Platform-Specific Restrictions on Concurrency in Model Checking of Java Programs

Pavel Parizek and Tomas Kalibera

Distributed Systems Research Group, Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University in Prague
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{parizek,kalibera}@dsrg.mff.cuni.cz

Abstract. The main limitation of software model checking is that, due to state explosion, it does not scale to real-world multi-threaded programs. One of the reasons is that current software model checkers adhere to full semantics of programming languages, which are based on very permissive models of concurrency. Current runtime platforms for programs, however, restrict concurrency in various ways — it is visible especially in the case of critical embedded systems, which typically involve only a single processor and use a threading model based on limited preemption. In this paper, we present a technique for addressing state explosion in model checking of Java programs for embedded systems, which exploits restrictions on concurrency common to current Java platforms for such systems. We have implemented the technique in Java PathFinder and performed a number of experiments on Purdue Collision Detector, which is a non-trivial multi-threaded Java program. Results of experiments show that use of the restrictions on concurrency in model checking with Java PathFinder reduces the state space size by an order of magnitude and also reduces the time needed to discover errors in Java programs.

Key words: model checking, Java programs, embedded systems, state explosion, restrictions of concurrency

1 Introduction

Software for mission- and safety-critical systems is typically based on multi-threaded programs, since such systems must be able to process concurrent inputs from their environment in a timely manner. This is the case, for example, of software in control systems in vehicles (cars, aircrafts) and software running on high-availability server systems. A significant part of the development process of programs for critical systems is devoted to testing and verification, since runtime errors in such programs are very costly. The errors particularly relevant for multi-threaded software used in critical systems are violations of temporal safety and liveness properties and also concurrency errors (e.g., deadlocks and race conditions). The verification technique most suitable for detection of such errors is model checking. A model checker systematically traverses the whole state space of a given program with the goal of detecting property violations and errors.

The main limitation of model checking is that it does not scale to complex multi-threaded programs, which are often used in critical systems, due to the well-known problem of *state explosion*. The state space of a program, which a model checker has to explore, captures all possible sequences of thread scheduling choices and all possible threads interleavings that can occur during program's execution — the size of the state space depends roughly exponentially on the number of threads. Although many techniques for addressing state explosion have been designed and implemented in various model checkers over the years [17], state explosion still occurs in model checking of large and complex multi-threaded programs written in mainstream programming languages. One of the reasons is that semantics of mainstream programming languages (e.g., Java, C and C#) are based on very permissive models of concurrency. For example, semantics of such languages allow preemption to happen at any program point and impose no restrictions on thread scheduling algorithms (e.g., any runnable thread can be scheduled when another thread is suspended). Model checkers for programs in such languages [24, 7, 2] then have to adhere to the full semantics of the languages for the sake of generality — in particular, the model checkers have to systematically check the behavior of programs under all possible thread scheduling sequences allowed by the semantics of the programming languages, including those thread scheduling sequences that cannot happen in practice due to concurrency-related characteristics of runtime platforms for programs.

We propose to address state explosion in model checking by exploiting restrictions on concurrency that are based on characteristics and behavior of runtime platforms formed by hardware, operating systems and, in case of languages like Java and C#, also by virtual machines. The goal is to reduce the state space size of multi-threaded programs such that the chance of traversing the whole state space of such programs in limited memory and reasonable time, and thus the chance of discovering more errors, is much greater.

To be more specific, the main contributions of this paper are:

- a technique for efficient model checking of multi-threaded Java programs, which is based on platform-specific restrictions of concurrency,
- an implementation of the technique as an extension to the Java PathFinder model checker (JPF) [24], and
- evaluation of the technique on Purdue Collision Detector, which is a non-trivial multi-threaded Java program.

We focus on model checking of Java programs, since Java is used for implementation of software for many critical server-side systems and is also becoming a language of choice for implementation of multi-threaded software for critical embedded systems.

2 Current Platforms for Java Programs

In this section, we provide an overview of concurrency-related characteristics and behavior of current runtime platforms for Java programs from the perspective of

model checking with JPF and we also define terminology that is used throughout the rest of the paper.

