# Static-analysis Assisted Dynamic Verification to Efficiently Handle Waitany Non-determinism *

Sarvani Vakkalanka[1], Grzegorz Szubzda[1], Anh Vo[1],
Ganesh Gopalakrishnan[1], Robert M. Kirby[1], and Rajeev Thakur[2]

[1] School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA
[2] Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA

**Abstract:** We consider the problem of verifying MPI using `MPI_Waitany` (and related operations wait/test some/all). Such programs have a higher degree of non-determinism over which user-written test harnesses cannot exert adequate control. This makes conventional MPI program testing even more ineffective. Our previous work has demonstrated the advantages of an MPI-specific dynamic partial order reduction algorithm called POE in a tool called ISP in dramatically reducing the number of interleavings during formal dynamic verification. We show that POE can become ineffective when naïvely applied to MPI programs with `MPI_Waitany`s. As a novel solution, we employ static analysis in a supporting role to POE to determine the extent to which the *out* parameters of `MPI_Waitany` can affect subsequent control flow statements. This informs ISP's scheduler to exert even more intelligent backtrack/replay control. Our results indicate that this combined use of static analysis and POE can lead to exponentially better overall verification time.

## 1 Introduction

The correctness of MPI programs is of paramount importance, especially considering the losses due to errant or crashing simulations. Conventional testing oriented approaches may work for straightforward (and slow) MPI codes, but become increasingly inadequate as the MPI program is progressively optimized. Three related optimizations that increase the ineffectiveness of testing are: (i) the use of `ANY_SOURCE` (wildcard, shown by "*" in this paper) receives that permit receiving the results of previously launched computations from one of many processors, and re-launching the next batch of work on those processors, (ii) the use of non-blocking MPI operations (`Isend` and `Irecv`) that enhance the computation/communication overlap, and (iii) the use of `MPI_Waitany`, `MPI_Testany`, `MPI_Waitsome`, and `MPI_Testsome` to start further processing as soon as the already issued MPI operations finish. It is clear that these three MPI features go hand-in-hand, all contributing to the explosion of non-deterministic outcomes and control paths. Much of this non-determinism is very difficult to control from

* Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, and the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

user-written test harnesses. This makes conventional MPI program testing even more ineffective for this class of programs.

MPI program testing tools are hampered by many factors. Consider an MPI program with five processes, each executing a *straightline program* of exactly five MPI calls (called *Generic Example*). Even this program has potentially 10 billion interleavings (schedules)! Now, if all these MPI calls are point-to-point and non-wildcard, they become *independent operations* in the parlance of [8, Chapter 10]. That is, rearranging the order in which different processes invoke their MPI functions *does not* affect the ability to detect deadlocks, resource leaks, and local assertion violations in such a program. Since conventional MPI testing tools do not employ any notion of independence to guide their schedule selection, they may un-necessarily test such a program over multiple interleavings (obtained, perhaps, by inserting random delay statements). Now, let us make a slight change in this example by introducing exactly one wild-card receive in process P0 which is matched by sends Send1(to P0) and Send2(to P0) issued by processes P1 and P2, respectively. Let all other MPI calls in this example remain as before. In this case, it is important to explore two *relevant* interleavings: one where Send1(to P0) matches Recv(from *) and the other where Send2(to P0) matches Recv(from *) (because these matches can affect verification outcome by conveying different values to the wildcard receive). However, a testing approach may never explore both these matches (as corroborated by the omissions documented in [1]).

Even if a testing tool intends to match these sends and receives, it must somehow enforce these matches within the MPI runtime, whose innards are not easily accessed. Thus, even though it may fire Send1 followed by Send2 and then Send2 followed by Send1, it is possible that Send1 matches Recv(from *) in both cases. This explains why MPI programs suddenly reveal bugs when ported to different machines, because porting typically affects such matches.

