An Inter-entry Invocation Selection Mechanism for Concurrent Programming Languages

Aaron W. Keen[1] and Ronald A. Olsson[2]

[1]Computer Science Department,
California Polytechnic State University,
San Luis Obispo, CA 93407 USA
akeen@csc.calpoly.edu

[2]Department of Computer Science,
University of California, Davis,
Davis, CA 95616 USA
olsson@cs.ucdavis.edu

Abstract. Application-level message passing is supported by many concurrent programming languages. Such languages allow messages to be generated by invoking an entry. Messages are removed from an entry by an invocation selection mechanism. Such mechanisms may allow selection from a set of entries (multi-way receive). Many concurrent languages provide support for multi-way receives, but they are limited in their expressive power. This paper presents three proposed inter-entry selection mechanisms. The proposed mechanisms overcome the limitations of the existing mechanisms. Each of these mechanisms allows an invocation selection algorithm to examine the entire set of pending invocations (and their parameters) as part of the invocation selection process. These mechanisms are analyzed and compared both qualitatively and quantitatively.

1 Introduction

Many concurrent programming languages support application-level message passing as a form of thread synchronization. A message, henceforth called an *invocation*, is generated when a process invokes an operation (i.e., an entry or port), optionally passing it parameters. Once generated, the invocation is added to the set of pending invocations associated with the invoked operation. An invocation is removed from an operation's set by a servicing process using an invocation servicing mechanism. Such a mechanism, depending on the language, may allow selection of an invocation from a single operation (a receive) or from one of potentially many operations (a multi-way receive).

This paper focuses on multi-way receives and the invocation selection support provided. Ada [10], Concurrent C [8], CSP [9], Hermes [15], JR [12], Limbo [7], occam [4, 14, 13], Orca [3], SR [2, 1], and SRR [6] each provide support for multi-way receives, but with differing expressive power. None of these languages provides general support for inter-entry invocation selection. Specifically, these languages do not provide simultaneous access to all pending invocations (both within an operation's set and over the set of operations) during selection. Inter-entry selection facilitates the implementation of, for example, lottery scheduling for resource management [16] (in which a manager process picks a request randomly) and preferential job scheduling [6] (in which preference is given to normal interactive jobs unless there is a superuser batch job).

This paper presents three proposed inter-entry invocation selection mechanisms that address the limitations discussed above. The designs combine aspects of functional, object-oriented, and concurrent programming languages. The result of this study is the addition of the most balanced (in terms of abstraction and performance) mechanism to the JR concurrent programming language [12], which provides threads and message passing.

The rest of this paper is organized as follows. Section 2 discusses the limitations of multi-way receives provided by other languages. Section 3 describes the details of the proposed selection mechanisms. Section 4 analyzes the proposed mechanisms. Section 5 compares the proposed mechanisms. Section 6 concludes. Further discussion and details appear in Reference [11].

2 Background

Many concurrent languages provide support for multi-way receives, but with limited expressive power. For example, Ada provides the `select` statement, which allows multiple `accept` statements, each servicing an operation. When a `select` statement is executed, an operation is nondeterministically selected from those with pending invocations, and an invocation from the selected operation is serviced. Figure 1 demonstrates, using JR syntax (where an `inni` statement specifies a multi-way receive), a simple two-way receive that services invocations from two operations. Each `accept` statement may be preceded by a boolean guard to restrict the operations considered for servicing. A guard, however, cannot access an invocation's arguments and, thus, cannot use the values of these arguments in deciding which invocation to select. These guards provide a very coarse-grained control over invocation selection.

```
 1 public class Server {
 2    public op void entryOne(int priority, int b);
 3    public op void entryTwo(int priority, float b);
 4    protected void server() {
 5    /* repeatedly service an invocation from entryOne or entryTwo */
 6       while (true) {
 7          inni void entryOne(int priority, int b)  { /* service */ }
 8          [] void entryTwo(int priority, float b)  { /* service */ }
 9        }
10    }
11 }
```

Fig. 1. Simple Two-way Receive.