We consider a runtime platform for Java programs, further denoted as a *Java platform*, to be a specific combination of a hardware configuration, an operating system (OS), and a Java virtual machine (JVM). The key concurrency-related characteristics of Java platforms are:

- the maximal number of Java threads that can run in parallel,
- a threading and scheduling model that determines the level, at which threads in Java programs (Java threads) are implemented and scheduled, and
- a set of program points (Java bytecode instructions), at which a running thread may be suspended (preempted) by a platform — such program points are further referred to as *thread yield points*.

The maximal number of threads that can possibly run in parallel is bounded by the number of processors in a particular hardware configuration, and can be further limited by JVM — some processors can be dedicated to garbage collection or non-Java tasks, or the JVM can explicitly support only a single processor. In this text, we use the term *processor* to denote a logical processing unit of any kind, including a single processor (CPU) in a multi-processor machine or a single core in a multi-core CPU. Threading and scheduling models are typically implemented by the operating system and/or the JVM. The scheduling models implemented in most of the current Java platforms are based on preemption. The platforms differ in the set of thread yield points, i.e. in the set of program points where a running thread can be suspended, and also in the kind of preemption that they use — some use *time preemption*, in which case a thread can be suspended at any program point (i.e., all program points are considered as thread yield points in that case), while other platforms use limited preemption, in which case only specific Java bytecode instructions and calls of specific methods are considered as thread yield points.

The specifications of the Java language [9] and JVM [15] define a very permissive model of concurrency with respect to the characteristics listed above. In particular, (i) they do not put any restrictions on the maximal number of Java threads running in parallel, (ii) they do not specify any particular threading and scheduling model that should be used, and (iii) they allow threads to be preempted at any bytecode instruction and at call of any method, i.e. they allow the set of thread yield points to be defined as an arbitrary subset of the set of all program points. On the other hand, the current Java platforms restrict the concurrency in Java programs in various ways. The platforms can be divided into the following two groups depending on the way they restrict concurrency: (1) Java platforms for embedded systems and (2) Java platforms for server and desktop systems.

2.1 Java Platforms for Embedded Systems

Typically, Java platforms for embedded systems use the *green threading* model, hardware configurations in such platforms are based on a single processor, and

the set of thread yield points contains only a subset of all program points. The key idea behind green threading is that Java threads are managed and scheduled by a JVM (at the level of JVM), out of control of the underlying operating system. All Java threads in a program are mapped to a single native (OS-level) process in which the JVM runs (Fig. 1a), and therefore only a single thread can run at a time — the threads in a program are effectively interleaved. The green threading model is supported, for example, by Purdue OVM [3], which is a research JVM aiming at embedded and real-time systems, and also by the CLDC HotSpot Implementation [5], which is an industrial JVM for embedded devices compliant with the Java ME CLCD profile (e.g., mobile phones). Java platforms in this group also do not use scheduling based on time preemption. The platforms typically consider as thread yield points only those program points that correspond to one of the following actions: acquiring and releasing of locks (monitors), calls of blocking native methods (I/O), calls of selected methods of the `Thread` class (e.g., `sleep`, `start`, and `yield`), calls of the `wait` and `notify` methods, and, in case of more complex JVMs, also method invocations (method prologues), returns from methods (method epilogues), and back-branches (back-jumps to the beginning of a loop). For example, Purdue OVM considers back-branches as thread yield points, while the CLDC Hotspot Implementation does not. Selection of program points to be used as thread yield points is motivated mainly by performance and implementation reasons — the goal is to ensure that all threads get a fair share of processor time and also to allow easier implementation of JVM.

Although most embedded systems use only a single processor, there are also some embedded systems that employ multiple processors and therefore can run multiple threads in parallel. In case of such systems, either native threading model (explained in Sect. 2.2) or *quasi-preemptive* threading model can be used. The quasi-preemptive threading model is a generalization of green threading to multiple processors, which supports mapping of Java threads to N native (OS-level) processes such that $N \geq 2$ — the native processes may run truly in parallel and therefore also Java threads mapped to them may run truly in parallel. Still, Java threads are suspended and scheduled only at specific program points enumerated above (at the same program points as in the green threading model). A specific variant of the quasi-preemptive threading model is implemented in the Jikes Research Virtual Machine (RVM) [14].

An important subclass of embedded systems is formed by real-time systems, which must satisfy temporal constraints (e.g., meeting all deadlines). The key characteristic of real-time systems is that thread schedulers strictly enforce thread priorities. In case of non-real-time systems, thread priorities are not strictly enforced; however, threads with higher priorities should be in general scheduled more likely compared to threads with lower priorities.

