Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques

Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha

Department of Computer Science
University of Illinois, Urbana, IL 61801, USA
{slauter2, rkumar8, marinov, agha}@illinois.edu

Abstract. Actor programs consist of a number of concurrent objects called actors, which communicate by exchanging messages. Nondeterminism in actors results from the different possible orders in which available messages are processed. Systematic testing of actor programs explores various feasible message processing schedules. Dynamic partial-order reduction (DPOR) techniques speed up systematic testing by pruning parts of the exploration space. Based on the exploration of a schedule, a DPOR algorithm may find that it need not explore some other schedules. However, the potential pruning that can be achieved using DPOR is highly dependent on the order in which messages are considered for processing. This paper evaluates a number of heuristics for choosing the order in which messages are explored for actor programs, and summarizes their advantages and disadvantages.

1 Introduction

Modern software has several competing requirements. On one hand, software has to execute efficiently in a networked world, which requires concurrent programming. On the other hand, software has to be reliable and dependable, since software bugs could lead to great financial losses and even loss of lives. However, putting together these two requirements—building concurrent software while ensuring that it be reliable and dependable—is a great challenge. Approaches that help address this challenge are in great need.

Actors offer a programming model for concurrent computing based on message passing and object-style data encapsulation [1,2]. An actor program consists of several computation entities, called actors, each of which has its own thread of control, manages its own internal state, and communicates with other actors by exchanging messages. Actor-oriented programming systems are increasingly used for concurrent programming, and some practical actor systems include ActorFoundry, Asynchronous Agents Framework, Axum, Charm++, Erlang, E, Jetlang, Jsasb, Kilim, Newspeak, Ptolemy II, Revactor, SALSA, Scala, Singularity, and ThAL. (For a list of references, see [16].)

A key challenge in testing actor programs is their inherent nondeterminism: even for the same input, an actor program may produce different results based on the *schedule* of arrival of messages. Systematic exploration of possible message

arrival schedules is required both for testing and for model checking concurrent programs [3–5, 7, 10–12, 18, 21, 22]. However, the large number of possible message schedules often limits how many schedules can be explored in practice. Fortunately, such exploration need not enumerate all possible schedules to check the results. *Partial-order reduction (POR)* techniques speed up exploration by pruning some message schedules that are equivalent [7,10,12–14,18,22]. *Dynamic partial-order reduction (DPOR)* techniques [10, 18, 19] discover the equivalence dynamically, during the exploration of the program, rather than statically, by analyzing the program code. The actual dynamic executions provide more precise information than a static analysis that needs to soundly over-approximate a set of feasible executions. Effectively, based on the exploration of some message schedules, a DPOR technique may find that it need not explore some other schedules.

It turns out that pruning using DPOR techniques is highly sensitive to the order in which messages are considered for exploration. For example, consider a program which reaches a state where two messages, m_1 and m_2, can be delivered to some actors. If a DPOR technique first explores the possible schedules after delivering m_1, it could find that it need not explore the schedules that first deliver m_2. But, if the same DPOR technique first delivers m_2, it could happen that it cannot prune the schedules from m_1 and thus needs to perform the entire exhaustive exploration. We recently observed this sensitivity in our work on testing actor programs [16], and Godefroid mentioned it years ago [12]. Dwyer et al. [8] evaluate the search order for different exploration techniques. However, we are not aware of any prior attempt to analyze what sorts of message orders lead to better pruning for DPOR.

This paper addresses the following questions:

- What are some of the natural *heuristics* for ordering scheduling decisions in DPOR for message-passing systems?
- What is the impact of choosing one heuristic over another heuristic?
- Does the impact of these heuristics depend on the DPOR technique?
- Can we predict which heuristic may work better for a particular DPOR technique or subject program?

The paper makes two contributions. First, it presents eight ordering heuristics (Sect. 5) and evaluates them on seven subject programs (Sect. 6). We compare the heuristics for two DPOR techniques: one based on dynamically computing persistent sets [10, 12] and the other based on dCUTE [18] (Sect. 2). As our evaluation platform, we use the Basset system [16]. The results show that different heuristics can lead to significant differences in pruning, up to two orders of magnitude. Second, the paper summarizes the advantages and disadvantages of various heuristics. In particular, it points out what types of programs, based on the communication pattern of the actors, may benefit the most from which heuristics. This provides important *guidelines* for exploring actor programs in practice: based on the type of the program, the user can instruct an exploration tool to use a heuristic that provides better pruning, resulting in a faster exploration and more efficient bug finding.