This paper is on sound verification of MPI programs containing the three related optimizations discussed above using *dynamic formal verification methods* – in particular, our ISP tool that supports a novel form of dynamic partial order (DPOR) reduction called Partial Order avoiding Elusive interleavings (POE) [20, 19, 15, 21]. ISP has been used to verify a number of real-world MPI programs (*e.g.*, ParMETIS [14], MADRE [17], and the Implicit Radiation Solver or IRS [6]) for the absence of deadlocks, resource leaks, communication races, and assertion violations [20, 21]. ISP has been released [2] to run on Linux, Mac OS/X, and Windows, supporting the MPI libraries MPICH2, Open-MPI, and Microsoft MPI, supporting more than 60 MPI 2.1 functions.

In our past work, we have described how POE helps verify MPI programs. For a given MPI program driven by a test harness (input data-set), POE guarantees the exploration of all relevant interleavings based on MPI operation dependencies. These interleavings are sufficient to unearth deadlocks, assertion violations, or resource leaks (hereafter called *safety properties* [8]). For ParMETIS this process resulted in one relevant interleaving due to the absence of non-determinism. Meaningful handling of non-determinism is key to the success of any formal verification tool. To understand how this works, let us revisit our *Generic Example*.

For this example, POE *enforces* the two matches that are required to occur, by dynamically rewriting Recv(from *) into Recv(from P1) and Recv(from P2) over two successive replays of the program. Since POE replaces wildcard receives by specific receives automatically and parsimoniously, it can ensure that all relevant interleavings are pursued.

**Contributions:** This work describes all the innovations that were necessary in order to add efficient support for MPI WaitAny, TestAny, WaitSome, and TestSome (generically referred to by 'WaitAny') into ISP. Briefly, the POE algorithm, despite all its attempts at finding relevant interleavings can find *too many relevant interleavings*! However, by analyzing how the *out* parameters of WaitAny are used further in the code path, one can dramatically reduce the number of interleavings explored. Determining this usage can be done through static analysis, which we have implemented using LLNL's Rose framework [3]. Here is a summary of our approach:

- After determining the use of these *out* parameters through static analysis, we emit `PMPI_Pcontrol` statements into the user code.
- During dynamic verification, the `PMPI_Pcontrol` statements convey vital information to ISP's scheduler, often helping it exponentially reduce the relevant interleavings set.

**Related Work:** Static analysis, when used by itself, can generate false alarms due to loss of information about control-flow paths. Any such alarm has to be separately checked by the designer, and may waste valuable designer time [12]. Model-based verification (*e.g.*, modeling programs in dialects (*e.g.* [16]) of SPIN [13]) is another formal approach. This approach requires designers to re-express an MPI program in an alternate notation, and hence may prove impractical for realistic MPI programs. The modeling burden is compounded by having to model the semantics of MPI calls and the surrounding C codes after each program tuning step.

**Dynamic Verification:** Our approach – dynamic formal verification – runs the program after replacing the native scheduler with a *formal* verification scheduler. Pioneered by Godefroid [11] and later improved in the dynamic partial order (DPOR) approach [10], dynamic verification enjoys a growing presence, in tools such as CHESS [4], MODIST [22], CREST [7], Bandera [9], Java PathFinder [5], and Inspect [23]. As said earlier, our dynamic verification approach for MPI programs based on a customized DPOR algorithm called Partial Order avoiding Elusive interleavings (POE).

**Roadmap:** Section 2 presents the motivations for our effort. Section 3 presents the details of our implementation through an example. Section 4 presents our experimental results and concluding remarks.

## 2 Motivations for Static Analysis Support for POE

Consider the MPI example shown in Figure 1(a). The MPI code for rank 0 (lines $1-12$) issues two `MPI_Irecv`s (lines $2-3$) and later invokes a `MPI_Waitany`

```
1: if (rank == 0) {
2:    Irecv (from 1, &req[0]);
3:    Irecv (from 2, &req[1]);
4:
5:    Waitany (2, req, &done,
                  &status);
6:    if (done == 0) {
7:       ...   // More MPI code
8:    } else if (done == 1) {
9:       ...   // More MPI code
10:   }
11:   Waitany(2, req, &done,
                  &status);
12: }
13: if (rank == 1) {
14:   Isend( to 0, &req[0])
15:   Wait(&req[0]);
16: }
17: if (rank == 2) {
18:   Isend( to 0, &req[0])
19:   Wait(&req[0]);
20: }
```