To summarize, the key restrictions of concurrency in Java platforms for embedded systems are: (i) a bound on the number of threads that can run in parallel, which is determined by the number of processors, (ii) use of green threading model such that only specific bytecode instructions and calls of specific methods

are considered as thread yield points, and, in case of real-time systems, (iii) strict enforcement of thread priorities.

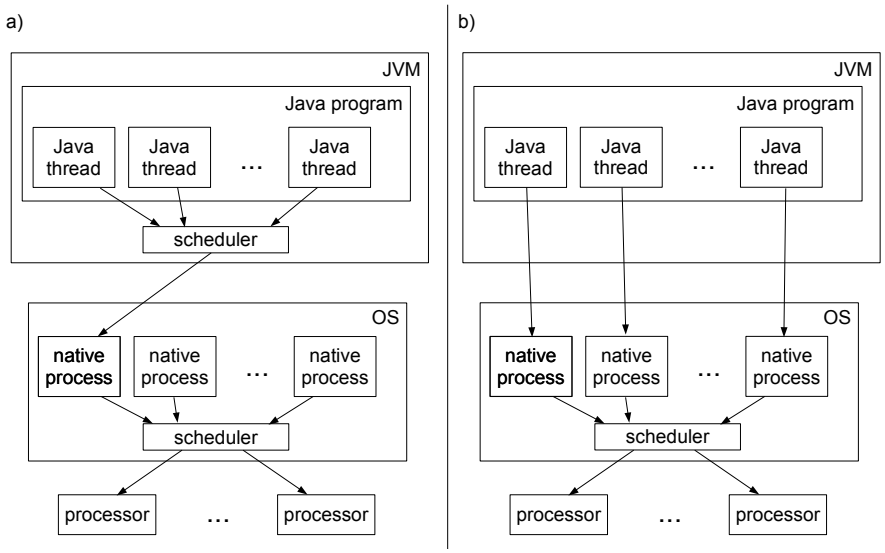

Fig. 1. Threading models in Java platforms: a) green threads; b) native threads

2.2 Java Platforms for Server and Desktop Systems

Java platforms for server and desktop systems use *native threading* model, and the hardware configurations in such platforms are typically based on multiple processors. In the case of a native threading model, Java threads are directly mapped to native (OS-level) threads that are scheduled by the OS-level scheduler (Fig. 1b); therefore, Java threads can run truly in parallel if multiple processors are available in a Java platform. Moreover, since schedulers in current operating systems use time preemption, the Java threads can be suspended at any program point (i.e., all program points have to be considered as potential thread yield points). It follows that the only important restriction of concurrency used in Java platforms for server and desktop systems is the bound on the number of threads that can possibly run in parallel.

Note that most industrial JVMs use native threading and time preemption-based scheduling — in particular, this applies to Sun Java Hotspot [23] and IBM J9 [12], which are both state-of-the-art industrial JVMs for desktop and server-side Java applications.

3 Running Example

The concepts and ideas presented in the paper will be illustrated on the Java program shown in Fig. 2, which is an instance of the producer-consumer design pattern. We selected the producer-consumer design pattern, since it forms the basis of many multi-threaded programs for critical systems (both embedded and server-side), including Purdue Collision Detector (PCD) that we use for evaluation of the proposed technique (details in Sect. 6).

4 Java PathFinder

Java PathFinder (JPF) [24] is an explicit state model checker for Java bytecode programs, which is based on a special Java virtual machine (JPF VM) that supports backtracking and state matching. It is highly extensible and configurable, and supports common optimizations of state space traversal like partial order reduction and thread/heap symmetry reduction.

The key features of JPF in the context of this paper are that (i) it implements the concurrency model of Java with no restrictions (i.e., it adheres to the full semantics of Java) and (ii) it does not model real time. To be more specific, JPF checks the behavior of a given Java program under the following assumptions:

- an unlimited number of processors is available,
- time preemption-based scheduling with an arbitrary (and dynamically changing) size of time slots is used, and
- all program points are considered as potential thread yield points.

