Architecture-aware Partial Order Reduction to Accelerate Model Checking of Networked Programs

Cyrille Artho
RCIS/AIST
Tokyo, Japan

Watcharin Leungwattanakit
University of Tokyo
Tokyo, Japan

Masami Hagiya
University of Tokyo
Tokyo, Japan

Yoshinori Tanabe
CVS/AIST
Tokyo, Japan

Abstract

Testing cannot cover all execution schedules in concurrent software. Model checking, however, is capable of verifying the outcome of all possible executions. It has been applied successfully to networked software, with all processes being analyzed in conjunction. Unfortunately, this approach does not scale very well. This paper presents a partial-order reduction through which a performance gain of up to 70 % was achieved.

1. Introduction

Model checking explores, computational resources permitting, the entire behavior of a system under test by investigating each reachable system state [8], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [2, 3, 6, 9, 10, 12, 21]. While model checking has enjoyed some success in the analysis of single-process systems, analysis of multiple communicating processes remains difficult.

If nothing is known about the architecture of the applications analyzed, analysis has to include all applications. This can be achieved with a model checker that supports multiple processes [7, 12, 16, 17], or by transforming multiple processes into a single process [1, 4, 19]. Both approaches cover all possible communication behaviors but suffer from the state space explosion problem.

To combat the state space explosion, model checkers employ various state space reduction techniques. One of these is *partial-order reduction,* which ignores redundant interleavings between independent actions [8, 14]. In software model checkers, these techniques try to determine if certain program steps (transitions in the model) result in changes that only affect the current thread, but no other threads. A sequence of such steps can be executed atomically, without taking interleavings between different threads into account [5, 11, 21]. As partial-order algorithms have to be conservative (they must not eliminate interleavings producing different results), they do not always optimize the state space as much as theoretically possible.

In Java and many other widespread programming languages today, a thread has two roles: It is both an executable task and a data structure [13]. The thread data structure holds information such as thread name and ID, and can be extended with other data. A thread as a task constitutes a light-weight process that shares the global heap with other threads [20]. A thread cannot be executed directly, but merely be *enabled for execution.* Execution itself may therefore be delayed [13]. If several threads are spawned at the same time, interleavings between start actions and actions of previously enabled threads are possible. Because one cannot a priori assume that each thread can be started independently of other events, conservative partial-order reduction among such interleavings is not always possible. We propose such a reduction for cases where this independence holds. The reduction can be applied directly to centralized programs [1], making it useful for a large family of programs.

This paper is organized as follows: Section 2 motivates our partial-order reduction for thread startup. Its implementation is shown in Section 3. Section 4 lists experimental results, and Section 5 concludes.

2. Elimination of Thread Interleavings

Parallel programs often delegate tasks to worker threads. A similar structure also exists in applications

where several processes have been merged ("centralized") into a single application. Such a transformation wraps processes as threads [19], and is used to model check networked programs [1]. A direct implementation of wrapping allows for interleavings between execution of client threads and the code of the wrapper thread that starts each client. This paper presents a partial-order reduction which, when applied to such programs, eliminates exploration of such interleavings.

In Java [13] and other widespread programming languages, execution of a child thread is enabled when the parent thread calls a special method, such as `start`. In this paper, we will refer to this as the *spawning* of the target thread. After spawning, a child thread is ready to run. Execution of the child thread code (the `run` method in Java) as a separate task may begin after an arbitrarily long delay [13]. Other threads may execute any number of actions in between. Such interleavings can often be ignored during model checking.

Many algorithms assign a subset of the entire problem to a worker thread, and collect aggregate results after computation. If individual subsets of data do not overlap, worker threads can be spawned independently of whether other worker threads are already executing. A similar property is inherent in many server architectures, where a master thread delegates requests to different worker threads. In such an architecture, actions that spawn individual workers threads are often completely independent of each other. This independence allows programmers to dispense of locking at that point, even if subsequent computation uses global data that requires lock protection. If a number of threads is spawned in a loop, independently of other events, then the sequence of spawnings can be considered to be atomic.[1] This allows for substantial state space reductions in model checking. However, the (safe) absence of mutual exclusion locks usually prevents on-the-fly heap analysis algorithms from safely deducing that certain instruction sequences can be executed independently of each other. Partial-order reduction based on heap reachability often falls short because intrinsic algorithmic properties (such as disjoint index ranges or other mechanisms making potentially unsafe access safe) are not recognized. It is here where architecture-specific partial-order reductions can enhance existing (generic) optimizations. In our case, the optimization is specific to software model checkers, and programming languages using a POSIX-like thread model in particular. Our optimization requires only that a loop that spawns multiple threads is independent of other actions.