2 Actor Language and Execution Semantics

For illustrative purposes, we describe an imperative actor language *ActorFoundry* that is implemented as a Java framework [15]. A class that describes an actor behavior extends `osl.manager.Actor`. An actor may have local state comprised of primitives and objects. The local state cannot be shared among actors. An actor can communicate with another actor in the program by sending asynchronous messages using the library method `send`. The sending actor does not wait for the message to arrive at the destination and be processed. The library method `call` sends an asynchronous message to an actor but blocks the sender until the message arrives and is processed at the receiver. An actor definition includes method definitions that correspond to messages that the actor can accept and these methods are annotated with `@message`. Both `send` and `call` can take arbitrary number of arguments that correspond to the arguments of the corresponding method in the destination actor class. The library method `create` creates an actor instance of the specified actor class. It can take arbitrary number of arguments that correspond to the arguments of the constructor. Message parameters and return types should be of the type `java.io.Serializable`. The library method `destroy` kills the actor calling the method. Messages sent to the killed actor are never delivered. Note that both `call` and `create` may throw a checked exception `RemoteCodeException`.

We informally present semantics of relevant ActorFoundry constructs to be able to more precisely describe the algorithms in Sect. 3. Consider an ActorFoundry program P consisting of a set of actor definitions including a *main* actor definition that receives the initial message. `send`(a, msg) appends the contents of the message msg to the message queue of actor a. We will use Q_a to denote the message queue of actor a. We assume that at the beginning of execution the message queue of all actors is empty.

The ActorFoundry runtime first creates an instance of the main actor and then sends the initial message to it. Each actor executes the following steps in a loop: remove a message from the queue (termed as an *implicit receive* statement from here on), decode the message, and process the message by executing the corresponding method. During the processing, an actor may update the local state, create new actors, and send more messages. An actor may also throw an exception. If its message queue is empty, the actor blocks waiting for the next message to arrive. Otherwise, the actor *nondeterministically* removes a message from its message queue. The nondeterminism in choosing the message models the asynchrony associated with message passing in actors. An actor executing a `create` statement produces a new instance of an actor.

An actor is said to be *alive* if it has not already executed a `destroy` statement or thrown an exception. An actor is said to be *enabled* if the following two conditions hold: the actor is alive, and the actor is not blocked due to an empty message queue or executing a `call` statement.

A variable pc_a represents the program counter of the actor a. For every actor, pc_a is initialized to the implicit receive statement. A scheduler executes a loop inside which it *nondeterministically* chooses an enabled actor a from the set

$\mathcal{P}$. It executes the next statement of the actor a, where the next statement is obtained by calling $statement\_at(pc_a)$. During the execution of the statement, the program counter pc_a of the actor a is modified based on the various control flow statements; by default, it is incremented by one.

The concrete execution of an internal statement, i.e., a statement not of the form `send`, `call`, `create`, or `destroy`, takes place in the usual way for imperative statements. The loop of the scheduler terminates when there is no enabled actor in $\mathcal{P}$. The termination of the scheduler indicates either the normal termination of a program execution, or a deadlock state (when at least one actor in $\mathcal{P}$ is waiting for a `call` to return).

3 Automated Testing of ActorFoundry Programs

To automatically test an ActorFoundry program for a given input, we need to explore all *distinct*, feasible execution paths of the program. A path is intuitively a sequence of statements executed, or as we will see later, it suffices to have just a sequence of messages received. In this work, we assume that the program always terminates and a test harness is available, and thus focus on exploring the paths for a given input. A simple, systematic exploration of an ActorFoundry program can be performed using a *naïve* scheduler: beginning with the initial program state, the scheduler nondeterministically picks an enabled actor and executes the next statement of the actor. If the next statement is implicit receive, the scheduler nondeterministically picks a message for the actor from its message queue. The scheduler records the ids of the actor and the message, if applicable. The scheduler continues to explore a path in the program by making these choices at each step. After completing execution of a path (i.e., when there are no new messages to be delivered), the scheduler backtracks to the last scheduling step (in a depth-first strategy) and explores alternate paths by picking a different enabled actor or a different message from the ones chosen previously.

Note that the number of paths explored by the naïve scheduler is exponential in the number of enabled actors and the number of program statements in all enabled actors. However, an exponential number of these schedules is equivalent. A crucial observation is that actors do not share state: they exchange data and synchronize only through messages. Therefore, it is sufficient to explore paths where actors interleave at message receive points only. All statements of an actor between two implicit receive statements can be executed in a single *atomic* step called a *macro-step* [2,18]. At each step, the scheduler picks an enabled actor and a message from the actor's message queue. The scheduler records the ids of the actor and the message, and executes the program statements as a macro-step. A sequence of macro-steps, each identified by an actor and message pair (a, m), is termed a *macro-step schedule*. At the end of a path, the scheduler backtracks to the last macro-step and explores an alternate path by choosing a different pair of actor and message (a, m).

