System-level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications

Marion Guthmuller, Martin Quinson, Gabriel Corona

System-level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications

Marion Guthmuller*†‡, Martin Quinson*†‡ and Gabriel Corona*†‡
*Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, France.
†CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, France.
‡Inria, AlGorille project-team, Villers-lès-Nancy, France.

***Abstract*—The ever increasing complexity of distributed systems mandates to formally verify their design and implementation. Unfortunately, the common approaches and existing tools to formally establish the correctness of these systems remain hardly applicable to the kind of legacy applications that are commonly found in the HPC community.**

We present how system-level memory introspection can be achieved directly at runtime without relying on the source code analysis. We use this mechanism to detect the equality of the application's state at system level. As the storage of the system state may be memory expensive, we compact the memory by sharing unchanged memory pages between snapshots. This enables the automated verification of safety and liveness properties on legacy distributed applications written in Fortran or C/C++ using the MPI standard. We demonstrate the effectiveness of our approach on several programs from the MPICH3 test suite.

I. Introduction

Model checking is an appealing automated technique to establish the correctness of distributed systems, but it is difficult to apply to legacy applications as it requires a complete model of the application. Manually building such models is error-prone and labor-intensive. Keeping the resulting model up to date when the real application is modified constitutes another challenge. A guided approach such as the CEGAR abstraction/refinement methodology [1] can ease this modeling step, but the user still needs a high level of expertise in formal methods. Static code analysis [2] can automatically reconstruct the model, thus removing the burden induced by the modeling step. This interesting approach is used in many existing tools [3], [4], [5]. Our work is in line with another approach called *formal dynamic verification* (or execution-based model checking), where the model is not explicitly known but only implicitly explored through the actual execution of the real application. The verification is thus performed on the concrete implementation of the application. It is in some sense orthogonal with the static analysis, as a tool could for example leverage information gathered statically to improve the dynamic exploration.

Ultimately, our goal is to dynamically verify unmodified legacy distributed applications which allow Communicating Sequential Processes (CSP) to interact through message passing using MPI or similar APIs. We do not aim at *certifying* such applications, but merely at finding unknown issues in real applications. We dynamically verify software toward bug finding through falsification.

We build upon a previous work of Rosa *et al.* described in [6]. Authors presented in this paper a multi-API solution enabling the formal verification of such applications within the SimGrid framework [7]. In the current work, we tackle the problem of introspecting the state of arbitrary legacy applications written in Fortran or C/C++. The motivation comes from the need to detect state equalities during the exploration, and thus to detect cycles and infinite executions. Detecting such cycles also proves important to verify liveness properties in the general case. Our approach faced with this problem is to leverage debugging techniques and tools to retrieve semantic information on the running application toward system-level detection of application's state equality.

Specifically, this paper makes the following contributions: we detail the OS-level challenges behind the considered problem and we propose solutions to mitigate each of those difficulties. We show the practical effectiveness of our proposal through three kinds of experiments: we find a liveness issue in a custom MPI code, we explore exhaustively an infinite-time MPI application as well as several of the MPI applications from the official MPICH3 testsuite.

The remainder of this paper is organized as follows: Section II introduces our motivations and the problem statement. Section III details our contribution, which is evaluated in Section IV. Section V presents the related work while Section VI concludes this paper and discusses some leads of future work.

II. Context, Motivation and Problem Statement

Originally, the SimGrid framework was intended to assess the performance of distributed applications through fast but realistic simulations [7]. SimGridMC extends this framework to evaluate the correctness of distributed applications through dynamic verification. As detailed in [6], SimGridMC leverages the simulator architecture, where the distributed processes are folded as threads into a system process, and where all communications are mediated by the simulator. Folding the evaluated application in a single process makes it much easier

to inspect the network state and to checkpoint the complete system state. The non-deterministic choices are controlled through the mediation of communication, enabling the whole verification process. In addition, every memory allocation is intercepted to two separate heaps: the application's memory gets overwritten when restoring the system before the exploration of another branch while the model checker's memory is naturally preserved on such rewinds.

Prior to this work, SimGridMC was *stateless*, only the system's initial state was checkpointed. When rewinding the application, this initial state was first restored and then all transitions leading to the desired state were replayed. This was satisfying in the considered context of Peer-to-Peer (P2P) protocols, because the computations are classically inexpensive in this case. A Dynamic Partial Ordering Reduction (DPOR) technique helped mitigating the state space explosion.

While relatively effective, this stateless approach suffered from several problems in terms of performance and applicability, as detailed below.