Related work in partial-order reduction for Java-like programs includes algorithms that reduce interleavings based on lock synchronization [5], and reductions based on reachability information from which thread-local data access can be inferred [21]. On a more abstract level, if perfect points-to and alias information is available, threads can automatically be summarized by an environment model, allowing each thread to be checked locally [11]. In our experiments, general algorithms failed to infer the information necessary, requiring architecture-specific enhancements to improve symmetry reduction.[2]

Sound application of our specific optimization requires certainty about the program to be analyzed. If assumptions about an algorithm are incorrect, or its implementation has an unknown flaw, then it is often impossible to detect the false assumption at run-time, and program analysis may be unsound (possibly overlooking flaws in the software).

This limits the applicability of specific optimizations. Indeed, in the general case, usage of such optimizations has to be complemented by prior analysis of the program (such as pointer escape analysis). However, unconditionality of thread spawning can be shown relatively easily at compile time. Furthermore, this assumption always holds for one specific kind of application: An application resulting from merging several processes into a single process by a transformation called centralization [1, 19]. This transformation uses wrapper threads to execute individual processes. Spawnings of wrapper threads are independent of each other. Therefore, when pruning redundant interleavings during model checking, significant gains are possible. As the transformation can be applied to any multi-process system implemented in Java, our proposed partial-order reduction is widely applicable.

3. Implementation

Centralized applications spawn independent worker threads that encapsulate client processes. Our partial-order reduction exploits this architecture to prune redundant paths.

[1] Note that the `run` methods of different worker threads can still be interleaved arbitrarily!

[2] Analysis of heap isomorphism is NP-complete and can therefore not be performed exhaustively for the purpose of partial-order reduction. In the contrary, the overhead of symmetry analysis should to be smaller than the gain from the resulting state space reduction. This precludes complex symmetry analyses.

3.1. Application Centralization

Centralization [19] allows to model check multiple processes in a single-process model checker, by wrapping several processes in a single process. Using a TCP/IP model library, networked applications can then be model checked [1]. Wrapper threads encapsulating application processes are derived from class `CentrProc`.[3] Wrapper threads behave like normal threads, except that their data structure is augmented by a process ID field. This process ID is used to distinguish the original processes. In the resulting transformed application, the main thread is just a wrapper for applications and spawns the original processes. The wrapper first spawns the server process as a separate thread (lines 5–7), and waits for it to be ready for client requests (lines 9–15).[4] After that, client processes are initialized and spawned, again within wrapper threads. Figure 1 shows an example using one server process and several client processes.[5] The code shown in Figure 1 generates two anonymous inner classes [13] of `CentrProc`, each with their own `run` method (lines 5–7 and 18–20). The constructor of these process wrapper classes (not shown) stores the process ID.

In the centralized application, spawning of client threads (lines 22 – 24) may be interleaved with execution of previously spawned threads. The built-in partial order reduction fails to recognize this redundancy. Figure 2 illustrates this problem on an example with two threads. The main thread executes two actions: `start1` and `start2`. Without any restrictions, `run1` may already execute prior to `start2`. After `start2`, the `run` methods of both client threads are eligible to run. Because `start2` is independent of `run1`, the schedules on the left hand side of Figure 2 produces the same result as the schedule on the right hand side.

3.2. Semaphore-based Implementation

Centralization as implemented in recent work [1] already includes one semaphore, which is used to prevent client processes from connecting to the server before the server is able to accept requests (lines 9 –15 in Figure 1). Premature connections would result in spurious error traces [1].

```
 1 public class Wrapper extends Thread {
     public static final void main(...) {
       CentrProc[] p = new CentrProc()[N];
       new CentrProc(0) {
 5       public void run() {
           new Server(server_args);
       }}.start();
       // wait for server to be ready
       try {
10       synchronized (Socket.port) {
           while (!Socket.port.isOpen) {
             Socket.port.wait();
           }
         }
15     } catch (InterruptedException e) { }
       for (int i = 1; i <= N; i++) {
         p[i] = new CentrProc(i) {
           public void run() {
             Client.main(client_args);
20       }};
       }
       for (int i = 1; i <= N; i++) {
         p[i].start();
   } } }
```

Figure 1. Application centralization.