SR provides a more flexible invocation selection statement called the input (`in`) statement. A guard on an arm of an input statement can contain a *synchronization* expression and a *scheduling* expression. The former specifies which invocations are acceptable for servicing; the latter specifies the order in which to service acceptable invocations. Figure 2 shows a modification of the two-way receive from the earlier example, lines 7–8 in Figure 1. This modified two-way receive uses scheduling expressions (specified via `by` clauses) to service each operation's invocations in order of highest priority (lowest integer value). Unlike the guards in Ada, SR's synchronization and scheduling expressions can access an invocation's arguments and use their values to determine an invocation's acceptability and to order invocations. Such expressions, however, cannot simultaneously access the arguments of multiple invocations, and cannot be used to order invocations between multiple operations.

```
inni void entryOne(int priority, int b) by -priority  { /* service */ }
[] void entryTwo(int priority, float b) by -priority  { /* service */ }
```

Fig. 2. Two-way Receive with Scheduling Expression.

SRR extends SR to provide the `rd` statement, which allows the examination of all pending invocations of a set of operations. The `rd` statement, however, cannot directly service an invocation. An invocation is marked for service by a `mark` statement and is serviced by a subsequent `take` statement. Though this approach allows selection based on the pending invocations, the separation of selection into three non-atomic stages (`rd`, `mark`, and `take`) can complicate solutions.

None of the aforementioned mechanisms provides support for implementing selection algorithms that require atomic access to the entire set of pending invocations. This class of selection algorithms includes debugging, visualization, and scheduling (e.g., lottery scheduling). Extending the example in Figure 2, these mechanisms cannot directly enforce the priority ordering between operations; an invocation in `entryTwo` may be serviced even if `entryOne` has a higher priority invocation pending.

3 Proposed Invocation Selection Mechanisms

An acceptable inter-entry invocation selection mechanism must:

- allow atomic selection of any pending invocation in the set of operations serviced.
- provide access to each invocation's actual arguments.
- disallow removal of multiple invocations from the set of pending invocations.
- disallow insertion into the set of pending invocations as a side-effect.

We devised three approaches that satisfy these criteria: Invocation Enumeration, Functional Reduction, and Hybrid. Invocation Enumeration and Functional Reduction are discussed below. The Hybrid approach combines aspects of the other approaches, but mixes programmer abstractions and performs poorly, so it is not discussed further (though the experiments do include Hybrid).

3.1 Invocation Enumeration

The Invocation Enumeration approach provides an enumeration of the currently pending invocations. This enumeration is passed to a programmer-specified method that selects an individual invocation. The `inni` statement services the invocation returned from this method.

We designed two approaches to Invocation Enumeration: View Enumeration and Named Enumeration. These variants differ in how they name invocations and invocation parameters. View Enumeration names each invocation's parameters as invocations are extracted from an enumeration. Named Enumeration names invocations as they are placed into an enumeration.

```
1   Invocation select_method(ArmEnumeration arm) { ... }
2    ...
3    inni with select_method over
4       void entryOne(int i, float f)          { ... }
5    [] void entryTwo(int i, float f, float g)  { ... }
```

Fig. 3. Invocation Enumeration: Specification of selection method.

Both approaches use a `with-over` clause to specify the selection method (line 3 of Figure 3). The method must have type signature $ArmEnumeration \rightarrow Invocation$. An `ArmEnumeration` enumerates `InvocationEnumerations`, each corresponding to the operation serviced by an arm of the `inni` statement. Each `InvocationEnumeration` is an enumeration of the respective operation's pending invocations. Figure 4 gives a pictorial example of the enumeration structure.

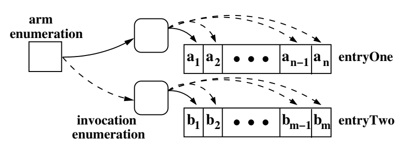

Fig. 4. Arm and Invocation Enumeration Structure.

View Enumeration An abridged implementation of the selection method used in Figure 3 is given in Figure 5. This example demonstrates accesses to individual invocations and their arguments. Line 3 shows the extraction of an `InvocationEnumeration` from the `ArmEnumeration`. An `Invocation` is extracted from an `InvocationEnumeration` on line 6. Under View Enumeration, the class `Invocation` is viewed as a variant type [5] of all invocation types and each invocation as a value of this variant type. A `view` statement is used to determine the underlying type of a specific invocation and to provide access to the arguments of the invocation. The `view` statement on lines 7–8 compares invocation `invoc` to the underlying invocation types $int \times float$ and $int \times float \times float$. The statement associated with the matching `as` clause is executed with the invocation's arguments bound to the respective parameters.