A. Efficiently Verifying HPC Programs and State Equality Reduction

By reusing the internals of the versatile SimGrid framework, SimGridMC could already verify applications specified with any of the APIs implemented on top of SimGrid, from the simple specific interfaces to the classical MPI standard which were partially implemented on top of SimGrid. But the stateless approach is less adapted to HPC applications, as computations become much more expensive on path replays. It then becomes interesting to checkpoint more states to save time when rewinding the system. Several verification tools use stateful explorations, but they either save only parts of the system state after decomposition [8] or only save a bitstate hash of each reachable state rather than the state itself [9]. In these cases, the intermediate states cannot be restored and must be recomputed. Until recently, memory limitations made it impossible to checkpoint the full state at each step, but these limitations now tend to become less pressing as systems with hundreds of gigabytes per node become available. It is then appealing to refine the state comparison using any memory information.

The stateful exploration can also reduce the size of the explored state space by detecting whether the current state was already visited previously, and cutting the exploration when it occurs. Implementing this approach is however much harder with real applications than with abstract models, as the application's state is obviously harder to explore. Bitstate hashing is often leveraged in the literature to that extend, even if any hash collision would hinder the exploration soundness. Moreover, technical considerations make very difficult to compute that hash for some systems that we consider in this work.

We checkpoint all states instead and detect cycles through the comparison of the current state with all previously checkpointed states. As detailed in next section, syntactic memory comparison byte per byte is not sufficient in our case, as many details may vary without altering the application's semantic. All details of that semantic must be considered accurately instead, including global variables, the heap, the simulated processes' stacks, as well as the simulator's network state. Our work builds upon the many existing solutions based on canonicalization by applying similar techniques to languages which are not garbage collected. Introspecting the raw data then reveal much more challenging.

B. Verifying Arbitrary Liveness Properties on Legacy Code

Execution loops as detected by the state equality mechanism constitute *non-progressive cycles*. They play a central role in the verification of liveness properties, as the counterexample to liveness properties are infinite paths. If the application state size is bounded (in particular, if the stack size is bounded), infinite paths must contain such an execution loop. Liveness properties are then verified through the search of acceptance cycles in the Cartesian product of the application with a Büchi automaton encoding the negation of the verified property. If found, such an acceptance cycle denotes an infinite execution path which constitutes a counterexample to the property.

This approach can sometimes be used to falsify the program termination using the property *"Always, Eventually, the program terminates"*. If it is naturally impossible to solve the Halting Problem in all generality, it remains sometimes possible to prove whether a given program terminates or not [10]. Dynamic verification can enforce the termination of any finite protocol (although sometimes inefficiently). It could also correctly diagnose applications that do not terminate because of non-progressive cycles (provided that these cycles are detected). This approach fails on applications which do not terminate but whose the state changes infinitely often, which is consistent with the indecidability of the Halting Problem.

MaceMC [11] can also verify liveness properties on concrete C++ implementations of distributed systems. Instead of detecting the acceptance cycles, it looks for the so-called critical transition that plunged the property into a violation. The exploration performance is then improved using state hashing. This approach remains however limited to the restricted set of liveness properties that can be expressed as $\Box\Diamond p$ (*Always Eventually* p, where p is a logical predicate), while detecting acceptance cycles can be used to verify arbitrary LTL_{-X} formula.

C. Verifying (infinite-time) Cyclic Protocols

Non-progressive cycles constitute an inherent part of a whole class of applications, such as the cyclic protocols which react to external periodic events (as most P2P protocols).

DPOR cannot study such systems because even if it reduces the amount of explored interleavings by detecting the independent actions, it does not detect all execution cycles. The state space is still infinite after the DPOR and thus cannot be explored explicitly during a dynamic verification. On the contrary, a state equality reduction would cut the repetitive patterns of the application behavior, which permits the exhaustive verification of cyclic protocols.

III. System-level State Equality Detection

Comparing memory byte per byte is not sufficient to detect the state equality, as it detects many syntactic differences that

are not significant to the application semantic. For example, the numerical values of pointers on different memory areas are syntactically different while the pointed memory areas could be semantically equals. Such false negative must be avoided by all means, as the tool could fail to detect some state equalities, possibly leading to the non-detection of a property violation.

The key to a better state equality detection lies in the ability to introspect the memory and to reconstruct the semantic of bits. Several approaches can be leveraged to that extent. For Java applications, all of the memory is handled by the virtual machine. Using the meta-data known to the JVM, it should thus be possible to reconstruct the semantic of each byte found in memory, making it possible to detect system state equalities. However, this would probably require to modify the JVM directly, as Java introspection is more intended to explore the values of objects' content rather than exploring the meaning of memory locations. Adding the needed meta-data and keeping them in sync in the whole JVM source code constitutes a daunting task, but remains feasible [12].

However, Java is almost never used in the context of HPC applications using MPI, calling for another approach that would be applicable to C/C++ applications. In [13], the user must provide a hashing function that can be used to detect the state equalities. Detecting system state equalities is easier when the complete data semantic is known, but this remains a burdensome and highly error-prone task, hardly adaptable to complex systems. In the following, we propose a generic solution which operates at the operating system level. Instead of specifying the segments of memory that are relevant to the system's semantic, we start by considering the whole system memory, and optionally ignore some sections that are known to be irrelevant.