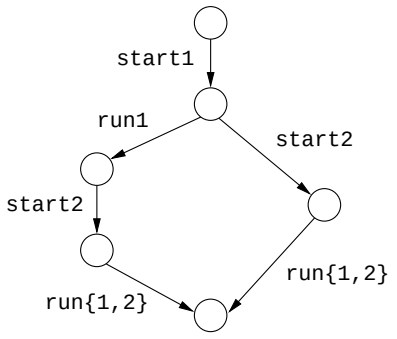

Figure 2. Possible transition schedules.

Our first attempt was to use similar code to avoid premature execution of client processes. The key changes in the code (in lines 16–24 in Figure 1) are shown in Figure 3. In the modified version shown in the center, a new flag `wrapperDone` (initial set to `false`) is added, which is set to `true` by the main (wrapper) thread after its termination. This ensures that all instances of `CentrProc` are ready to run at this point, but their `run` methods have not yet progressed past the initial (instrumented) statement.[6]

[3] Short for `CentralizedProcess`.

[4] The socket used here is a socket model class used for centralization [1]. Semaphores in Java are implemented by using a boolean flag in conjunction with `wait`/`notify` [13, 15].

[5] For brevity, parameters to `main` have been omitted. The aspect of centralization discussed here only covers execution of processes as threads. Several other program transformations are also required to preserve the semantics of the original program [1, 19].

[6] The full Java code also requires lock usage and a `try`/`catch` clause for wait, see Figure 1 and [13].

```
for (int i = 1; i <= N; i++) {
  p[i] = new CentrProc(i) {
    public void run() {

      Client.main(args);
  }};
}
for (int i = 1; i <= N; i++) {
  p[i].start();
}
```

```
for (int i = 1; i <= N; i++) {
  p[i] = new CentrProc(i) {
    public void run() {
      if (!wrapperDone) wait();
      Client.main(args);
  }};
}
for (int i = 1; i <= N; i++) {
  p[i].start();
}
wrapperDone = true;
notifyAll();
```

```
for (int i = 1; i <= N; i++) {
   p[i] = new CentrProc(i) {
    public void run() {
      Verify.ignoreIf(!wrapperDone);
      Client.main(args);
  }};
}
for (int i = 1; i <= N; i++) {
  p[i].start();
}
wrapperDone = true;
```

Figure 3. Optimization. Left: Original wrapper code to execute a client process; center: Optimization using `wait`/`notify`; right: Optimization using model checker API functions.

3.3. API-based Implementation

The first implementation manages to delay execution of clients until all clients have been spawned. Unfortunately, it falls short of achieving the desired state space reduction. The `wait` statement (see Figure 3) does not eliminate all interleavings but simply suspends each client thread until the wrapper thread has terminated. Despite the synchronization, it is still possible to execute all interleavings of the main thread and the first line of the modified `run` method. At that point, the state of all client processes is equivalent (isomorphic) under different interleavings. However, the model checker does not recognize this, and continues to explore several "copies" of equivalent states. Furthermore, the overhead of `wait`/`notify` actually increases the state space.

The desired solution prunes the path on the left hand side of Figure 2. While the Java language does not contain a mechanism to achieve this, model checkers have internal mechanisms to ignore states that are considered redundant. The Java PathFinder model checker [21] (JPF) offers an API method `ignoreIf`, which ignores a state if a given condition holds. Our partial-order reduction uses a flag to suppress all paths other than one where the wrapper thread finishes first. No restriction is imposed on the interleavings of the body of the `run` methods (after the instrumented code). Therefore, the full state space of the centralized child processes is still explored. The resulting code is shown on the right hand side of Figure 3.[7] Note that the original semaphore that waits for the server to be ready to accept requests can be replaced with a similar call to `ignoreIf`, but the improvement in performance is negligible. The check cannot be eliminated, as completion of the wrapper code does not entail readiness of the server.

[7] Data races on flag `wrapperDone` are prevented by declaring it as `volatile` [13].

