtree rooted at s.

Figure 3: Coloring algorithm

CCR and join continuations are extracted from smaller CCRs in the subtree. As before, all `request` sends are transformed into non-blocking sends with a reply address. Finally, some bookkeeping code are generated to keep track of the number of requests that have actually been sent.

4 Performance

We present the performance results of three applications which are written using CCRC. All the programs were written in THAL [10, 12] and executed on a 32-node partition of a TMC CM-5 [14]. Each node contains a 33 MHz Sparc processor. Table 1 has timings of primitive operations or the runtime system.

Operation		Time
send-and-dispatch	local	0.45/0.67
	remote	9.91
reply	local	2.76
	remote	9.26
continuation	creation	2.27
	deallocation	0.75

Table 1: Performance of primitive operations of THAL on the CM-5 (μsec). Time for sending messages and replies are measured by repeatedly sending zero-size messages.

The first example is a Fibonacci number generator where each actor creates two children, waits for the

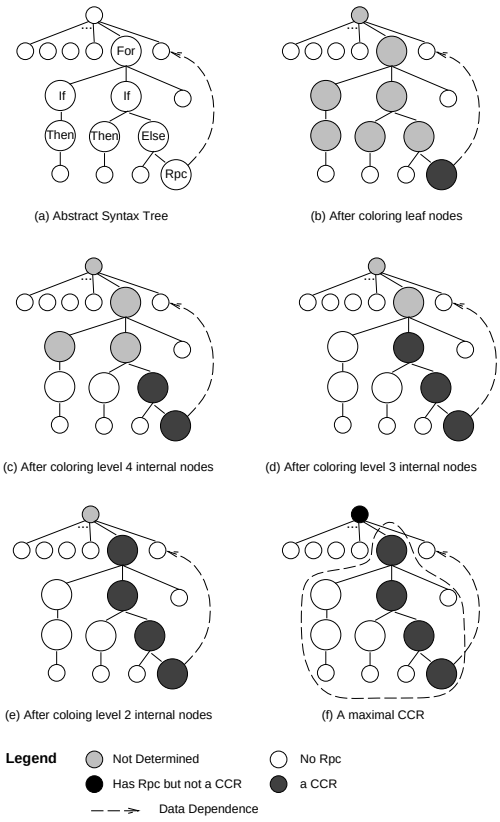
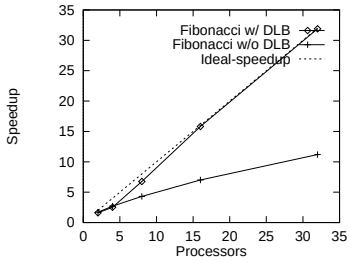


Figure 4: Coloring of the abstract syntax tree of an N-Queen implementation. The implementation has a `request` send inside `if` block contained in a `for` loop. There is a data dependence from the `request` send to a statement outside the `for` loop (shown as a broken arc). Only the dependence from the `request` send to other statements is shown. Small white leaf nodes represent simple statements.

results, computes the sum, and replies the sum to its creator. Although it is a very simple program, it is extremely concurrent and fine-grained and shows a great deal of load imbalance. The compiler optimizes away all actor creations since Fibonacci actors are purely functional. Figure 5 has two sets of performance numbers, one without and the other with dynamic load balancing [10]. As a point of comparison, executing the Fibonacci of 33 using the Cilk system [6] takes 73.16 seconds and an optimized C version completes in 8.49 seconds on a single node of a TMC CM-5.

The second example is an N-Queen problem. The goal of N-Queen problem is to find a solution which places N queens on an $N \times N$ chess board such that no two queens are placed on the same row, column, or diagonal (i.e., no queens attack each other). In our evaluation we used a version given in [13] which computes the number of the solutions for a given chess board (Figure 1). Each actor represents a queen placed on a square defined by a column and a row. An actor's ancestors plus the actor define a partial solution up to the column that the actor represents. An actor computes the number of the solutions by creating actors which represent queens placed on squares of the next column which are compatible with the partial solution. It waits for the replies from its children, computes the sum, and returns it to its creator. The computation is load-balanced by using a subtree-subcube placement. The performance result is shown in Figure 6. In spite of the fine-granularity, we were able to obtain about 24 times speedup using 32 processors.