The system state that we consider aggregates the application's global variables, the application's heap and the stack of each process (Fig. 1). The global variables of the simulator are also included in the comparison, as they contain the network's state during the simulation. At the OS-level, these data are stored in several memory *segments* that must be considered separately. In the following, we detail our approach using Linux as an example, but our work could probably be implemented for any other operating system.

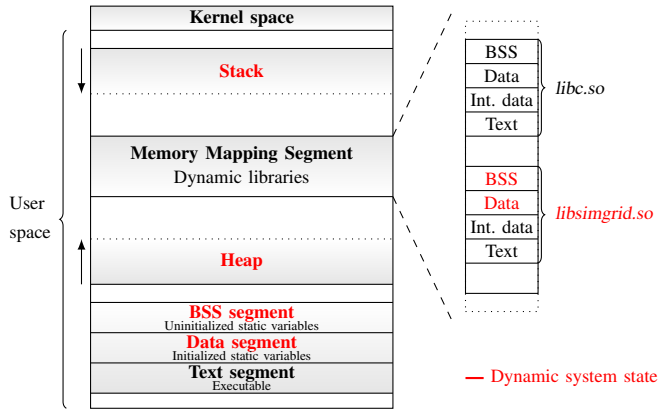

Fig. 1. Memory layout of a process.

In the remainder of this section, we detail the causes of the syntactic differences that defeat the byte per byte comparison of memory segments. For each of these challenges, we show how additional information could be retrieved from the system, and leveraged to rebuild semantic information needed. For some of these difficulties, we must rely on heuristics that could conclude on the difference of semantically equal states. This must be avoided when possible, as this could jeopardize the soundness of the verification process. That is why we do not aim at certifying the verified applications. Our tool is rather intended to *find bugs* that are hard to detect on real applications.

A. Memory Overprovisioning

Most implementations of the `malloc` library allocate memory chunks whose sizes are powers of two to avoid the memory fragmentation. For example, a memory area of 64 bytes will be used to serve a request of 48 bytes. It is expected that the application only uses the requested area while the extra area remains unused (Fig. 2). In our context, this memory overprovisioning may result in irrelevant byte differences, as the unwanted area contains unspecified values: the data written by the previous user of each remaining memory chunk.

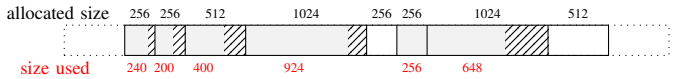

Fig. 2. Memory allocation with overprovisioning.

As detailed earlier, SimGridMC uses two separated heaps for the verified application and the model checker itself. We extended this implementation of `malloc` (based on `mmalloc`) to address the issue of memory overprovisioning.

A first approach to not take these unspecified values into account is to ignore the unwanted area during the state comparison. But this would not be robust against applications with buffer overflows. In particular, if write accesses overflow from the requested area but remain within the allocated area, the application would fail with our malloc but work with a classical one. In order to address this issue, we fill each newly allocated area with zeroes before handing it to the application. This ensures that every value is specified while remaining robust to limited buffer overflows. Although SimGridMC is not specialized in the detection of this kind of bug, we could add an option to detect and report such overflows to the user.

Memory overprovisioning also occurs within the processes' stacks. When a function returns, the memory occupied by its stack frame is not zeroed by the system, resulting in irrelevant syntactic differences. In this case, instead of zeroing the memory area after each function return, we retrieve the stack pointer in a portable way using the libunwind library[1], and only consider the valid stack frames during the comparison. It is safe to assume that no buffer overflow occurs on the stack given the protection added by the compiler against *stack smashing* security attacks.

[1] http://www.nongnu.org/libunwind/

B. Padding Bytes

When necessary, the compiler adds some *padding bytes* between the variables to enforce memory alignment constraints that speed up the data movement between the memory and the CPU registers (Fig. 3). Since the application's structures are sparse, this mechanism results in irrelevant syntactic differences if the values of padding bytes are compared.

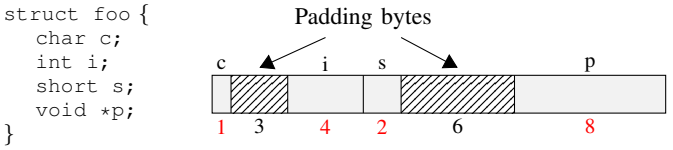

Fig. 3. Data memory alignment is preserved by adding unused padding bytes.

In the heap, our solution to the previous issue happens to address that problem too, since the padding bytes are zeroed and thus specified. This remains problematic in the stack because of irrelevant differences induced by the system. Even if we zero the stack before each call (as done in §III-E), the compiler or the system could add syntaxic differences such as the *canaries* used to detect stack smashing security attacks.