The contention of multiple threads for shared data (variables) under such a model of concurrency is captured in JPF by systematic exploration of all interleavings of concurrent accesses to shared variables that are performed by individual threads. Technically, JPF suspends threads also at Java bytecode instructions corresponding to accesses to shared variables in addition to thread yield points, and selects the thread to be scheduled from the set of all runnable threads (including the one that was just suspended, if it is not blocked) at each thread scheduling choice point — this means that when a thread performs an access to a shared variable and is then suspended, any runnable thread may execute the next access to a shared variable. This is sufficient to capture the contention for shared variables both in case of threads running concurrently on a single processor, which are effectively interleaved, and also in case of threads running truly in parallel (on multiple processors), since parallel accesses to shared memory are actually serialized in the hardware (a particular interleaving of the parallel accesses is non-deterministically selected by hardware).

The actual state space of a given Java program is constructed by JPF on-the-fly in a way that reflects the concurrency model described above and the supported optimizations of state space traversal. A transition is a sequence of bytecode instructions performed by a single thread, which is terminated either by a scheduling-relevant instruction (thread yield point or an access to a shared

```
1  class Consumer extends Thread {
2    private Object[] buffer;
3
4    public void run() {
5      int pos = 0;
6      while (true) {
7        synchronized (buffer) {
8          while (buffer[pos] == null) buffer.wait();
9        }
10
11       Object msg = buffer[pos];
12       buffer[pos] = null;
13       pos++;
14       synchronized (buffer) {
15         buffer.notify();
16       }
17     }
18   }
19 }
20
21 class Producer extends Thread {
22   private Object[] buffer;
23
24   public void run() {
25     int pos = 0;
26     while (true) {
27       synchronized (buffer) {
28         while (buffer[pos] != null) buffer.wait();
29       }
30
31       // code that creates a message is omitted
32       buffer[pos] = msg;
33       pos++;
34       synchronized (buffer) {
35         buffer.notify();
36       }
37     }
38   }
39 }
40
41 public static void main(String[] args) {
42   Object[] buffer = new Object[10];
43   Consumer cons = new Consumer(buffer);
44   Producer prod = new Producer(buffer);
45   cons.start();
46   prod.start();
47 }
```

Fig. 2. Example Java program: producer-consumer pattern

variable) or by an instruction corresponding to non-deterministic data choice (e.g., use of a random generator). A state in the state space of a Java program is a snapshot of the current state of the JPF VM at the end of a transition, including complete heap and stacks of all threads.

Given the Java program on Fig. 2, JPF may terminate a transition at any instruction corresponding to the following program points:

- accesses to the `buffer` variable in all threads (source code lines 8, 11, 12, 28, and 32),
- attempts to acquire a monitor, i.e. attempts to enter a `synchronized` block (lines 7, 14, 27, and 34),
- calls of the `wait` and `notify` methods (lines 8, 15, 28, 35), and
- start of a new thread (lines 45 and 46).

JPF provides a powerful API that allows to extend it in various ways. With respect to the technique proposed in this paper, the key parts of JPF's API are: (i) configurable scheduling model and (ii) choice generators. Configurability of thread scheduling allows to use a domain-specific scheduling algorithm (e.g., one based on thread priorities) and, in particular, to restrict concurrency in various ways. The mechanism of choice generators unifies all possible causes for a branch in the state space of a Java program, including thread scheduling and non-deterministic data value choice. A specific instance of a choice generator is associated with each state — it maintains the list of enabled and unexplored transitions leading from the state. The choice generator API also provides means for altering the set of enabled and unexplored transitions; for example, it is possible to select specific transitions that should be explored.

5 Restrictions of Concurrency in Model Checking Java Programs with Java PathFinder

The key idea behind our approach is that it is not necessary to check the behavior of a Java program under all possible sequences of thread scheduling choices that are allowed by the concurrency model and semantics of Java, when Java platforms restrict concurrency in such a way that some of the sequences of thread scheduling choices cannot occur during execution of a Java program.

It follows from the overview of current Java platforms (Sect. 2) and JPF (Sect. 4) that no significant restriction of concurrency in model checking with JPF is possible in case of Java platforms for server and desktop systems, which typically involve multiple processors and use native threading with time preemption-based scheduling. Therefore we focus only on Java platforms for embedded systems, where significant restrictions of concurrency are possible. Such platforms typically involve only a single processor, in which case the green threading model is used, or a very low number of multiple processors, in which case the quasi-preemptive threading can be used. In both cases, time preemption-based scheduling is not used and thus only specific program points are considered as thread yield points. As indicated in Sect. 2.1, there are also Java platforms for

real-time embedded systems that strictly enforce thread priorities and perform priority-based thread scheduling. However, in this paper we focus only on Java platforms for non-real-time embedded systems that involve JVMs like CLDC [5] — we assume (i) that all threads in a Java program have the same priority during the whole lifetime of a program and (ii) that bytecode instructions corresponding to back-branches, method prologues and method epilogues are not used as thread yield points by the JVM. We leave support for thread priorities and priority-based scheduling to our future work (see the end of Sect. 7 for details).