Note that the number of paths explored using a macro-step scheduler is exponential in the number of deliverable messages. This is because the scheduler,

```
scheduler(P)
  pc_{a_1} = l_0^{a_1}; pc_{a_2} = l_0^{a_2}; ...; pc_{a_n} = l_0^{a_n};
  Q_{a_1} = [ ]; Q_{a_2} = [ ]; ...; Q_{a_n} = [ ];
  i = 0;
  while (∃a ∈ P such that a is enabled)
    (a,msg_id) = next(P);
    i = i + 1;
    s = statement_at(pc_a);
    execute(a, s, msg_id);
    s = statement_at(pc_a);
    while (a is alive and s ≠ receive(v))
      if s is send(b, v)
        for all k ≤ i
            such that b == path_c[k].receiver
            and canSynchronize(path_c[k].s, s)
          // actor a' "causes" s
          path_c[k].S_p.add((a', _));
      execute(a, s, msg_id);
      s = statement_at(pc_a);
  compute_next_schedule();
```

```
compute_next_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].S_p is not empty
      path_c[j].schedule =
        path_c[j].S_p.remove();
      path_c = path_c[0 ... j];
      return;
    j = j − 1;
  if (j < 0) completed = true;

next(P)
  if (i ≤ |path_c|)
    (a,msg_id) = path_c[i].schedule;
  else
    (a,msg_id) = choose(P);
    path_c[i].schedule = (a,msg_id);
    path_c[i].S_p.add((a, _));
  return (a,msg_id);
```

Fig. 1. Dynamic partial-order reduction algorithm based on persistent sets.

for every step, executes all permutations of actor and message pairs (a, m) that are enabled before the step. However, messages sent to different actors may be independent of each other, and it may be sufficient to explore all permutations of messages for a *single* actor instead of all permutations of messages for *all* actors [18].

The independence between certain events results in equivalent paths, in which different orders of independent events occur. The equivalence relation between paths is exploited by dynamic partial-order reduction (DPOR) algorithms to speed-up automatic testing of actor programs by pruning parts of the exploration space. Specifically, the equivalence is captured using the *happens-before relation* [9, 18], which yields a partial order on the state transitions in the program. The goal of DPOR algorithms is to explore only one linearization of each partial order or equivalence class.

We next describe two stateless DPOR algorithms for actor programs: one based on dynamically computing persistent sets [10] (adapted for testing actor programs), and the other one used in dCUTE [18].

DPOR based on Persistent Sets

Flanagan and Godefroid [10] introduced a DPOR algorithm that dynamically tracks dependent transitions and computes persistent sets [12] among concurrent processes. They presented the algorithm in the context of shared-memory programs. Figure 1 shows our adaptation of their algorithm for actor programs, which also incorporates the optimization discussed by Yang et al. [23].

The algorithm computes persistent sets in the following way: during the initial run of the program, for every scheduling point, the scheduler *nondeterministically* picks an enabled actor (call to the *choose* method, which is underlined) and adds all its pending messages to the persistent set S_p. It then explores all

```
scheduler(P)
  pc_{a_1} = l_0^{a_1}; pc_{a_2} = l_0^{a_2}; ...; pc_{a_n} = l_0^{a_n};
  Q_{a_1} = [ ]; Q_{a_2} = [ ]; ...; Q_{a_n} = [ ];
  i = 0;
  while (∃a ∈ P such that a is enabled)
    (a,msg_id) = next(P);
    i = i + 1;
    s = statement_at(pc_a);
    execute(a, s, msg_id);
    s = statement_at(pc_a);
    while (a is alive and s ≠ receive(v))
      if s is send(b, v)
        for all k ≤ i
            such that b == path_c[k].receiver
            and canSynchronize(path_c[k].s, s)
          path_c[k].needs_delay = true;
      execute(a, s, msg_id);
      s = statement_at(pc_a);
  compute_next_schedule();
```

```
compute_next_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].next_schedule ≠ (⊥, ⊥)
      (a, m) =path_c[j].schedule;
      (b, m') =path_c[j].next_schedule;
      if a == b or path_c[j].needs_delay
        path_c[j].schedule =
          path_c[j].next_schedule;
        if a ≠ b
          path_c[j].needs_delay = false;
        path_c = path_c[0 ... j];
        return;
    j = j − 1;
  if (j < 0) completed = true;

next(P)
  if (i ≤ |path_c|)
    (a,msg_id) = path_c[i].schedule;
  else
    (a,msg_id) = choose(P);
    path_c[i].schedule = (a,msg_id);
  path_c[i].next_schedule = next(a,msg_id);
  return (a,msg_id);
```

Fig. 2. Dynamic partial-order reduction algorithm for the dCUTE approach.

permutations of messages in the persistent set. During the exploration, if the scheduler encounters a $\texttt{send}(a, v)$ statement, say at position i in the current schedule, it analyzes all the receive statements executed by a earlier in the same execution path (represented as *path_c*). If a receive, say at position $k < i$ in the schedule, is not related to the send statement by the happens-before relation (checked in the call to method *canSynchronize*), the scheduler adds pending messages for a new actor a' to the persistent set at position k. The actor a' is "responsible" for the send statement at i, i.e., a receive for a' is enabled at k, and it is related to the send statement by the happens-before relation.