Our approach is to focus on the actual data stored on the stack instead and to ignore the unspecified values of padding bytes. We retrieve the needed information from the debugging symbols. The description of all variables contained in the system state (name, type, size and memory address) is retrieved from the DWARF[2]-formatted debugging information. It is a debugging file format used by many compilers and debugger to support source level debugging. This information is sufficient to rebuild a fine-grain semantic of each byte in the global segment. It can also be coupled with stack frame information extracted from libunwind to explain each byte of the stack area. This way, the comparison is only done on information which characterizes the system state at a given execution time.

It is possible to disable these padding bytes with some compilers by setting the alignment of all aggregate members to a specified byte boundary. This operation may be performed thanks to the directive `#pragma pack(1)`. However, this may have a significant impact on the performance of the application as the compiler must (on some platforms) generate code to access a misaligned member a byte at a time. Moreover, it requires to modify all parts of the application but also the simulator in our case.

C. User-defined Irrelevant Differences

Specific variables and memory areas can be explicitly ignored during the comparison. The user may leverage this to ignore the step number in a cyclic protocol, the value of an iterator in a loop or any value that he considers as irrelevant to its state. A different system behaviour resulting from the value of the counter will be detected in the other memory areas, while this counter could be ignored.

This information is stored in a new meta-data of our `malloc` implementation and used during the comparison. The same mechanism is used to mask irrelevant differences caused by the simulator internals, such as the total amount of messages sent during the simulation that is constantly incremented.

[2]http://www.dwarfstd.org/

D. Dynamic Semantic Comparison of Heap

The last and most difficult technical lock encountered during the system state comparison is due to the fact that the order in memory of `malloc`ed blocks rarely matters to the application semantic. Fig. 4 depicts two heaps that are semantically similar despite the block ordering. Blocks `0x30`, `0x40` and `0x50` are syntactically different, yet both heaps are semantically equivalent. All observed differences can be explained by the fact that the block `0x30` of one heap corresponds to the block `0x40` of the other heap.

Such situation often occurs in our context because the block ordering stems from the order of malloc requests, which change when the process execution order changes, as in the dynamic verification.

Moreover, the numerical value of a pointer remains unchanged in C when the pointed area is freed. As the system may reallocate the previously freed memory, the pointed data may be completely different. It is a good habit as a programmer to set the pointer variables to NULL in this case, but this is not requested by the C standards. The manual detection of these *dangling pointers* by the user is however essential to the soundness of our approach, as the comparison of unrelated memory areas leads to the non-detection of the system state equality.

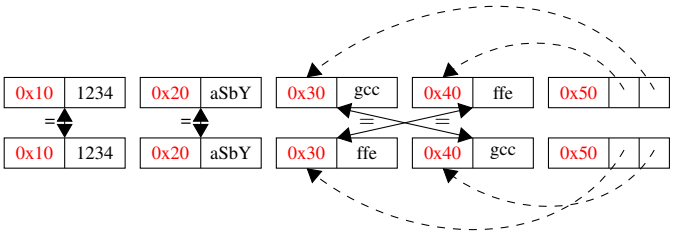

Fig. 4. Two heaps syntactically different but semantically identical.

Even if there is no dangling pointer, detecting the semantic equality between heaps in which blocks were reordered remains hard. We cannot directly leverage DWARF debugging information, as the considered memory is allocated during the execution and thus unknown at compile time when this information is produced.

In [14], the author presents a *heap canonicalization algorithm* that can reorder the blocks in a canonical form. It is then much easier to compare the heaps together. This approach relies on a garbage collecting mechanism in a language where all references to allocated data are clearly known to the system. A mark-and-sweep algorithm is then conducted from all variables down to the blocks. The graph traversal is deterministic by construction, and the blocks are reordered on-the-fly into a canonical form.

This approach cannot be applied as is to our context because we cannot assume that the verified application uses one of the existing garbage collector for C/C++. We can retrieve the

Issue	Heap solution	Stack solution
Overprovisioning	`memset 0`	Stack pointer detection
Padding bytes	`memset 0`	DWARF + libunwind
Irrelevant differences	Ignore explicit areas	DWARF + libunwind + ignore
Syntactic differences	Dynamic semantic comparison	N/A (sequential access)
Uninitialized data	`memset 0`	Stack cleaner by binary code modification

TABLE I. SUMMARY OF ISSUES AND SOLUTIONS FOR THE SYSTEM-LEVEL STATE EQUALITY DETECTION.

references located in local and global variables according to their DWARF signature, but we will probably miss some references located in heap blocks as we lack any semantic information on the content of these blocks. Because of these missing references, it is impossible to move memory blocks: any reference still pointing to an old block location would cause the application to abort immediately.

Yet, our heuristics compares states on the fly by traversing the heap, starting from the globals and locals down to the blocks it can reach with the pointers it knows. It does so until all blocks are matched (in which case the heaps are equivalent) or until an inexplicable difference is detected.

One extra difficulty in this traversal is that local variables may be hidden at some execution points, for example because another variable of this name exists in a closer lexical scope. Since we cannot consider all variables, and since we lack any semantic information on the heap, we can only design a heuristics, whose practical effectiveness is evaluated experimentally in the next section.