We propose to use two platform-specific restrictions of concurrency for the purpose of addressing state explosion in model checking of Java programs for such platforms with JPF — specifically, we propose (i) to bound the maximal number of threads that can run in parallel, and (ii) to consider only specific bytecode instructions and calls of specific methods as thread yield points.

Maximal number of parallel threads. The rationale behind this restriction is that if there are N processors in a Java platform and M threads running in parallel, such that $M > N$, only at most N threads can compete for a particular shared variable at a specific moment in time (exactly N threads can compete only if there are N active threads at the moment). Therefore, if this restriction is applied, JPF has to explore only those thread interleavings that correspond to parallel execution of some N threads at most at each point in the program's running time. In particular, it is not necessary to explore those interleavings that involve suspending of a thread T_i, $1 \leq i \leq N$, and scheduling of another thread T_j, $N + 1 \leq j \leq M$, when an access to a shared variable occurs in T_i.

Thread yield points. Since we focus only on Java platforms for non-real-time embedded systems that involve JVMs like CLDC, for the purpose of model checking of Java programs with JPF it is sufficient to consider as thread yield points only those bytecode instructions and method calls, whose effects are visible to other threads and may therefore influence their behavior. Effects visible to other threads are, most notably, changes of shared variables' values and changes of threads' status (including synchronization). Therefore, the set of program points considered as thread yield points by JPF has to include (i) bytecode instructions that correspond to acquiring and releasing of monitors (locks) and (ii) calls of methods that change status of a thread (specific methods of the `Thread` class, and the `wait` and `notify` methods). If the platform involves multiple processors, JPF has to consider as thread yield points also bytecode instructions corresponding to accesses to shared variables — this is necessary in order to properly capture the contention for the shared variables among multiple threads running concurrently or truly in parallel.

Consequences on model checking with JPF. A consequence of application of both restrictions of concurrency in model checking with JPF is that the number of thread scheduling choices on any execution path in a checked Java

program is greatly reduced, which implies that the number of paths (branches) in the state space of the Java program is greatly reduced. Therefore, the whole reachable state space of a multi-threaded Java program is much smaller and thus model checking of such a program with JPF is less prone to state explosion. Note, however, that if there is a thread yield point (e.g., acquire of a monitor or call of `wait`) between each pair of accesses to shared variables in the program code, then the proposed restrictions would not help very much (even if the number of processors is set to 1) — the state space size would be the same as in the case of default JPF with no restriction.

Given the Java program on Fig. 2, JPF with both restrictions and the number of processors set to 1 may terminate transitions (i.e. suspend and schedule threads) only at program points corresponding to bytecode instructions for: entry to a `synchronized` block (source code lines 7, 14, 27 and 34), call of the `wait` method (lines 8 and 28), call of the `notify` method (lines 15 and 35), and start of a thread (lines 45 and 46). This means, for example, that the code at lines 11 and 12 will be executed atomically (in a single transition) by JPF. If the number of processors is set to 2, JPF may terminate transitions also at program points (bytecode instructions) corresponding to accesses to shared variables — this includes, in particular, accesses to the `buffer` variable (source code lines 8, 11, 12, 28 and 32).

JPF extension. We have implemented the proposed restrictions of concurrency in a JPF extension. The extension has two components: a custom choice generator, which maintains the mapping of threads to processors, and a custom scheduler, which creates an instance of the choice generator at each scheduling-relevant instruction (corresponding to a thread yield point or an access to a shared variable). An instance of the choice generator, which is associated with a particular state, determines the correct set of threads that can be scheduled at that state with respect to the JPF extension's configuration. The configuration consists of the number of processors in a Java platform (unlimited by default) and of a boolean flag that determines whether thread yield points should be attached only to selected program points (for green or quasi-preemptive threading) or to all program points (for time preemption).