DPOR in dCUTE

Figure 2 shows the DPOR algorithm that is a part of the dCUTE approach for testing open, distributed systems [18]. (Since we do not consider open systems here, we ignore the input generation from dCUTE.) It proceeds in the following way: during the initial run of the program, for every scheduling point, the scheduler *nondeterministically* picks an enabled actor (call to the *choose* method, which is underlined) and explores permutations of messages enabled for the actor. During the exploration, if the scheduler encounters a send statement of the form $\texttt{send}(a, v)$, it analyzes all the receive statements seen so far in the same path. If a receive statement is executed by a, and the send statement is not related to the receive in the happens-before relation, the scheduler sets a flag at the point of the receive statement. The flag indicates that all permutations of messages to some other actor a' (different from a) need to be explored at the particular point. The exploration proceeds in a nondeterministic fashion again from there on. A more detailed discussion of the algorithm can be found in [18].

Note that the algorithms discussed above re-execute the program from the beginning with the initial state in order to explore a new program path. The algorithms can be easily modified to support checkpointing and restoration of intermediate states, since these operations do not change DPOR fundamentally.

4 Illustrative Example

To illustrate key DPOR concepts and how different message orderings can affect the exploration of actor programs, we use a simple example actor program that computes the value of π. It is a porting of a publicly available [17] MPI example, which computes an approximation of π by distributing the task among a set of worker actors.

Figure 3 shows a simplified version of this code in ActorFoundry. The `Driver` actor creates a *master* actor that uses a given number of *worker* actors to carry out the computation. The `Driver` actor sends a start message to the master actor which in turn sends messages to each worker, collects partial results from them, reduces the partial results, and after all results are received, instructs the workers to terminate and terminates itself.

Figure 4 shows the search space for this program with master actor M and two worker actors A and B. Each state in the figure contains a set of messages. A message is denoted as X_Y where X is the actor name and Y uniquely identifies the message to X. We assume that the actors are created in this order: A, B, M. Transitions are indicated by arrows labeled with the message that is received, where a transition consists of the delivery of a message up to the next delivery.

The *boxed* states indicate those states that will be visited when the search space is explored using a DPOR technique, and when actors are chosen for exploration according to *the order in which the receiving actors are created*. Namely, the search will favor exploration of messages to be delivered to A over those to be delivered to B or M, so if in some state (say, the point labeled K) messages can be delivered to both A and B, the search will first explore the delivery to A and only after that the delivery to B. To illustrate how this ordering affects how DPOR prunes execution paths, consider the state at point G. For this state, the algorithm will first choose to deliver the message B_1. While exploring the search space that follows from this choice, all subsequent sends to actor B are causally dependent on the receipt of message B_1. This means that DPOR does not need to consider delivering the message M_A before B_1. This allows pruning the two paths that delivering M_A first would require. Similar reasoning shows that DPOR does not need to consider delivering B_2 before A_2 at points S and T, and that it does not need to consider delivering B_1 at point K. In total, this ordering prunes *10 of 12* paths, i.e., with this ordering, only 2 of 12 paths are explored.

The *shaded* states indicate those states that will be visited when the search space is explored using the same DPOR, but when actors are chosen for exploration according to *the reverse-order in which the receiving actors are created*. This means that the search will favor exploration of messages to be delivered

```
class Master extends Actor {
   ActorName[] workers;
   int counter = 0;
   double result = 0.0;
   public Master(int N) {
      workers = new ActorName[N]
      for (int i = 0; i < N; i++)
         workers[i] =
            create(Worker.class, i, N);
   }
   @message void start() {
      int n = 1000;
      for (ActorName w: workers)
         send(w,"intervals", self(), n);
   }
   @message void sum(double p) {
      counter++;
      result += p;
      if (counter == workers.length) {
         for (ActorName w: workers)
            send(w,"stop");
         destroy("done");
      }
   }
}
```

```
class Worker extends Actor {
   int id;
   int nbWorkers;
   public Worker(int id, int nb) {
      this.id = id;
      this.nbWorkers = nb;
   }
   @message void intervals(ActorName master, int n) {
      double h = 1.0 / n; double sum = 0;
      for (int i = id; i <= n; i += nbWorkers) {
         double x = h * (i - 0.5);
         sum += (4.0 / (1.0 + x*x));
      }
      send(master, "sum", h * sum);
   }
   @message void stop() {destroy("done");}
}

class Driver extends Actor {
   static void main(String[] args) {
      ActorName master =
         create(Master.class, args[0]);
      send(master, "start");
   }
}
```

Fig. 3. ActorFoundry code for the pi example.