Named Enumeration Named Enumeration allows the programmer to "name" the invocation type used within an invocation enumeration. These types are specified via `as` clauses (Figure 6). Each "named" type (e.g., `nameOne` and `nameTwo`)

```
Invocation select_method(ArmEnumeration arm) {
   while (arm.hasMoreElements()) {
      InvocationEnumeration invoc_enum = arm.nextElement();
      if (invoc_enum == null) continue;
      while (invoc_enum.hasMoreElements()) {
         Invocation invoc = invoc_enum.nextElement();
         view invoc as (int i, float f)  { ... }  // block using i and f
         as (int i, float f, float g)    { ... }  // block using i, f, and g
      }
   }
}
```

Fig. 5. View Enumeration: Implementation of selection method.

is a class that extends the `Invocation` class and provides a constructor with type signature matching the operation being serviced. The "named" types are used within the selection method to access each invocation's parameters.

```
inni with select_method over
   void entryOne(int i, float f) as nameOne          { ... }
[] void entryTwo(int i, float f, float g) as nameTwo  { ... }
```

Fig. 6. Named Enumeration: Specification of selection method and invocation types.

3.2 Functional Reduction

The Functional Reduction approach splits invocation selection into two phases to provide a simple means for accessing the invocation parameters. Figure 7 gives a pictorial example of the two phases. The first phase (depicted by the horizontal arrows), which has direct access to the parameters of the invocations, selects a single invocation from each arm of the `inni` statement; this invocation is called the representative for its respective arm. The second phase (depicted by the vertical arrow), which can access invocation parameters through user-defined invocation types, selects the actual invocation to service from the representatives. Each phase selects an invocation through a reduction, using a programmer-specified method, over the appropriate set of invocations.

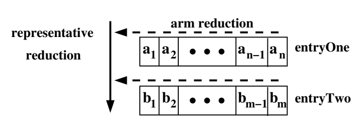

Fig. 7. Arm Reduction and Representative Reduction Structure.

4 Analysis of Mechanisms

Though the proposed mechanisms all satisfy the criteria for acceptable invocation selection mechanisms, they differ in terms of abstraction and performance.

This section discusses solutions to representative examples from three problem domains. In the interest of space, and since it is ultimately adopted, only the View Enumeration solutions are shown. The problem domains are:

– Selection Independent of Arguments: Does not require access to the arguments of the invocations. The representative, RANDOM, selects a random invocation from the pending invocations.
– Single Comparison Required: A comparison between two invocations is sufficient to eliminate one from consideration. The representative, PRIORITY, selects an invocation based on a "priority" argument.
– Multiple Comparisons Required: Simultaneous examination and comparison of multiple invocations is required. The representative, MEDIAN, selects the invocation with the median first argument of all pending invocations.

```
Invocation select_random(ArmEnumeration arm) {
  int num = 0;
  while (arm.hasMoreElements()) {  // tally invocations
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum != null)  num += invoc_enum.length();
  }
  int rand = RandomInvoc.rand.nextInt(num);
  arm.reset();  // return to beginning of enumeration
  while (arm.hasMoreElements()) {  // find invocation
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum != null) {
      if (rand >= invoc_enum.length)  rand -= invoc_enum.length;
      else {
        while (rand > 0) {  // find invocation in this "arm"
          invoc_enum.nextElement();   rand--;
        }
        return (Invoc) (invoc_enum.nextElement());
  } } }
  return null; // Shouldn't get here
}
```

Fig. 8. View Enumeration: RANDOM.

4.1 Selection Independent of Arguments

Invocation Enumeration Figure 8 gives the selection method used in both Invocation Enumeration solutions to RANDOM. The `select_random` method calculates the total number of pending invocations in the enumeration, generates a random number in the range, and, finally, returns the "random" invocation.

Functional Reduction A Functional Reduction solution to RANDOM first selects a representative invocation for each arm by randomly determining if the "current" invocation should be the representative based on the number of invocations examined thus far (i.e., the "previous" invocation carries a weight based on the number of invocations that it has beaten out). Finally, the invocation to service is selected from the representatives in the same manner.