6 Experiments

We have performed a number of experiments with our JPF extension in order to find how much the proposed platform-specific restrictions of concurrency help in addressing state explosion in model checking of multi-threaded Java programs with JPF. To be more specific, we performed two sets of experiments — the goal of the first set of experiments was to show how much the restrictions of concurrency reduce the size of the whole state space, and the goal of the second set of experiments was to show how much the restrictions reduce the time and memory needed to find concurrency errors.

All the experiments were performed on the Purdue Java Collision Detector (PCD), which is a plain-Java version of the Purdue Real-Time Collision Detector [18, 1] developed at the Purdue university as a benchmark for Java virtual machines. PCD is a non-trivial model (12Kloc in Java) of a multi-threaded Java application that could be run on Java platforms for embedded systems. The architecture of PCD is an instance of the classic producer-consumer pattern that involves three threads running in parallel — the main thread that starts other threads and waits till they finish (via `Thread.join`), simulator thread (producer), and detector thread (consumer). The simulator thread computes actual positions of physical objects (aircrafts) with respect to time and generates messages with information about positions of the objects, which it sends to the detector thread via a shared buffer. The detector thread performs the actual detection of collisions on the basis of information received from the simulator thread. The aspect of PCD's code that has the greatest influence on the size of its state space is the number of messages generated by the simulator thread and sent to the detector thread — the number of messages to be exchanged between the threads in a particular run of PCD can be specified via one of the PCD's configuration variables.

Configuration of each experiment consists of (i) the number of messages exchanged between the simulator and detector threads and (ii) the list of restrictions of concurrency that are applied. If the restriction of the maximal number of threads that can run in parallel is used, then also the number of processors in a platform has to be provided. We selected two relatively low numbers of messages in PCD — 5 and 10 — in order to make checking with JPF finish in reasonable time. As for JPF, we have used three different configurations:

- native threading with time preemption and no bound on the number of threads running in parallel (default in JPF),
- quasi-preemptive threading with two processors, and
- green threading with a single processor.

Note that both the restriction of the set of thread yield points and the bound on the number of threads running in parallel are applied in the latter two configurations. The difference is in the number of processors, which determines the bound on the number of threads running in parallel. For the purpose of experiments in the first set, we turned off the search for errors of any kind in JPF in order to let JPF traverse the whole state space of PCD and we also put a limit on the maximal running time of JPF and on the available memory — the limits were set to 5 days (432000 seconds) and 3 GB, respectively.

The results of experiments are listed in Tables 1 and 2. We measured the following characteristics of JPF runs: time in seconds, memory in MB and total number of states. If checking with JPF exceeded the time limit in case of experiments in the first set, then the value of the 'Time' column is set to "> 432000" (number of seconds in 5 days) and the value of the 'Memory' column shows the peak in memory usage up to the point of limit's exceeding. Similarly, if checking with JPF run out of available memory, then the value of the 'Memory' column

Restrictions	Time (s)	Mem (MB)	States
5 messages			
default JPF (no restriction)	> 432000	1967	19992569
quasi-preemptive threading + two processors	2317	1535	102482
green threading + single processor	127	740	5217
10 messages			
default JPF (no restriction)	> 432000	1725	19986412
quasi-preemptive threading + two processors	41803	2249	2016072
green threading + single processor	3369	1336	162093

Table 1. Results of experiments on PCD: traversal of the whole state space

Restrictions	Time (s)	Mem (MB)	States
5 messages			
default JPF (no restriction)	38	487	1260
quasi-preemptive threading + two processors	16	396	236
green threading + single processor	17	401	236
10 messages			
default JPF (no restriction)	42	496	1420
quasi-preemptive threading + two processors	21	416	391
green threading + single processor	20	411	391

Table 2. Results of experiments on PCD: search for concurrency errors

is set to "> 3 GB" and the value of the 'Time' column shows the time of running out of memory. The 'States' column shows the number of states traversed up to the moment of exceeding the limit in both cases. The concurrency errors discovered by experiments in the second set were race conditions in accesses to the shared buffer that were already present in the code of PCD.