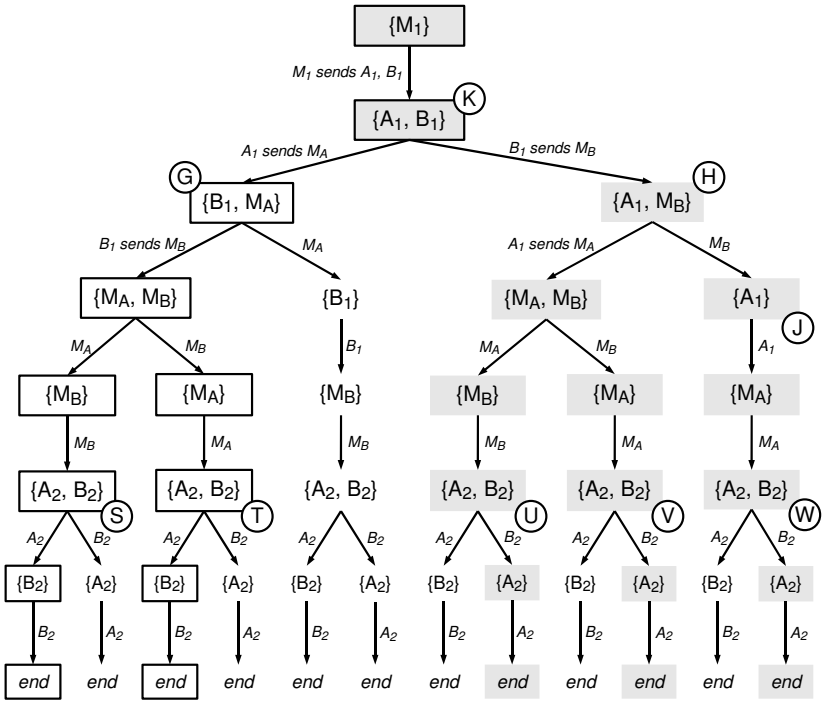

Fig. 4. State space for the pi example with two worker actors.

to M over those to be delivered to B or A. This reverse-ordering causes DPOR to prune execution paths differently. Consider the state at point H. For this state, the algorithm will first choose to deliver the message M_B. Following this path, it comes to point J, where the delivery of message A_1 results in message M_A being sent. This send to actor M is *not* causally dependent on the receipt of message M_B. This means that the DPOR also needs to consider delivering the message A_1 before M_B at point H. As the search continues, it discovers that it does not need to consider delivering A_2 before B_2 at points U, V, and W; and also it does not need to consider delivering A_1 at point K. In total, the reverse-ordering prunes *9 of 12* paths, which is one fewer than when the messages are selected in the order in which the receiving actors are created. As shown in Sect. 6, this difference in the number of paths pruned increases as the number of worker actors increases.

5 Heuristics

The example in Sect. 4 illustrates the idea that scheduling decisions may affect the efficiency of DPOR techniques. In the algorithms presented in Sect. 3, the scheduling choices are represented by the calls to the *choose* method (underlined). Observe that these DPOR algorithms first collect all possible messages for an actor at a given state, and then explore some orders for processing this set of messages. The key question, therefore, is how to order these messages for a given state.

We present eight possible heuristics for ordering messages:

1. *Earliest created actor (ECA)* sorts the enabled actors by their creation time in the *ascending* order. The intuition is to capture the "asymmetry" between some actors in terms of the communication pattern.
2. *Latest created actor (LCA)* is similar to ECA but sorts the enabled actors by their creation time in the *descending* order.
3. *Queue (FIFO)* sorts the actors based on the time of the *earliest* message sent to them, in the *ascending* order. This heuristic captures the common implementation order of choosing messages from a scheduling queue.
4. *Stack (LIFO)* sorts the actors based on the time of the *last* message sent to them, in the *descending* order.
5. *Lowest number of deliverable messages (LDM)* sorts the actors by the number of messages in their respective message queue, in the *ascending* order. The intuition is that the actors that have received more messages are more likely to receive more messages later in the computation.
6. *Highest number of deliverable messages (HDM)* sorts the actors by the number of messages in their respective message queue, in the *descending* order.
7. *Highest average messages sent (HMS)* prioritizes the actors which have been sending the highest number of messages per received message, based on the exploration history. The intuition is that the actors that have been sending more messages in the past are more likely to send more messages in the future.