We perform a partial mark-and-sweep, starting from the variables we know and iterating on data for which we have the debugging information. For each variable that is a pointer to a block, we retrieve the datatype of the pointed data, and iterate. This approach allows to match the pointed blocks even if their numerical position in each heap does not match. This traversal stops in the case where all blocks can be traversed and matched, or because of a detected difference, or because of some information that is missing to further guide the traversal. That information may be missing because of hidden local values or because of `void*` datatypes hiding the real type of pointed data.

The blocks for which we lack any debugging information are then compared byte per byte. If we find a difference, we attempt to explain this as a pointer difference: in each heap, we read the aligned 8 bytes that contain the differing byte as a numerical value. If each value corresponds to the address of a valid block, the comparison iterates recursively on the designated blocks.

E. Uninitialized data

Uninitialized variables remain a problem as the memory area is allocated but the corresponding bytes are not specified. In the heap, we modified the `malloc` implementation in order to zero the allocated memory before handling it to the application. However, this approach does not work for local variables allocated on the stack.

In the stack, this problem is managed by zeroing the stack frame at the beginning of each function. This is currently implemented by modifying the assembly of the function generated by the compiler before giving it to the assembler: a script evaluates the size of the stack frame of each function by parsing its assembly code, and prepends to the function the assembly instructions that set to 0 that stack area.

F. Same-page-sharing snapshots

At each node of the execution graph, SimGridMC takes a full snapshot of the application state: the whole content of the application heap, its stacks and the sections containing its global variables is copied. Making a copy of the whole state of the application at each node of the graph rapidly exhausts the available memory when the number of states increases. In many applications, only a small part of the memory changes between consecutive states. In some applications 99% of the memory pages do not change between consecutive states.

As an answer, our snapshots are compacted by storing each memory page only once. When storing a new page, we detect whether a page of the same content is already stored by first computing a hash of the page content, and then comparing byte-per-byte the content with the already stored pages that have the same hash value. If the page is already stored, it is reused in the new snapshot to save memory.

We used the memory page granularity (4KiB on x86 and x86_64), but smaller chunks could be used to increase the data sharing between snapshots, at the cost of an increase of the metadata used to manage the chunks.

G. Summary

As a final optimization, we test first and foremost several efficient criteria such as the total amount of allocated blocks, allocated data, and stack sizes of all processes. This allows to eliminate many candidate states even before traversing the heaps.

Strictly speaking, this heuristics is not a heap canonicalization algorithm, as it does not change the order of blocks in place. Nevertheless, this memory introspection technique is not restricted to object-based programs. To the best of our knowledge, this constitutes the first known approach to detect the system state equality for programs that do not use garbage collections (see Section V). Moreover, the verification is performed on the actual program memory without using state vectors. It differs to methods based on state abstractions in which the system state is described in a first time according to some memory information then analysed. Despite its apparent simplicity, this heuristics proves efficient in practice. As shown in the next section, it makes it possible to actually verify and exhaustively explore legacy distributed applications.

IV. EXPERIMENTAL EVALUATION

This section evaluates our contribution through three sets of experiments which illustrate each motivating example of

Application	# P	Stateless exploration			Stateful exploration without same-page-sharing			Stateful exploration with same-page-sharing		
		# States	Time	Memory	# States	Time	Memory	# States	Time	Memory
bcasttest (C)	3	> 1.4 millions	> 27 h	0.38 GiB	5,160	18 min	38.5 GiB	5,160	31 min	1.02 GiB
bcastzerotype (C)	5	12,135,948	23 min 50 s	0.34 GiB	4,734	6 min 10 s	5.54 GiB	4,734	6 min 50 s	0.83 GiB
	6	> 200 millions	> 20 h	0.34 GiB	56,054	5 h 56 min	59 GiB	56,054	9 h 58 min	6.93 GiB
commcreate1 (C)	4	102,289	13 s	0.34 GiB	1,556	1 min 11 s	3.36 GiB	1,556	1 min 19 s	0.48 GiB
	5	12,710,034	25 min 37 s	0.34 GiB	8,559	18 min 52 s	10 GiB	8,359	20 min 30 s	1.42 GiB
	6	> 750 millions	> 27 h	0.33 GiB	99,235	21 h 55 min	103 GiB	99,235	25 h 54 min	13.7 GiB
dup (C)	2	907	2 s	0.34 GiB	105	2 s	0.47 GiB	105	2 s	0.34 GiB
	3	138,678	17 s	0.34 GiB	574	5 s	0.91 GiB	574	6 s	0.36 GiB
	4	78,082,843	3 h 42 min	0.34 GiB	3,058	1 min 44 s	3.67 GiB	3,058	1 min 54 s	0.7 GiB
groupcreate (C)	4	102,289	12 s	0.34 GiB	1,205	42 s	1.76 GiB	1,205	45 s	0.43 GiB
	5	12,710,034	24 min 31 s	0.34 GiB	6,237	9 min 56 s	7.26 GiB	6,237	11 min 03 s	1.16 GiB
	6	> 780 millions	> 27 h	0.36 GiB	80,878	16 h 03 min	85 GiB	80,878	17 h 38 min	11.1 GiB
inplacef (Fortran)	3	> 570 millions	> 27 h	0.37 GiB	22,223	25 min 37 s	27.22 GiB	22,223	28 min 35 s	3.50 GiB
op_commutative (C)	3	358	1 s	0.34 GiB	94	2 s	0.45 GiB	94	3 s	0.34 GiB
	4	102,289	12 s	0.34 GiB	1,545	1 min 10 s	2.36 GiB	1,545	1 min 18 s	0.47 GiB
	5	12,710,034	25 min 39 s	0.34 GiB	10,998	41 min 20 s	13.6 GiB	10,998	54 min 20 s	1.72 GiB
sendrecv2 (C)	2	> 270 millions	> 27 h	0.36 GiB	1,877	20 s	3.09 GiB	1,877	19 s	0.48 GiB