7 Evaluation and Related Work

The results of experiments on PCD (Sect. 6) show that the proposed platform-specific restrictions of concurrency help quite significantly in addressing state explosion in model checking of multi-threaded Java programs for embedded systems with JPF. Specifically, the restrictions reduce the size of the state space of such Java programs by an order of magnitude and they also reduce the time needed to discover concurrency errors in the code. In case of really complex multi-threaded Java programs, for which model checking with JPF may still not be realistic due to state explosion, the proposed restrictions at least make it possible for JPF to explore a larger part of the programs' state space and thus also to discover more errors in the code. This is very important in particular for programs used in critical systems, where the costs of errors (and fixing of errors in already deployed systems) are typically very high.

An inherent drawback of the proposed restrictions is that results of model checking with JPF are specific to a particular platform. A given Java program

has to be verified separately for each Java platform on which it will be deployed, since a run of JPF will discover only those errors that may occur on a particular Java platform characterized by the specific configuration of the restrictions. Nevertheless, this is not a big issue in the domain of embedded systems, since both the hardware and software configurations of an embedded platform are typically specified in advance — this is the case especially for critical embedded systems.

Also, the restrictions of concurrency are not specific to Java programs and neither to model checking with JPF — the same or similar platform-specific restrictions could be applicable to programs in any mainstream programming language that supports multi-threading (e.g., C# and C), and they could be implemented in any model checker for such languages with the goal of improving performance and scalability of verification.

Related work. Many techniques for addressing state explosion in model checking were proposed and implemented over the years — an extensive overview of the techniques can be found in [17]. Focusing on model checking of multi-threaded programs, the techniques most closely related to our approach can be divided into two groups:

1. techniques for reduction of the number of states and paths in the state space that have to be explored in order to check the behavior of a given program under all possible sequences of thread scheduling choices, and
2. techniques for efficient discovery of some errors only that are based on traversal of a part of the state space.

The first group of techniques includes, for example, partial order reduction [8] and thread symmetry reduction [13], while the second group includes heuristics for state space traversal [10] (directed model checking) and techniques based on bounding of the number of thread context switches [19, 20, 16]. All these techniques are complementary to our approach — they can be applied in combination with the platform-specific restrictions of concurrency in order to mitigate state explosion even further.

We are, however, not aware of any approach or technique that attempts to exploit concurrency-related characteristics and properties of a specific platform (runtime environment) for programs with the goal of addressing state explosion. The only approach in a similar direction that we are aware of is memory model-sensitive model checking [11, 6], which aims to improve the completeness of model checking with respect to behavior of state-of-the-art compilers for modern programming languages (Java, C#). The key idea is to also take into account possible reorderings of operations that are allowed by the memory model and/or concurrency model of a language — typically, reorderings of writes to variables are performed by compilers for the purpose of performance optimization.

Future directions. Although the proposed restrictions of concurrency help quite significantly in addressing state explosion in model checking of multi-threaded Java programs with JPF, still there is much space for further optimization. We have identified several additional platform-specific restrictions of

concurrency that could reduce the state space size of multi-threaded Java programs even further, thus making model checking with JPF even more scalable. The additional restrictions can be divided into two groups: (i) restrictions imposed by the Real-Time Specification for Java (RTSJ) [4] and (ii) more realistic modeling of time preemption.

Ad (i) The key aspect of concurrency imposed by RTSJ is strict enforcement of thread priorities, which means that a runnable thread with the highest priority always has to be scheduled at each thread scheduling choice point. The number of thread scheduling sequences can be significantly reduced in this way. On the other hand, it is necessary to capture asynchronous unblocking of threads with high priorities (e.g., when a higher priority thread was blocked in an attempt to read data from a file and the data become available) and dynamic changes of priorities during a program run (e.g., via the `setPriority` method of the `Thread` class). Moreover, bytecode instructions corresponding to back-branches, and probably also bytecode instructions corresponding to method prologues and epilogues, would have to be considered as thread yield points by JPF in order to faithfully capture the concurrency-related behavior and characteristics of Java platforms for real-time systems (e.g., such that involve Purdue OVM [3]). Another option related to real-time programs is to consider only those sequences of thread scheduling choices that are determined as valid with respect to temporal constraints (e.g., real-time deadlines) by WCET analysis [25]. Technically, the restriction to valid sequences of thread scheduling choices can be implemented by a specific choice generator and the WCET analysis can be performed by an external tool.