8. *Send graph reachability (SGR)* is based on information collected during prior executions. Specifically, it maintains a directed graph where nodes represent actors and edges indicate that a message was sent from the first node to the second at some point in the exploration. Now, consider two messages: one to actor A, and one to actor B. If actor B is reachable in the graph from actor A and no such path exists from actor B to actor A, then SGR will prioritize actor A over actor B. The intuition is that actor B is less likely to cause a message to be sent to actor A.

These eight heuristics capture some intuition based on the functioning of the DPOR algorithms and on the patterns of communication in actor programs. While our list of heuristics is not complete by any means, we believe that it is sufficiently representative to help us answer the questions raised by our study (Sect. 1).

6 Evaluation

To evaluate the different heuristics for dynamic partial-order reduction, we conducted experiments using two different DPOR techniques. The heuristics and DPOR techniques are implemented in the Basset framework [16]. Basset provides an extensible environment for exploration of Java-based actor programs. It is built on top of Java PathFinder (JPF), a popular explicit state model checker for Java bytecode [20].

We first describe the subject programs used to quantitatively evaluate the heuristics. We then present experimental results comparing the different heuristics for the two DPOR techniques. All experiments are performed using Sun's JVM 1.6.0_16-b01 on a 2.80GHz Intel Core2 Duo running Ubuntu release 9.04.

6.1 Subject Programs

Our experiments use the seven actor programs listed in Table 1. All of these subjects are either originally written using the ActorFoundry library [1, 2] or ported to that environment.

The `pi` subject is the example described in Sect. 4. However, the results shown here are for a configuration using five worker actors. Two of the subjects implement more complex algorithms previously used in the dCUTE study [18]: `leader` is an implementation of a leader election algorithm; and `shortpath` is an implementation of the Chandy-Misra's shortest path algorithm [6]. The `shortpath` subject appears twice in the results: once for a graph with 4 nodes (`shortpathA`), and again for a graph with 5 nodes (`shortpathB`). Note that the two graphs are dissimilar. The `fibonacci` subject computes the n-th element in the Fibonacci sequence. `quicksort` is an implementation of a distributed sorting algorithm that use a standard divide-and-conquer strategy to carry out the computation. `pipesort` is a modified version of the sorting algorithm used in the dCUTE study [18]. `chameneos` is an implementation of the `chameneos-redux` benchmark from the Great Language Shootout (`http://shootout.alioth.debian.org`).

Table 1. Comparison of different ordering heuristics. (The best result is bold.)

		dCUTE		Persistent			dCUTE		Persistent	
Heur.	Subject	# of Paths	time [sec]	# of Paths	time [sec]	Subject	# of Paths	time [sec]	# of Paths	time [sec]
ECA		3821	300	19683	1474		**288**	**22**	**288**	**23**
LCA		**216**	**19**	**216**	**20**		5970	441	5970	453
FIFO		972	75	3240	267		1794	128	1791	138
LIFO	chameneos	2031	142	4899	320	pipesort	1080	77	1080	78
LDM		753	67	3375	279	size=4	384	33	384	32
HDM		3821	312	19683	1626		2072	154	1480	126
HMS		3691	301	19683	1639		307	25	307	26
SGR		3821	280	19683	1422		**288**	**24**	**288**	**24**
ECA		684	65	327	31		7038	514	3822	327
LCA		**16**	**5**	**16**	**5**		**32**	**6**	**32**	**6**
FIFO		68	9	40	7		572	48	368	31
LIFO	fib(5)	81	12	81	13	quicksort	243	26	243	25
LDM		508	51	261	28	size=6	6390	512	2502	206
HDM		526	59	263	31		5118	424	2804	250
HMS		82	12	66	10		195	21	183	21
SGR		684	70	327	34		7038	514	3822	325
ECA		101	9	101	9		516	32	392	25
LCA		188	16	188	15		680	43	640	33
FIFO		122	12	119	12		**360**	**24**	**238**	**18**
LIFO	leader	125	11	125	11	shortpath	859	48	750	36
LDM		133	12	133	12	graph A	585	42	492	33
HDM		**88**	**9**	**88**	**9**		562	39	419	30
HMS		141	14	126	12		540	35	453	32
SGR		101	9	101	10		516	33	392	25
ECA		**120**	**25**	**120**	**22**		7216	397	2658	127
LCA		945	142	19845	2921		7462	570	1865	109
FIFO		**120**	**22**	**120**	**22**		**3488**	**244**	**528**	**41**
LIFO	pi	945	149	19845	2833	shortpath	6472	489	2638	167
LDM	5 workers	**120**	**23**	**120**	**24**	graph B	7326	509	1178	71
HDM		706	120	3424	614		13438	1111	2756	273
HMS		945	179	19845	3542		3618	268	783	44
SGR		153	29	567	77		7940	493	3349	186