(a)

```
1: if (rank == 0) {
2:    Irecv (from 1, &req[0]);
3:    Irecv (from 2, &req[1]);
4:
5:    for (i = 0; i < 2 ; i++)
6:       Waitany (2, req, &done,
                        &status);
7: }
8: if (rank == 1) {
9:    Isend( to 0, &req[0])
10:   Wait(&req[0]);
11: }
12: if (rank == 2) {
13:   Isend( to 0, &req[0])
14:   Wait(&req[0]);
15: }
```

(b)

Fig. 1: (a) Waitany example with conditionals (b) Example without conditionals

(line 5, line 11) on the request handles returned by the `MPI_Irecv`s. The code in lines $6-10$ will execute along code paths determined by the value of the `done` variable (recall that this variable represents the index into the request array returned by `MPI_Waitany`). For sound verification, ISP must ensure that a sufficient number of values of the `done` variable are returned. In this example, `done = 0` and `done = 1` are both feasible, since both the posted `Irecv` have matching sends available. Thus, ISP will execute two interleavings, one for each value of `done`.

Now consider the MPI program in Figure 1(b) which is a slight modification of that in Figure 1(a). The MPI program issues two `MPI_Waitany`s in a for loop (lines $5-6$). This program would result in $2! = 2$ interleavings due to `MPI_Waitany`s. This is because the `Waitany` posted during the first iteration of the `for` loop would have two potential `done` values, and the subsequent `Waitany` would have one potential `done` value. Now, if rank 0 posted yet another `Irecv` from, say rank 3, and the `for` loop in Figure 1(b) iterated three times, this would result in $3! = 6$ interleavings (3 interleavings from the Waitany due to the first iteration, 2 interleavings from the second iteration and 1 from the last iteration of the `for` loop, resulting in $3 \times 2 \times 1 = 6$ interleavings). In general, Waitanys in an MPI program can result in exponential increase in the number of interleavings even though a single Waitany can cause a linear number of interleavings in the

number of requests[3]. However, observe that the MPI program in Figure 1(b) has no conditional paths based on the value returned in the *done* variable. Thus, if ISP had some "look-ahead," it could have gotten away with just one interleaving, because it would have known that regardless of the value of `done`, *all residual code paths would be the same!*

In summary, since ISP has no information about the code structure except for the dynamic MPI functions invoked by the program, the current ISP will blindly execute the two interleavings for the above program. Our contribution in this paper is to (i) devise an annotation mechanism to inform ISP of the residual code paths following a Waitany, (ii) develop static analysis to compute and decorate the user code with these annotations, and (iii) modify ISP to take the annotations into account. Annotations are placed through `MPI_Pcontrol` calls. `MPI_Pcontrol` has the following prototype:

```
MPI_Pcontrol (const int level , int count, bit_vector vec);
```

where `level` must be set to `ISP_INDEX_VEC` so that ISP understands that the Pcontrol corresponds to a following Waitany. `count` is the number of requests in the Waitany and `vec` is a bit vector of 0s and 1s of size `count`. A 0 in a bit indicates that ISP can ignore that bit and hence will not have any interleaving for the request corresponding to that bit. A 1 in a bit indicates that ISP must interleave on the request corresponding to that index.

For the MPI program in Figure 1(a), using a syntactic extension for bit vectors, the static analysis procedure would modify the `MPI_Waitany` function call as follows:

```
MPI_Pcontrol (ISP_C_INDEX_VEC, 2, <1,1>)
 MPI_Waitany (2, reqs, &done, &status);
```

The bit vector "`<1,1>`" indicates to ISP that all the requests for the following `Waitany` are relevant.

For the example in Figure 1(b), the static analysis procedure would modify the `MPI_Waitany` function call as follows:

```
MPI_Pcontrol (ISP_C_INDEX_VEC, 2, <0,0>)
 MPI_Waitany (2, reqs, &done, &status);
```

The bit vector "`<0,0>`" indicates to ISP that none of the requests for the following `Waitany` are relevant and hence ISP can randomly select any of the requests to generate an interleaving.