3.4. Final Implementation

Other implementation approaches are possible. JPF offers an API to check whether the `main` thread, used to spawn child threads, is still running. Instead of a flag, this function can be used to ignore child threads until the `main` thread has terminated. We have tried several other implementation approaches in addition to the ones shown here, and came up with a final version that improved efficiency further by minimizing the overhead of the partial-order reduction. These final refinements improved efficiency by up to another 18 %. Figure 4 shows the full original wrapper code and the final optimized version. The code changes in lines 15–24 concern the partial-order reduction:

1. The call to `CentrProc.start` is performed just after the constructor has been called. This dispenses with the need of keeping a reference to all child threads to be started. The fact that this does not reduce performance seems to be somewhat counter-intuitive, as it seems to allow interleavings between thread initializations and thread spawnings. Nonetheless, in our experiments, the model checker still did not generate any redundant interleavings, as long as the abovementioned barrier (flag `wrapperDone`) was used. We assume that the internal partial-order reduction of JPF recognizes the redundancy between interleavings of different thread spawnings and initializations. In isolation, though, this change achieves very little.

```
 1 public class Wrapper extends Thread {
     public static final void main(...) {
       CentrProc[] p = new CentrProc()[N];
       new CentrProc(0) {
 5       public void run() {
           new Server(server_args);
       }}.start();
       // wait for server to be ready
       try {
10       synchronized (Socket.port) {
           while (!Socket.port.isOpen) {
             Socket.port.wait();
           }
         }
15     } catch (InterruptedException e) { }
       for (int i = 1; i <= N; i++) {
         p[i] = new CentrProc(i) {
           public void run() {
             Client.main(client_args);
20       }};
       }
       for (int i = 1; i <= N; i++) {
         p[i].start();
   } } }
```

```
 1 public class Wrapper extends Thread {
     public static final void main(...) {
       static volatile int nProc = 0;
       new CentrProc(0) {
 5       public void run() {
           new Server(server_args);
       }}.start();
       // wait for server to be ready

10


       Verify.ignoreIf(!Socket.port.isOpen);


15     for (int i = 1; i <= N; i++) {
         new CentrProc(i) {
           public void run() {
             Verify.ignoreIf(nProc != pid);
             ++nProc;
20           Client.main(client_args);
         }}.start();
       }
       nProc = 1;
   } }
```

Figure 4. Application centralization: Original code (left) and final optimized version (right).

2. A counter instead of a flag is used to eliminate redundant interleavings of thread spawnings. This counter is compared to the process ID field (`pid`) of each centralized process. Counter `nProc` imposes a total order on thread spawnings. This method only allows a single thread schedule to pass the barrier. It is stricter than the flag-based version, which eliminates redundant interleavings between thread initializations and spawnings, but permits multiple interleavings at the beginning of the `run` method. Note that interleavings of centralized processes are not restricted by this optimization: Calls to `Client.main` can still be made in any order.

As mentioned earlier, the initial check against a premature client startup (lines 9–15 in the original code) can also be streamlined using the JPF API (line 12 in the optimized code), eliminating lock usage.

For larger instances, the final optimized version has shown to be about 15 % more efficient than the previous optimization. In small instances, a performance loss of up to 3 % was seen, compared to the implementation using a flag. This shows how small variations in the implementation of the same design can have a large impact when model checking a program.

Our approach to partial-order reduction is generic; however, it requires usage of specific features of the model checker. The actual code therefore requires minor adaptations for each model checker. When our partial-order reduction is used on centralized processes, the centralization tool itself does not require changes. Thread spawning code is contained in a code template, which can be customized without changing the centralization tool [1]. Other program transformations are orthogonal to thread spawning, and not affected by our optimization.

4. Experiments

We used four example benchmarks to test our approach. The *daytime client* connects to a server, which sends a date string back to the client. Multiple clients initiate concurrent requests.

HTTP refers to a scenario modeled after Jget [18]. Jget is a multi-threaded download client, which issues a number of concurrent partial download requests in addition to the main request. Depending on which task finishes first, Jget either uses the entire file downloaded by the main thread, or it assembles the file from the pieces returned by the partial downloads. Essentially, the worker threads are in a (controlled) race condition against the main thread. This creates the challenge of ensuring that the complete file is received when the program shuts down. As the original program was too complex for model checking using centralization, we created a new version specifically for this case study. In this version, a simpler centralization mechanism that does not require process IDs could be employed, exploiting application-specific properties such

as the lack of shared data in `static` fields [1, 19]. In addition to an abstract HTTP protocol, the code also features a simplified socket mechanism. The simplification consists of closing the model *server* socket (the open port) on the client side instead of the server side. This change allowed us to eliminate session management code in the socket abstraction, making the system small enough for model checking.