TABLE II. EXHAUSTIVE VERIFICATION OF MPI APPLICATIONS FROM MPICH3 TESTSUITE.

Section II. Section IV-A focuses on the formal verification of several applications distributed as part of the MPICH3 testsuite. Several point to point and collective communications patterns are exhaustively explored this way. Section IV-B then evaluates the tool's ability to detect a liveness violation on a custom MPI application. Section IV-C demonstrates how state equality detection enables the exhaustive exploration of cyclic applications whose behavior is regular but not bounded in time.

We used SimGrid (git version e214f3, 60,000 lines of code), as our contribution is integrated to the public version of this framework. These experiments were conducted on a Intel(R) Xeon(R) CPU E7540 @ 2.00GHz, RAM 512GiB, 48 cores, with debian wheezy environment 3.2.0-4-amd64 and 3 extra packages (cmake 2.8.9, libunwind7, and gfortran 4.7.2).

In all subsequent tables, '#P' is the total amount of processes in the studied application, '# States' corresponds to the number of expanded states before finding the counterexample (or the total number of states in the case of exhaustive exploration), and 'Depth' is the depth in which the counterexample has been found (only for the verification of a liveness property).

A. Verification of the MPICH3 testsuite

This first experiment focuses on the MPICH3 testsuite [15]. This testsuite allows to test any MPI implementation to be in compliance with the standards. Not all of the tested features are currently implemented in SimGrid so we eliminated the corresponding tests. We verified several applications (about 1,300 lines of code per application) from this testsuite, written in C or Fortran. Since our tool is written in C and C++ itself, these experiments demonstrate our ability to verify programs written in C/C++ or Fortran.

We looked for deadlock or livelock corresponding to non-progressive cycles. We managed to exhaustively explore the state space of these applications for up to 6 processes. No error has been found during these exploration. Table II clearly demonstrates that a stateful exploration reducing the state space is mandatory to perform an exhaustive verification on even basic tests with few processes. It also shows the effectiveness of our memory compaction approach, which reduce the memory consumption by a factor of ten on large scenarios while only slowing the exploration by 10%.

B. Dynamic verification of a liveness property

This experiment uses a custom MPI implementation of the centralized mutual exclusion algorithm (about 100 lines of code), where a coordinator grants infinitely often a mutex to the clients that request it. We introduce an error so that one of the clients never gets the requested critical section

# P	With same-page-sharing				Without same-page-sharing			
	# States	Time	Memory	Depth	# States	Time	Memory	Depth
3	64	2.97 s	0.33 GiB	57	64	2.40 s	0.48 GiB	57
4	301	3.26 s	0.35 GiB	278	301	3.46 s	1.2 GiB	278
5	1,041	9.97 s	0.6 GiB	949	1,041	11.37 s	3.4 GiB	949
6	4,494	33 s	1.9 GiB	4,116	4,494	48 s	14 GiB	4,116
7	25,097	9 min 12 s	11 GiB	23,207	25,097	8 min 40 s	83 GiB	23,207

TABLE III. STATEFUL VERIFICATION OF MPI BUGGED MUTUAL EXCLUSION EXAMPLE (INFINITE-TIME VERSION) WITH LTL PROPERTY $\Box(r \rightarrow \Diamond cs)$.

# P	With same-page-sharing			Without same-page-sharing		
	# States	Time	Memory	# States	Time	Memory
3	4,884	1 min	0.75 GiB	4,884	1 min 12 s	6.8 GiB
4	478,396	17 h 50 min	49 GiB	-	-	$>$ 300 GiB

TABLE IV. STATEFUL EXHAUSTIVE EXPLORATION OF MPI CENTRALIZED MUTUAL EXCLUSION ALGORITHM (INFINITE-TIME VERSION).

(its requests are discarded). We verify the following liveness property: $\Box(r \rightarrow \Diamond cs)$ (*any process that requests (r) it must obtain the critical section (cs)*). Since one client never gets the cs, this property is violated and a counterexample must be eventually found.