Now consider the example in Figure 2(a) where the annotations are a complex function of the loop iteration (hence, without automation, would be too error-prone to obtain manually). Each dynamic instance of `MPI_Waitany` in the loop will have a different bit vector ("`<1,0>`" for the first iteration and "`<0,1>`" for the second iteration) while the code has only a single `MPI_Waitany`:

[3] The situation for MPI_Waitsome is worse as each invocation of Waitsome results in $2^{count}$ interleavings where *count* is the number of requests passed to MPI_Waitsome

```
1: if (rank == 0) {
2:    Irecv (from 1, &req[0]);
3:    Irecv (from 2, &req[1]);
5:
6:    for (i = 0; i < 2; i++) {
7:       Waitany (2, req, &done,
                        &status);
8:       if (done == i) {
9:          ...
10:       }
11:    }
12: }
13: if (rank == 1) {
14:   Isend( to 0, &req[0])
15:   Wait(&req[0]);
16: }
17: if (rank == 2) {
18:   Isend( to 0, &req[0])
19:   Wait(&req[0]);
20: }
```

(a)

```
1: if (rank == 0) {
2:    Irecv (from *, &req[0]);
3:    Isend (to 1, &req[1]);
4:
5:    Waitany (2, req, &done, &status);
6:    if (done == 0) {
7:       ...
8:    }
9:    Barrier
10:   Waitany (2, req, &done, &status);
11: }
12: if (rank == 1) {
2:    Irecv (from 0, &req[0]);
14:   Isend( to 0, &req[1]);
15:   Barrier;
16: }
17: if (rank == 2) {
18:   Barrier;
18:   Isend( to 0, &req[0])
20: }
```

(b)

Fig. 2: (a) Non-trivial example of annotations (b) MPI program showing priority conflict

It must be noted that the above example also has a single interleaving except that unlike the MPI example in Figure 1(b), the order of `done` returned by ISP is important. We show in section 3, how our static analysis tool will modify the program in Figure 2(a) so that ISP will only generate one interleaving instead of two. We are now ready to describe the implementation details of `Waitany` in ISP.

## 3 Implementation of MPI_Waitany

A brief summary of how ISP exerts scheduling control is as follows. We over-ride each MPI function $MPI\_f$ with our own definition, using the PMPI mechanism. In the overriding function, we communicate with the ISP scheduler to ensure that it is timely to issue an MPI command. When the ISP scheduler gives the go-ahead, the corresponding function PMPI_$f$ is issued. Now consider a code fragment `Isend(.. &handle); Wait (&handle)`. In this case, the overriding definition for `Isend` simply collects, but does not issue this function. However, when an `MPI_Wait` is encountered, a *fence* is reached within this process (all subsequent operations may complete only after Wait completes). In this case, the overriding function `Wait` will eventually issue a `PMPI_Wait.` Details are provided in [19].

In this section we will show how `MPI_Waitany` is implemented in ISP using the examples in Figure 1(a) and Figure 2(a). The MPI program in Figure 1(a) will be executed by our Waitany algorithm as follows:

- The two `Irecv`'s from rank 0 (lines $2-3$) are collected but none are issued.
- Collect the first `Waitany` from rank 0 (line 7) but do not issue it.
- Since `Waitany` is a fence, stop collecting and switch to rank 1.
- Collect the `Isend` of line 14 and issue it (reasons discussed in the Iprobe paper submitted) but mark it as "not matched".
- Collect the `Wait` in line 15. Since `Wait` is a fence, stop collecting and switch to rank 2.
- Collect the `Isend` of line 18 and similarly to the Isend of line 14, issue it, but mark it as "not matched".
- Collect the `Wait` in line 19. Since `Wait` is a fence, stop collecting.
- At this point all the processes are at their fence instructions.
- Match the `Irecv` of line 2 to `Isend` of 14 and issue the `Irecv` and mark the `Isend` as "matched".
- Similarly match the `Irecv` of line 3 to `Isend` of line 18 and issue the `Irecv` and mark the `Isend` as "matched".
- Since the `Isend`s are all matched, the corresponding `Wait`s on line 15 and 19 are issued.
- For the `Waitany` (line 11) on which rank 0 is blocked, since both `req[0]` and `req[1]` are now complete, the `done` can be set to either 0 or 1. ISP interleaves on both the above possibilities - Instead of issuing the `Waitany`, issue `Wait(req[0], &status)` and set the value of `done` to 0 and return the control back to the MPI program.
- ISP will now receive another `Waitany` (line 11). In this case issue `Wait(&req[1], &status)` and set the value of `done` to 1 (note that since `req[0]` is now set to `MPI_REQUEST_NULL` due to the Wait in the previous step.
- Restart the MPI program and execute exactly from step $1-11$. However, now for the first `Waitany` (line 5) issue `Wait(&req[1], &status)` and set `done` to 1 and return the control back to the program.
- For the second `Waitany` (line 11), issue `Wait(&req[0], &status)` and set `done` to 0.