HTTP features a single-threaded web server. Because the model checker still accounts for all possible delays in the responses from the server, the state space explored in the client code is equivalent to what is encountered when using a genuine, concurrent web server. Such a concurrent web server was used (with the same concurrent client) in case study *HTTP2*. Both HTTP and HTTP2 include only a single client process. Even though the client process utilizes multiple threads, these threads are not spawned independently of each other. These two test cases show the overhead of our instrumentation in the case when the partial-order reduction is ineffective because it cannot be applied to multiple calls to `start`.

The *chat* server, described in more detail in [1], sends the input of one client back to all clients, including the one that sent the input. The architecture is modeled after existing servers and uses worker threads to serve each client. The size of a test scenario can be varied by an optional limit on the number of client connections allowed by the server.

All experiments were run on an Intel Core 2 Duo Mac 2.33 GHz with 2 GB of RAM, running Mac OS 10.4.11. JPF version 4, revision 353 was used, with 1 GB of memory. The standard properties used by JPF were verified: We checked against deadlocks, uncaught exceptions, and assertion violations. No program contained a critical error that would have terminated the state space search by JPF and resulted in an error message. JPF therefore investigated the full state space, allowing for a comparison of the full state space explored without and with our partial-order reduction.[8]

Table 4 compares the original centralized code with our optimized centralization. As can be seen, our partial-order reduction is very effective for fully centralized programs containing multiple embedded client processes. The state space could be reduced by up to 70 %. Several larger benchmarks that could previously not be analyzed can now be explored fully. HTTP and HTTP2 constitute the worst-case scenario where only one client thread is spawned. In these cases, our partial-order reduction cannot reduce the size of the state space further and adds an overhead of about 20 %. We consider this overhead acceptable compared to the possible gains, and due to the fact that it is known a priori how many client processes are tested. Our partial-order reduction requires no manual transformation and is applicable to any centralized program [1].

[8] Faulty versions result in very short error traces [1], with or without using our partial-order reduction.

5. Conclusions and Future Work

Model checking explores the outcome of all possible thread and communication schedules in concurrent software. However, analysis of complex systems suffers from the state space explosion problem. Partial-order reductions are needed to eliminate redundancy.

We have proposed a partial-order reduction mechanism that successfully avoids interleavings between independent thread spawnings, exploiting architectural properties of an application. The required properties always hold for centralized applications, where multiple processes have automatically been transformed into a single process, allowing for automated use of our optimization. The resulting speed-up is considerable in any cases where several threads are spawned at the same time. Future work includes investigating the applicability of similar transformations to different families of programs.

References

[1] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *ASE 2006*, pages 177–188, Tokyo, Japan, 2006. IEEE Computer Society.

[2] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *CAV 2004*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[3] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *TACAS 2001*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.

[4] E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *ASE 2007*, pages 24–33, New York, USA, 2007. ACM.

Application	# conn./ threads	Limit in # of accepted connections	Original centralization			Optimized centralization		
			Time	States		Time	States	
			[hh:mm:ss]	new	revisited	[hh:mm:ss]	new	revisited
Daytime	2	n/a	0:57	33123	71608	0:23	15366	25957
	3		55:39	715941	4726181	22:32	604527	1511117
	4		out of memory after 40 hours			18:24:28	20823249	69873373
HTTP	2	n/a	0:08	12677	23249	0:09	15470	29259
	3		6:20:42	25444563	86025235	7:55:30	31993671	110615518
HTTP2	2	n/a	0:30	41937	95622	0:36	51305	121403
	3		out of memory after 10 hours			out of memory after 10 hours		
Chat	2	1	2:12	50284	143439	0:53	23624	55882
		2	77:08	714830	2668937	21:12	289568	933718
	3	1	89:28	1819345	6786956	33:32	742474	2395318
		2	out of memory after 71 hours			30:16:23	20487094	84792286
	4	1	out of memory after 38 hours			25:36:20	27370449	111955885

Table 1. Results of our experiments.

[5] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[7] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *ICSE 2002*, pages 431–441, New York, USA, 2002. ACM.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[9] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[10] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *CAV 2005*, volume 3576 of *LNCS*, pages 148–152, Edinburgh, UK, 2005. Springer.

[11] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 2003*, volume 2648 of *LNCS*, pages 213–224, Portland, USA, 2003. Springer.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL 1997*, pages 174–186, Paris, France, 1997. ACM Press.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[14] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[15] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[16] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[17] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multi-process applications using SBUML and GDB. In *Proc. Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.

[18] S. Paredes. jget, 2006. `http://www.cec.uchile.cl/~sparedes/jget/`.

[19] S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *SPIN 2001*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.

[20] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.

[21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.