That counterexample can be found either after a few seconds if it is on the first explored branch, or after several hours if located in another branch. Table III presents the worst case results, when the buggy process is the last one. With such infinite-time applications, a stateful exploration is mandatory to detect and avoid non-progressive cycles that could prevent the verification from terminating.

C. Stateful exhaustive exploration of infinite-time application

This last experiment performs a stateful exhaustive exploration of an infinite-time MPI application (about 100 lines), without a property to verify. The goal is to benchmark an exploration of the whole state space, using the same centralized mutual exclusion algorithm than in previous section (without the bug introduced). The results presented in Table IV clearly show the usefulness of the memory compaction: without this mechanism, the exploration exhausts the available memory (300 GiB) for four processes already, while we managed to explore exhaustively the same example with this mechanism.

V. RELATED WORK

Since this work leverages dynamic analysis techniques, we will not discuss approaches based on static analysis; that is orthogonal. Both approaches could be used jointly to benefit both several sources of information during the verification.

Many tools exploit dynamic analysis either for the verification of C programs [16], [4], [17] with abstract interpretation in some cases and focused on memory-related errors [18], [19] or in particular for the termination analysis [20], [21]. This approach is also commonly used for Java [12] or object-based program analysis, with a garbage collection mechanism for some [22]. As explained in section III, it is easier to implement this approach for languages served by a virtual machine, as the meta-data known to the virtual machine are precious in this context. The semantic of any byte of memory is known to the VM, that can in particular retrieve and follow pointers. This operation is much harder in our case, especially with generic pointers and invisible variables. We must rely on a heuristic to deal with the possible memory leaks and dangling pointers, that cannot happen when using a garbage collector.

The need of formal methods for HPC applications is well acknowledged [23], but there exists few verification tools for MPI applications. To the best of our knowledge, MPI-CHECK [24] is the only verification tool of Fortran 90 MPI programs. Thanks to compile-time and runtime tests, it detects deadlocks and some inconsistencies in MPI calls such as negative message lengths. But the exploration is not exhaustive and the achieved tests are limited. Gauss [25] and MPI-Spin [26] are model extractors for MPI applications which can then be checked with Zing [27] and SPIN [28] respectively.

Finally, ISP [29] and its distributed counterpart DAMPI [30] are dynamic verifiers specifically tailored for the verification of MPI applications written in C. They check for deadlock and local assertions violations without requiring users to manually model their code. ISP hijacks the PMPI profiling interface that is normally intended for tracing tools to gather information about the MPI calls. ISP mediates these MPI calls and performs the dynamic verification at that level. A distributed protocol is then used between nodes to determine which messages should be delayed within the "profiling" call, and which ones can proceed (after being rewritten).

This approach leads however to two major drawbacks. First, collective operations are seen as atomic calls from the profiling interface perspective. The point-to-point communications that compose these collectives are completely invisible at this level. It means that unlike SimGridMC, ISP cannot properly verify collective operations, that must be assumed intrinsically correct. This seems unfortunate given the current momentum on asynchronous group collectives, that are known to be error-prone to implement [23]. Moreover, the real execution of the application on a real distributed platform may pose subtle challenges to ensure that the simulation is reproducible. Likewise, rewriting the MPI calls may change the synchronization semantic, that may depend on the buffer sizes in some border cases [31]. SimGridMC is based on the versatile SimGrid framework instead which is reproducible by design. It is much easier to observe and control the distributed system when it is folded into a unique system process as in SimGridMC, resulting in simpler and thus more robust setups.

VI. CONCLUSION AND FUTURE WORK

System state equality detection is essential for the dynamic verification of applications. It is important to verify arbitrary liveness properties, to explore exhaustively infinite cyclic protocols and constitutes an efficient reduction mechanism during stateful explorations. Detecting system state equalities is however much harder with legacy distributed applications than with abstract models.

In this paper, we detailed the root causes of these difficulties, and proposed various solutions leveraging debugging information and tools. To the best of our knowledge, it is the first solution to reconstruct the needed semantic information about systems that use programming languages without automatic garbage collection. We evaluated our implementation on several programs, including tests from the official MPICH3 testsuite. Our work is integrated in the SimGrid framework[3].

[3]SimGrid is freely available from http://simgrid.org/ (LGPL license).

Since the state space equality detection relies on a heuristic, our approach cannot be used for certification but only for bug finding. The improvement beyond prior work on verifying unmodified legacy HPC programs is many-fold: any safety or liveness property can be verified, and some class of infinite time-applications can be verified. These properties can be assessed on arbitrary C/C++ or Fortran mono-threaded applications based on MPI, using only the debugging symbols even if the source code is not available.

In the future, we want to combine our approach with other classical methods to increase the amount of available information during the reduction. For example, visibility information that can be extracted from instrumentation are mandatory to combine the DPOR and stateful reductions [32], and would perfectly complement the runtime system-state analysis that our approach provides. Memory graphs that can be reconstructed from static analysis would help extending the applicability of our approach to multithreaded MPI applications.