ISP hence generates two interleavings for the two different possibilities for the `done` variable.

**Priority conflict with wildcard receive:** ISP's POE algorithm works by prioritizing the issue/execution order of various MPI functions [20, 19]. In this priority order, wildcard receives have the least priority (this allows finding all possible matching sends to the wildcard receives). Consider the MPI program shown in Figure 2(b). The wildcard receive (line 2) has a matching send from rank 1 (line 14) available when the `Waitany` on line 5 is encountered. However, the send from rank 2 (line 19) is not yet seen due to the presence of the `Barrier` (line 18). Since `Irecv` is of the lowest priority, ISP would not allow the `Irecv` to complete and instead force the `done` to be 1 which completes the `Isend` of rank 0. This means that the conditional path from line $6-8$ will never be executed.

Since our goal is to guarantee bug detection, ISP must also execution the code in line $6-8$ in some interleaving. To allow for this, we allow a `Waitany` and a wildcard receive to have equal priority when the waitany is associated by the request of the wildcard receive and that wildcard receive has some matching sends for it. When two MPI operations $x$ and $y$ have equal priority, ISP executes one interleaving with $x$ having higher priority than $y$ and another interleaving with $y$ having higher priority than $x$.

Hence, for the example in Figure 2(b), we will have two interleavings. One interleaving with the wildcard receive having higher priority than Waitany in which case, the conditional path lines $6-8$ are executed and another interleaving where the Waitany has higher priority than wildcard receive and in this case, the wildcard receive will match with the Isend of rank 2 (line 19). This will ensure full path coverage as well as exploit all possible non-determinism in the program.

**Static Analysis Support:** In order to reduce redundant inteleavings our static analysis tool conservatively finds those requests that ISP must interleave on instead of blindly interleaving on *all* possible outcomes of `Waitany`. Consider the example in Figure 3(a) which is same as Figure 2(a) (shown here for convenience).

```
1: if (rank == 0) {
2:    Irecv (from 1, &req[0]);
3:    Irecv (from 2, &req[1]);
5:
6:    for (i = 0; i < 2; i++) {
7:       Waitany (2, req, &done,
                        &status);
8:       if (done == i) {
9:          ...
10:       }
11:    }
12: }
13: if (rank == 1) {
14:   Isend( to 0, &req[0])
15:   Wait(&req[0]);
16: }
17: if (rank == 2) {
18:   Isend( to 0, &req[0])
19:   Wait(&req[0]);
20: }
```

(a)

```
1: for (i = 0; i < 2; i++) {
2:    bit_vector vec(2);
3:    for (j = 0; j < 2; j++) {
4:       if (j == i)
5:          set_bit(vec, j);
6:       else
7:          clear_bit(vec, j);
8:    }
9:    MPI_Pcontrol(ISP_C_INDEX_VEC,
                                  2, vec)
10:    Waitany(2, req, &done,
                                &status);
11:   if (done == i) {
12:      ...
13:   }
14: }
```

(b)

Fig. 3: (a) Example with conditional path on `done` (b) `for` loop after static analysis