Ad (ii) A possible approach to more realistic modeling of time preemption in JPF is to suspend a thread at a Java bytecode instruction with visible effects on shared variables only when the thread has run out of its time slot. This way it could be possible to model-check complex server-side business applications in Java (i.e. such as those that typically run on Java platforms for server and desktop systems), which are characteristic by a great number of accesses to shared variables. Nevertheless, a prerequisite for this optimization is support for modeling of real time and execution cost of bytecode instructions in JPF.

Restrictions of concurrency of some form could also be applied for the purpose of efficient model checking of programs that use actor concurrency and huge numbers of lightweight threads (JVM-level threads). This is, for example, the case of programs written in the Scala language [21], which are compiled to Java bytecode and run on JVMs, or Java programs using the Kilim library [22].

8 Conclusion

In this paper, we proposed a technique for addressing state explosion in model checking of multi-threaded Java programs for embedded systems, which is based on restrictions of concurrency and thread scheduling that are common in current Java platforms for embedded systems. The technique is complementary to existing approaches for addressing state explosion — it aims to reduce the size of

the whole state space of a given program, while most of the existing techniques aim to reduce the number of states and paths in the state space that have to be explored by a model checker to check all behaviors of a program. We have implemented the technique as an extension to Java PathFinder and performed several experiments on Purdue Collision Detector, which is a non-trivial multi-threaded Java program, in order to show the benefits of the technique. The results of our experiments show that the proposed restrictions (i) reduce the state space size of Java programs by an order of magnitude and (ii) reduce the time needed to discover concurrency errors.

While the proposed technique helps in addressing state explosion quite significantly, there is a number of additional platform-specific restrictions of concurrency and optimizations that could be used to mitigate state explosion even further. We plan to focus especially on restrictions and optimizations related to Real-Time Specification for Java (RTSJ), since software for critical embedded systems often has real-time characteristics.

Acknowledgments. This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and by the Grant Agency of the Czech Republic project 201/08/0266.

References

1. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java, In Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP'06), LNCS, vol. 4067, 2006.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software, In Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'04), LNCS, vol. 3114, 2004.
3. A. Armbuster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A Real-Time Java Virtual Machine for Avionics, ACM Transactions on Embedded Computing Systems, vol. 7, no. 1, 2007.
4. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Java Series. Addison-Wesley, 2000.
5. CLDC HotSpot Implementation Virtual Machine, White Paper, Sun Microsystems, available at `http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf` (accessed in March 2009).
6. A. De, A. Roychoudhury, and D. D'Souza. Java Memory Model aware Software Validation, In Proceedings of the 8th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'08), ACM Press, 2008.
7. M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework, In Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05), LNCS, vol. 3576, 2005.
8. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem, LNCS, vol. 1032, 1996.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, 3rd Edition, Addison-Wesley, 2005.

10. A. Groce and W. Visser. Heuristics for Model Checking Java Programs, International Journal on Software Tools for Technology Transfer, vol. 6, no. 4, 2004.
11. T. Q. Huynh and A. Roychoudhury. A Memory Model Sensitive Checker for C#, In Proceedings of the 14th International Symposium on Formal Methods (FM'06), LNCS, vol. 4085, 2006.
12. IBM J9 Java Virtual Machine, `http://wiki.eclipse.org/index.php/J9` (accessed March 2009).
13. R. Iosif. Symmetry Reductions for Model Checking of Concurrent Dynamic Software, International Journal on Software Tools for Technology Transfer (STTT), vol. 6, no. 4, 2004.
14. Jikes RVM (Research Virtual Machine), `http://jikesrvm.org` (accessed in March 2009).
15. T. Lindholm and F. Yellin. The Java Virtual Machine Specification, 2nd Edition, Prentice Hall, 1999.
16. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), ACM Press, 2007.
17. R. Pelanek. Fighting State Space Explosion: Review and Evaluation, In Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08), to appear in LNCS.
18. F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java Scoped Memory: Design patterns and Semantics, In Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), IEEE CS, 2004.
19. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software, In Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), LNCS, vol. 3440, 2005.
20. I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs, In Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05), LNCS, vol. 3576, 2005.
21. The Scala Programming Language, `http://www.scala-lang.org/` (accessed in March 2009).
22. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java, In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), LNCS, vol. 5142, 2008.
23. Sun Java SE HotSpot, Sun Microsystems, `http://java.sun.com/javase/technologies/hotspot/` (accessed in March 2009).
24. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, 2003.
25. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenstrom. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools, ACM Transactions on Embedded Computing Systems, vol. 7, issue 3, 2008.