We also want to evaluate complex MPI code, such as the asynchronous collective calls implemented in MPICH and OpenMPI or even full MPI applications. Thanks to the SimGrid framework, the correctness and performance of these code could be evaluated jointly.

Acknowledgments

This work was partially funded by the ANR project SONGS (ANR-11-INFRA-13). Some experiments were carried out using the Grid'5000 experimental testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr/).

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *12th Intl Conf of Computer Aided Verif. (CAV)*, 2000.

[2] D. Engler, "Concur 2005 - concurrency theory," M. Abadi and L. de Alfaro, Eds. London, UK, UK: Springer-Verlag, 2005, ch. Static Analysis Versus Model Checking for Bug Finding.

[3] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Communications of the ACM*, Jul. 2011.

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to soft. engineering," *Int. J. Softw. Tools Technol. Transf.*, 2007.

[5] A. Gurfinkel, O. Wei, and M. Chechik, "Yasm: A software model-checker for verification and refutation," in *18th Intl Conf on Computer Aided Verif*, ser. CAV, 2006.

[6] S. Merz, M. Quinson, and C. Rosa, "SimGridMC: Verification Support for a Multi-API Simulation Platform," in *Intl Conf on Formal Tech for Net and Dist Sys*, 2011.

[7] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Dist. Experiments," in *Intl Conf on Computer Modeling and Sim.*, 2008.

[8] H. Saissi, P. Bokor, C. A. Muftuoglu, N. Suri, and M. Serafini, "Efficient verif of dist protocols using stateful model checking," *IEEE Symp on Reliable Dist. Sys.*, 2013.

[9] G. Holzmann, "An analysis of bitstate hashing," *Form. Methods Syst. Des.*, 1998.

[10] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Communications of the ACM*, pp. 88–98, May 2011.

[11] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building dist. sys." in *ACM Conf Prog.Lang. Design and Implem*, 2007.

[12] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," in *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[13] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *6th Conf on Computer Systems*, ser. EuroSys, 2011.

[14] R. Iosif, "Exploiting heap symmetries in explicit-state model checking of software," in *Proc. of 16th IEEE Conf on Automated Soft. Eng.*, 2001.

[15] MPICH. [Online]. Available: http://www.mpich.org/

[16] L. Mauborgne, "Astrée: Verification of absence of run-time error," in *Building the Information Society*. Springer, 2004.

[17] R. Bagnara, P. M. Hill, A. Pescetti, and E. Zaffanella, "On the design of generic static analyzers for modern imperative languages," *CoRR*, vol. abs/cs/0703116, 2007.

[18] K. Dudka, P. Müller, P. Peringer, and T. Vojnar, "Predator: a tool for verif. of low-level list manipulation," in *Tools and Algo for the Analysis of Systems*, 2013.

[19] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[20] S. Falke, D. Kapur, and C. Sinz, "Termination analysis of C programs using compiler intermediate languages," in *22nd Intl Conf Rewriting Tech. and Applications*, 2011.

[21] G. Andrieu, C. Alias, and L. Gonnord, "SToP : Scalable Termination analysis of (C) Programs (tool presentation)," in *Tapas 2012*, 2012.

[22] N. H. A. de Brugh, V. Y. Nguyen, and T. C. Ruys, "Moonwalker: Verification of net programs," in *Tools and Algo. for the Construction and Analysis of Systems*, 2009.

[23] G. Gopalakrishnan, R. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. de Supinski, M. Schultz, and G. Bronevetsky, "Formal Analysis of MPI-Based Parallel Programs: Present and Future," *Communications of the ACM*, 2011.

[24] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "Mpi-check: a tool for checking fortran 90 mpi programs," *CCPE*, vol. 15, no. 2, pp. 93–100, 2003.

[25] R. Palmer, S. Barrus, Y. Yang, G. Gopalakrishnan, and R. M. Kirby, "Gauss: A framework for verifying scientif comp soft," *Electr Notes in Theor Comp Sci*, vol. 144, no. 3, 2006.

[26] S. F. Siegel, "Using mpi-spin to model check mpi programs with nonblocking communication," *Recent Advances in PVM / MPI*, 2006.

[27] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *16th Intl Conf of Computer Aided Verif. (CAV)*, 2004.

[28] G. Holzmann, "The model checker spin," *IEEE Transac. on Soft. Engineering*, 1997.

[29] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical mpi programs," *SIGPLAN Not.*, vol. 44, no. 4, 2009.

[30] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for mpi programs," in *Conf of SuperComputing'2010*.

[31] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, and G. L. Lee, "Determinism and reproducibility in large-scale hpc systems," in *4th Works. Determinism and Correctness in Par. Prog. (WoDet)*, 2013.

[32] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby, "Efficient stateful dynamic partial order reduction," in *Model Checking Software*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5156.