6.2 Results and Observations

Table 1 shows the results of experiments comparing the different heuristics for both the DPOR based on persistent sets and the one used for dCUTE. For each heuristic, we tabulate the total number of paths executed and the total exploration time in seconds. The results suggest that the efficiency of the two DPOR techniques is greatly dependent on the order in which messages are selected for exploration.

Recall the four research questions posed in Sect. 1. The first question is discussed in Sect. 5 where we describe some intuitive ordering heuristics to guide DPOR algorithms. We address the remaining three questions now by making observations on the results in Table 1.

1. What is the impact of choosing one heuristic over another heuristic?

The table shows that for 6 out of 8 experiments, one of the heuristics (but not necessarily the same) performs the best, i.e., there is no tie for the best performing heuristic. In the case of `pipesort` the tie between ECA and SGR is due to the relationship between the two heuristics. Specifically, ECA is the tie-breaking heuristic for SGR.

SGR performs the same as ECA for 6 out of 8 experiments. However, for the remaining two experiments, SGR performs worse than ECA. This suggests that the SGR heuristic, despite its usage of additional information, does not offer any advantage over ECA.

We also observe that the difference between the best and the worst heuristic can be very large. For example, for the `quicksort` subject sorting an array of size 6 and dCUTE DPOR, the best heuristic (LCA) has 2 orders of magnitude (more precisely, 220X) fewer executions than the worst performing heuristic (ECA). Note that both these heuristics are natural orders on the scheduling queue. In fact, the dCUTE DPOR algorithm as originally presented [18] employs the ECA ordering. The second best performing heuristic (HMS) for `quicksort` still explores 6 times as many executions as the best heuristic. For the other subjects, the ratio between the number of executions in the worst and the best case ranges from 2X (for `leader`) to 91X (for `chameneos`).

In general, the exploration time strongly correlates with the number of executed paths. This observation suggests that the better heuristics do not have a significant computation cost, and thus their reduction in the number of executions directly translates into savings in the exploration time. There are exceptions: for the subject `shortpathB`, the exploration time does not correlate with the number of paths executed as closely as other experiments. We believe that this is due to our experiments using Basset which is built on top of JPF and uses checkpointing and restoring to explore different paths, rather than re-execution. Hence, the time may relate more to the number of states visited instead of the number of executions, or stated differently, the time may depend more strongly on the length of executions instead of the number of executions.

2. Does the impact of these heuristics depend on the DPOR technique?

Although the results differ between the two DPOR algorithms for the experiments, the results exhibit a *similar ranking* of heuristics for both algorithms. In other words, for a given subject, heuristics that perform well for one DPOR technique tend to perform well for the other. Similarly, a heuristic that performs poorly typically does so for both DPOR algorithms.

It is evident from the table that for all 8 experiments, the best heuristic exactly matches for both DPOR algorithms. Moreover, even the worst heuristic matches for 7 out of 8 experiments.

3. Can we predict which heuristic may work better for a particular DPOR technique or subject program?

We found that which heuristic performs the best relates to the communication patterns employed by the program. For example, in a *pipelined computation*, it is more efficient to schedule first the actors that represent the early stages in the pipeline. On the other hand, in a *divide-and-conquer tree*, it is more efficient to schedule child actors before the parent actor.

Indeed, the ECA heuristic is the best performing heuristic for `pipesort`. ECA prioritizes actors in the early stages of a pipeline, and this enables the DPOR algorithms to collect all possible messages for actors in the later stages of the pipeline.

For 3 out of 8 subjects, the LCA heuristic performs the best among all heuristics. Two of these subjects—`fib` and `quicksort`—employ a divide-and-conquer approach. The remaining subject, `chameneos`, has a request-reply pattern between a broker and many clients. LCA allows the DPOR algorithm to collect all possible messages sent from the clients to the broker before exploring all the permutations of this set of messages.

For subjects with arbitrary graphs and communication patterns, the FIFO heuristic outperforms the remaining heuristics. For instance, the input graphs for `shortpathA` and `shortpathB` are dissimilar, and the effectiveness of several heuristics varied between the two experiments. Yet, the FIFO heuristic is the most effective heuristics for both inputs.