4.2 Single Comparison Required

Invocation Enumeration Figure 9 gives the selection method used in a View Enumeration solution to PRIORITY. The algorithm used is very simple: loop through the invocations for each arm and record the invocation with the highest priority. A `view` statement (line 9) is used to access each invocation's arguments.

```
 1 Invocation prio_select(ArmEnumeration arm_enum) {
 2   Invocation cur = null;
 3   int best_prio = 0, cur_prio;
 4   while (arm_enum.hasMoreElements()) {
 5     InvocationEnumeration invoc_enum =arm_enum.nextElement();
 6     if (invoc_enum == null) continue;
 7     while (invoc_enum.hasMoreElements()) {
 8       Invocation invoc = invoc_enum.nextElement();
 9       view invoc as (int prio_tmp, int i)  //params of invoc
10         cur_prio = prio_tmp;
11       ... // for each invocation type
12       if ((cur == null) || (cur_prio < best_prio)) {
13         cur = invoc;   best_prio = cur_prio;
14   } } }
15   return cur;
16 }
```

Fig. 9. View Enumeration: PRIORITY.

Functional Reduction The Functional Reduction solution to PRIORITY compares the "current" invocation with the "previous" invocation and returns the one with highest priority. This general reduction is used to first select the representative invocations and then to select the invocation to service.

4.3 Multiple Comparisons Required

Invocation Enumeration Figure 10 outlines the selection method used in a View Enumeration solution to MEDIAN. The algorithm used gathers the invocations into a `Vector`, converts the `Vector` into an array, sorts the array, and, finally, selects the median invocation.

```
Invocation median_select(ArmEnumeration arm) {
  Vector v = new Vector();
  while (arm.hasMoreElements()) {
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum == null) continue;
    view invoc as (int firstarg, int i)  // params of invoc
        v.add(new Element(invoc, firstarg));
    ... // for each invocation type
  }
  ... // convert vector, sort, and return median invocation
}
```

Fig. 10. View Enumeration: MEDIAN.

Functional Reduction The Functional Reduction solution to MEDIAN highlights the drawbacks of the approach. The arm reduction method gathers each invocation into a `Vector` stored within a user-defined object. It is necessary to gather the invocations in this manner because the Functional Reduction approach does not provide direct simultaneous access to all of the invocations. With the invocations for each arm gathered into objects, a reduction over the arms selects the final invocation to service. Unfortunately, since a specific call of the reduction method cannot determine if it is the last, each call of the reduction method must select the "final" invocation to service (i.e., the median invocation of those examined thus far). As such, each call of the representative reduction method performs the costly median invocation selection.

4.4 Performance

A number of experiments were run to evaluate the performance of each of the proposed mechanisms. The experiments were conducted on a 850 MHz Intel Pentium III with 128 MB of RAM running Linux kernel 2.2.19 and IBM's JRE 1.3.0. The experiments apply the solutions to an `inni` statement servicing 10 distinct operations with pending invocations initially evenly distributed. The results report, for each solution, the time to select and service all pending invocations. (Other experiment sizes were also run with similar results.)

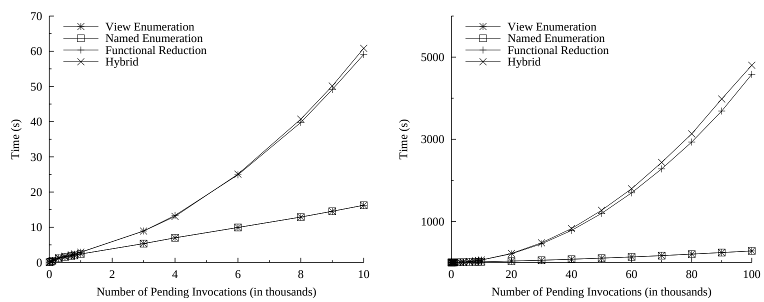

Fig. 11. RANDOM: Time to service all pending invocations.

RANDOM Figure 11 plots the number of initially pending invocations versus time for each solution to RANDOM. In this experiment, Functional Reduction does not compare favorably to Invocation Enumeration. It must be noted, however, that the Invocation Enumeration solution to RANDOM implements a more efficient algorithm than does Functional Reduction. The Invocation Enumeration solutions use a divide-and-conquer approach to examine a fraction of the pending invocations. An operation's entire set of pending invocations is skipped if the random invocation is not a member of the set (see lines 12-16 in Figure 8). The same divide-and-conquer algorithm cannot be implemented using the Functional Reduction approach because the reduction examines every invocation.

PRIORITY Figure 12 plots the number of initially pending invocations versus time for each solution to PRIORITY. This problem requires access to invocation arguments, which is the reason that the Named Enumeration solution performs poorly. To support the abstraction provided by Named Enumeration, an object (of the programmer "named" type) must be created for each pending invocation. This accounts for the performance difference between View and Named Enumeration.