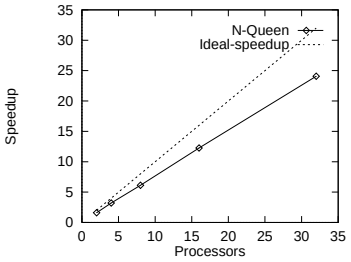
PEs	w/o DLB	w/ DLB
1	55.5	60.7
2	32.3	37.9
4	20.8	24.1
8	12.9	8.98
16	7.94	3.84
32	4.96	1.87



(a) Times (sec.) (b) Speedup

Figure 5: Fibonacci number generator

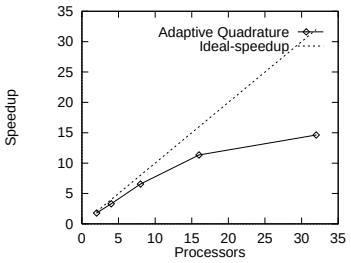
PEs	Times
1	186.4
2	116.5
4	58.0
8	30.4
16	15.2
32	7.74



(a) Times (sec.) (b) Speedup

Figure 6: 13-Queen problem

PEs	Times
1	0.912
2	0.513
4	0.276
8	0.139
16	0.0804
32	0.0623



(a) Times (sec.) (b) Speedup

Figure 7: Adaptive quadrature problem

The last example is an adaptive quadrature problem. It was implemented using master-worker configuration. First, the grand master creates a worker on each processing node and assigns it an equal-sized subinterval. Each worker computes the integral estimate on the subinterval as well as the error estimate. If the error estimate is sufficiently small, it returns its integral estimate to its master as the result. If the error estimate is too large, it refines the integral estimate by creating child workers, dividing its subinterval, and assigning one to each child. Then, it waits for the results, computes the sum, and returns it to its master. In the experiment the following function was integrated over the interval from 0 to 10π with $c_i = 1000$, i = 1,2,3 to intentionally cause load imbalance between workers.

$$f(x) = \begin{cases} c_1 & \text{if } 0 \le x \le 5\pi, \\ \mid c_2 \sin(c_3 x) + c_1 \mid & \text{if } 5\pi < x \le 10\pi. \end{cases}$$

The initial grid size is 0.001 and the error bound is set to 10^{-5}. In the implementation, a worker halves its interval as it refines its integral estimate. It creates one new worker and reuse itself as its own worker to reduce overhead. The newly created worker is placed using a round-robin placement to dynamically balance load among processors. However, we could not use arbitrarily small grid size because of rounding error. As a result, computation granularity decreases as intervals are getting smaller and the poor speedup was resulted in (Figure 7).

5 Conclusion

In this paper we presented a compilation technique and runtime support to implement concurrent call/return communication (CCRC) in actor-based programming languages. A technique similar to our base algorithm has been used in compilation of explicit message passing programs [8]. Data structures similar to join continuation closure (JCC) have been used in fine-grained functional languages [5, 7] and an explicit continuation passing style language [6]. CCRC's blocking yet concurrent semantics requires context switching for efficient utilization of computation resources. Our compilation technique minimizes the number of context switches necessary for implementing CCRC by transforming request sends into non-blocking asynchronous sends and separating join continuations into JCCs.

References

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

[2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[4] G. Agha, W. Kim, and R. Panwar. Actor Languages for Specification of Parallel Computations. In G. E. Blelloch, K. Mani Chandy, and S. Jagannathan, editors, *DIMACS. Series in Discrete Mathematics and Theoretical Computer Science. vol 18. Specification of Parallel Algorithms*, pages 239–258. American Mathematical Society, 1994.

[5] R. S. Nikhil Arvind and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, 1994.

[7] D.E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of ASPLOS*, pages 166–175, 1991.

[8] J. G. Holm, A. Lain, and P. Banerjee. Compilation of Scientific Programs into Multithreaded and Message Driven Computation. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 518–525, Knoxville, TN, May 1994.

[9] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, 1985.

[10] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Supercomputing '95*, 1995.

[11] W. Kim and G. Agha. THAL: A High-level Actor Language and its Compilation. submitted for publication, 1996.

[12] R. Panwar, W. Kim, and G. Agha. Parallel Implementations of Irregular Problems using High-level Actor Language. In *Proceedings of IPPS '96*, 1996.

[13] K. Taura. Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers. Master's thesis, The University of Tokyo, Feburuary 1994.

[14] Thinking Machine Corporation. *Connection Machine CM-5 Technical Summary*, revised edition, November 1992.