We performed some additional experiments for `shortpath` (not shown in the table) to identify how much the choice of heuristic depends on the program *input* rather than program *code*. In particular, the input to `shortpath` is a graph, and the messages exchanged depend on the topology of this graph. We considered seven more graphs (all with four or five nodes) in addition to the two for which the results are shown. While there is some variation of the results, in all the cases, FIFO is the best heuristic, either by itself, or together with some other heuristics (e.g., for a graph that is a list, there is only one execution path for any heuristic). These results are not conclusive, but they strongly suggest that the choice of heuristic depends on the program (and its communication pattern) more than on the input. Ideally, we would like to evaluate how `shortpath` performs for all graphs of a given size (but some explorations time out after an hour even for graphs of size just four). We would also like to evaluate sensitivity of heuristics to the inputs for other programs. We leave that as future work.

In summary, the results suggest the following set of guidelines for selecting a heuristic before the exploration of a program. (1) If there is no well-defined topology and communication pattern in the program (or if this communication pattern is not known a priori), then the default heuristic should be FIFO, since it is never the worst and sometimes is even the best heuristic. (2) If the communi-

cation pattern is a pipeline, then ECA should be used. (3) If the communication pattern is a divide-and-conquer tree, then LCA should be used.

7 Conclusions

Systematic exploration of message schedules is a viable approach to address the important but challenging problem of testing actor programs. Dynamic partial-order reduction (DPOR) techniques can significantly speed up systematic exploration, but they are highly sensitive to the order in which messages are explored. We described and compared several heuristics that can be used for ordering messages. Our results show up to two orders of magnitude difference in the number of executions explored. Moreover, our analysis of the results discovered guidelines that, based on the type of program, can aid selection of a good heuristic before the exploration. There has been recent work on combining DPOR techniques with stateful exploration [24,25], and we plan to evaluate the effectiveness of heuristics for such approaches. Similarly, we plan to evaluate the impact of heuristics on DPOR algorithms based on sleep sets [12].

Acknowledgments. The authors would like to thank Mirco Dotta, Stoyan Gaydarov, and Bobak Hadidi for their help in preparing evaluation subjects, and Samira Tasharofi for discussions and other assistance during the course of this project. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0916893, CNS-0851957, CCF-0746856, and CNS-0509321.

References

1. Agha, G.: Actors: A model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–72 (1997)
3. Artho, C., Garoche, P.L.: Accurate centralization for applying model checking on networked applications. In: 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006. pp. 177–188. IEEE Comp. Society (2006)
4. Arts, T., Earle, C.B.: Development of a verified Erlang program for resource locking. In: Formal Methods in Industrial Critical Systems (2001)
5. Barlas, E., Bultan, T.: NetStub: A framework for verification of distributed Java applications. In: 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2007. pp. 24–33. ACM (2007)
6. Chandy, K.M., Misra, J.: Distributed computation on graphs: Shortest path algorithms. Comm. ACM (1982)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
8. Dwyer, M.B., Person, S., Elbaum, S.G.: Controlling factors in evaluating path-sensitive error detection techniques. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006. pp. 92–104. ACM (2006)

9. Fidge, C.J.: Partial orders for parallel debugging. In: Workshop on Parallel and Distributed Debugging. pp. 183–194 (1988)
10. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005. pp. 110–121. ACM (2005)
11. Fredlund, L.Å., Svensson, H.: McErlang: A model checker for a distributed functional programming language. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 125–136 (2007)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
13. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Informatica (2009)
14. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: Computer Aided Verification, 21st International Conference (CAV). LNCS, vol. 5643, pp. 398–413. Springer (2009)
15. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: A comparative analysis. In: Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (2009)
16. Lauterburg, S., Dotta, M., Marinov, D., Agha, G.: A framework for state-space exploration of Java-based actor programs. In: 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009. IEEE (2009)
17. Pi original source code webpage, `http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/simplempi/main.htm`
18. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer (2006)
19. Vakkalanka, S.S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Computer Aided Verification, 20th International Conference, CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer (2008)
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (April 2003)
21. Yabandeh, M., Knežević, N., Kostić, D., Kuncak, V.: CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In: NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation. pp. 229–244. USENIX Association (2009)
22. Yang, J., Chen, T., Wu, M., Xu, Z., Xuezheng Liu, H.L., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: Transparent model checking of unmodified distributed systems. In: NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation. pp. 213–228. USENIX Association (2009)
23. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: Model Checking Software, 14th International SPIN Workshop. LNCS, vol. 4595, pp. 58–75 (2007)
24. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Model Checking Software, 15th International SPIN Workshop. LNCS, vol. 5156, pp. 288–305. Springer (2008)
25. Yi, X., Wang, J., Yang, X.: Stateful dynamic partial-order reduction. In: Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006. LNCS, vol. 4260, pp. 149–167. Springer (2006)