Our static analysis tool changes the `for` loop of Figure 3(a) (lines 6-11) as shown in Figure 3(b). The `MPI_Waitany` API is annotated with a bit vector where the number of bits in the string is the number of requests passed to `Waitany`. Each request has a corresponding bit in the bit vector where a 0 in the

bit indicates that the corresponding request will have no effect on the control flow of the program and hence can be ignored while a 1 in the bit vector for a request indicates that the request can change the program flow and hence must be explored by ISP. When ISP receives the `Waitany` along with the bit vector, it interleaves on `Waitany` based on the bit vector with requests that have 1. Hence, for the program modified as in Figure 3(b), ISP will generate only a single interleaving while for the MPI program in Figure 3(a), since there is no additional information being passed to ISP, it generates two interleavings.

Our static analysis determines the construct in which the out variable `done` is used and duplicates that construct in the `for` loop of Figure 3(b) (lines 3-8). The construct can only be duplicated if it satisfies the following conditions: (i) the construct must be idempotent such that the evaluation of the expression has no side-effects, (ii) the construct must not use computation done after the call to Waitany; such computation, for example, includes variables mutated after `Waitany`, and (iii) the construct must not make the out variable available globally such as by assigning it to a global data structure or using the variable as an argument in an opaque function call. If any construct in which the out variable is used cannot be duplicated, our static analysis cannot determine a subset of all the possible out variable values. In such a case there is no reduction in the number of interleavings explored.

## 4 Summary, Results, and Concluding Remarks

This paper presents a novel use of static analysis to determine the degree to which the *out* parameters of MPI's Waitany are decoded in control-flow constructs, and based on this information reduce the number of interleavings that a dynamic analysis tool must perform. Our specific contributions are as follows:

- Recognizing this lurking exponential explosion associated with Waitany (and related constructs).
- Device a static analysis procedure that captures usage of the value of the *out* parameter in forward flow paths after Waitany.
- Employ the `MPI_Pcontrol` mechanism to pass information from static analysis into the dynamic scheduler.
- Perform preliminary feasibility of the ideas on a working extension of ISP.

We have run our static analysis tool and ISP on the various example presented in this paper. At present, larger examples such as MADRE [17] did not significantly benefit from our static analysis. However, the very purpose of including such a static analysis procedure as described here into ISP would be to smoothly handle cases where an exponential explosion of interleavings is likely to occur. In Section 2, we have argued that this explosion is quite possible. All our test cases were handled correctly by our modified static analysis. This project also allowed us to assess the state-of-the-art static analyzer Rose, and demonstrate that it can be successfully integrated into dynamic analysis tools such as ISP. In our assessment the static analysis itself added negligible run-time

overheads. Our static analysis tool is at a preliminary state of development. As part of our future work, we will further our static analyzer and benchmark the improved tool on a large suite of public-domain examples.

## References


1. http://www.cs.utah.edu/formal_verification/ISP_Tests/.
2. http://www.cs.utah.edu/formal_verification/ISP-release/.
3. http://rosecompiler.org.
4. http://research.microsoft.com/en-us/projects/chess/.
5. http://javapathfinder.sourceforge.net/.
6. The IRS Benchmark Code. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/%.
7. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, Univ. of California, Berkeley, Sep 2008.
8. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
9. M. Dwyer, J. Hatcliff, and D. Schmidt. Bandera : Tools for automated reasoning about software system behavior. In *ERCIM News, 36*, Jan. 1999.
10. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.
11. P. Godefroid, B. Hanmer, and L. Jagadeesan. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal*, 3(2), April-June 1998.
12. P. Godefroind and N. Nagappan. Concurrency at microsoft - an exploratory survey, 2008.
13. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
14. G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
15. S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. *EuroPVM/MPI 2008.*
16. S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. *SPIN 2004.*
17. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. *EuroPVM/MPI 2008.*
18. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. *PADTAD-VI 2008.*
19. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Implementing efficient dynamic formal verification methods for MPI programs. *EuroPVM/MPI 2008.*
20. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *CAV 2008.*
21. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. *PPoPP 2009.*
22. J. Yang, et al. MODIST: Transparent Model Checking of Unmodified Distributed System. NSDI 09. To appear.
23. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. *SPIN 2008.*