MEDIAN Figure 13 plots the number of initially pending invocations versus time for each solution to MEDIAN. Functional Reduction performs poorly because the expensive selection computation (selecting the median element) is executed for each call of the arm reduction method (equal to the number of arms).

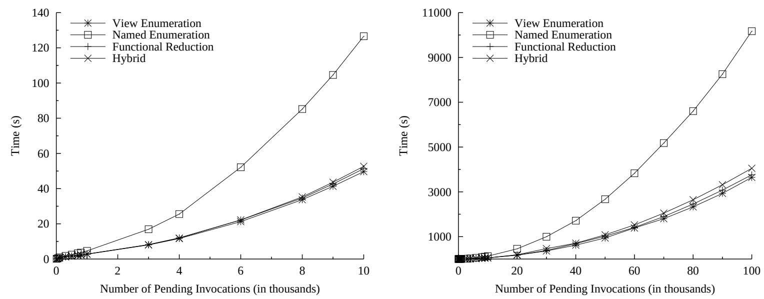

Fig. 12. PRIORITY: Time to service all pending invocations.

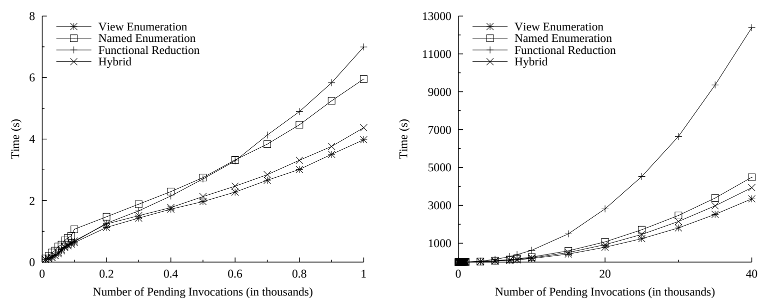

Fig. 13. MEDIAN: Time to service all pending invocations.

5 Discussion

View Enumeration performs the best. The `view` statement incurs little overhead over direct access to the underlying operation implementation, while providing sufficient abstraction of that implementation. Unfortunately, the structure of a `view` statement must closely match that of the associated `inni` statement. This structural dependence potentially limits selection method reuse. Named Enumeration reduces this dependence, but incurs object creation overhead.

Functional Reduction is elegant, but suffers greatly in terms of performance. This strategy's performance penalties can be categorized into method call overhead, repeated selection, and state maintenance. A method call overhead is incurred for each invocation, even when the invocation to service has already been found. In such a case, the number of method calls could be minimized if it were possible to abort reduction. Repeated selection is the repeated execution of selection code to satisfy an invariant (as in MEDIAN). A predicate indicating the last method call of a reduction could reduce the repeated selection penalty. State maintenance is the need to explicitly carry extra state through a reduction (as in MEDIAN). This, unfortunately, makes it necessary to create extra objects.

6 Conclusion

This paper discussed different candidate selection mechanisms considered for addition to the JR programming language. For consideration, a candidate had to satisfy criteria requiring selection of any invocation from the set of pending invocations, access to each invocation's actual arguments, and the prevention of selection side-effects. We have extended the JR programming language with support for the View Enumeration approach because of its high-level of abstraction and performance. Because View Enumeration does not rely on subclassing, this approach can also be used in concurrent languages that are not object-oriented.

References

1. G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.
2. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
3. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
4. A. Burns. *Programming in Occam*. Addison Wesley, 1988.
5. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
6. M. Chung and R. A. Olsson. New mechanisms for invocation handling in concurrent programming languages. *Computer Languages*, 24:254–270, December 1998.
7. S. Dorward and R. Pike. Programming in Limbo. In *Proceedings of the IEEE Compcon 97 Conference*, pages 245–250, 1997.
8. N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.
9. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
10. Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/Ada- IC/standards/95lrm_rat`.
11. A. W. Keen. *Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study*. PhD dissertation, University of California, Davis, Department of Computer Science, June 2002.

12. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 575–584, April 2001.
13. University of Kent. Communicating sequential processes for java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.
14. University of Kent. Kent retargetable occam compiler. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
15. R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
16. C.A. Waldspurger and W.E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management". In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, Monterey, CA, November 1994.